NitroSketch: Robust and General Sketch-based Monitoring in Software Switches

Zaoxing Liu[1], Ran Ben-Basat[2], Gil Einziger[3], Yaron Kassner[4],
Vladimir Braverman[5], Roy Friedman[4], Vyas Sekar[1]
[1]Carnegie Mellon University, [2]Harvard University, [3]Ben-Gurion University
[4]Technion, [5]Johns Hopkins University

ABSTRACT

Software switches are emerging as a vital measurement vantage point in many networked systems. Sketching algorithms or *sketches*, provide high-fidelity approximate measurements, and appear as a promising alternative to traditional approaches such as packet sampling. However, sketches incur significant computation overhead in software switches. Existing efforts in implementing sketches in virtual switches make sacrifices on one or more of the following dimensions: performance (handling 40 Gbps line-rate packet throughput with low CPU footprint), robustness (accuracy guarantees across diverse workloads), and generality (supporting various measurement tasks).

In this work, we present the design and implementation of NitroSketch, a sketching framework that systematically addresses the performance bottlenecks of sketches without sacrificing robustness and generality. Our key contribution is the careful synthesis of rigorous, yet practical solutions to reduce the number of per-packet CPU and memory operations. We implement NitroSketch on three popular software platforms (Open vSwitch-DPDK, FD.io-VPP, and BESS) and evaluate the performance. We show that accuracy is comparable to unmodified sketches while attaining up to two orders of magnitude speedup, and up to 45% reduction in CPU usage.

CCS CONCEPTS

• **Networks** → **Network monitoring**; **Network measurement**;

KEYWORDS

Sketch, Software Switch, Virtual Switch, Flow Monitoring, Sketching Algorithm

ACM Reference Format:
Zaoxing Liu, Ran Ben-Basat, Gil Einziger, Yaron Kassner, Vladimir Braverman, Roy Friedman, Vyas Sekar. 2019. NitroSketch: Robust and General Sketch-based Monitoring in Software Switches. In *SIGCOMM '19: 2019 Conference of the ACM Special Interest Group on Data Communication, August 19–23, 2019, Beijing, China.* ACM, New York, NY, USA, 17 pages. https://doi.org/10.1145/3341302.3342076

SIGCOMM '19, August 19–23, 2019, Beijing, China

ACM ISBN 978-1-4503-5956-6/19/08…$15.00
https://doi.org/10.1145/3341302.3342076

1 INTRODUCTION

Traffic measurements are at the core of network functions such as traffic engineering, fairness, load balancing, quality of service, and intrusion detection [3, 4, 12, 28, 36, 38, 44, 49, 65, 74, 75]. While monitoring on dedicated switching hardware continues to be important, measurement capabilities are increasingly being deployed on *software switches* with the transition to virtualized deployments and "white-box" capabilities (e.g., Open vSwitch [67], Microsoft Hyper-V [62], Cisco Nexus 1000V [22], FD.io VPP [34], and BESS [39]).

Naturally, we want measurement capabilities to run at high line rates but impose a small resource footprint to avoid constraining the main network functions and services that run atop the switch. In this respect, *sketching algorithms* appear promising as they provide rigorous accuracy guarantees and support a variety of measurement tasks. Example tasks include per-flow frequency estimation [17, 27], tracking heavy hitters [10, 61, 63, 68, 69], detecting hierarchical heavy hitters [8, 9, 20, 26], counting distinct flows [6, 7, 55], estimating frequency moments [5], and change detection [51, 55].

In practice, the performance of sketching algorithms in software switches is far from ideal [1, 2, 43]. For instance, Count-Min Sketch [27] and Count Sketch [17] do not achieve 10M packets per second (Mpps), and UnivMon [55, 56] runs at <2Mpps (Figure 2). In retrospect, this may be unsurprising as sketches are theoretically designed for memory efficiency and are not optimized for CPU consumption, which is a more critical bottleneck in software switches.

In this context, recent efforts have sought to optimize the performance of software sketches. These include SketchVisor [43], Hashtable-based monitoring [1, 2], R-Hierarchical Heavy Hitters (R-HHH) [8], and ElasticSketch [73]. Although these efforts have made significant contributions, they compromise on one or more dimensions (Section 2), as summarized in Table 1: (1) *Performance* in handling modern line rates with minimal CPU and small memory (e.g., 40Gbps in a single thread); (2) *Robustness* in achieving provable worst-case accuracy guarantees (e.g., arbitrary workload distributions and an arbitrary number of flows); and (3) *Generality* to support a variety of measurement tasks. Specifically, SketchVisor relies on the skewness of the workload to achieve accuracy and speedup, and thus does not work well for heavy-tailed workloads; in many cases, it cannot even attain 10Gbps line-rate. ElasticSketch lacks robustness as its accuracy guarantee breaks for various tasks such as distinct flows and entropy if the workload contains too many flows. While R-HHH achieves 10Gbps (14.88Mpps) line rate and provides accurate results, it supports a specific task of hierarchical heavy hitters and cannot be extended to other tasks.

Our motivation question is simple to state: *Can we develop a performant sketching framework in software switches that does not sacrifice robustness and generality?*

Solutions	Category	OVS Packet Rate	Robustness	Generality
SketchVisor [43]	Sketch	1.7Mpps	X	✓
R-HHH [8]	Sketch	14Mpps	✓	X
ElasticSketch [73]	Sketch	5Mpps	X	✓
Small-HT [1]	Non-Sketch	13Mpps	X	✓

Table 1: Summary of existing solutions on software platforms.

We answer this question by presenting **NitroSketch** — a software sketching framework that optimizes performance, provides accuracy guarantees, and supports a variety of sketches. Our design is inspired by the observation that many sketches incur similar performance bottlenecks as they share a similar data structure of multiple counter arrays updated with independent hash functions. With that in mind, we analyze the key bottlenecks of sketches in software and identify three fundamental issues (Section 3): (1) several expensive per-packet hash computations; (2) multiple random memory accesses and arithmetic operations to update counters; and (3) additional overheads in tracking some heavy flow keys.

To understand the intuition behind NitroSketch, let us consider two natural strawman solutions to tackle these bottlenecks: (a) reducing the number of hashes and using a single hash-indexed array; or (b) retaining the sketch structure but reducing the number of processed packets via uniform sampling [13, 25, 47]. Unfortunately, to achieve comparable accuracy bounds, both approaches require a large amount of memory. Given the random access patterns of sketch counter updates, this actually hurts performance due to cache misses. Furthermore, we observe that even a single per-packet hash operation can be expensive. While sampling avoids this hashing operation, it still incurs per-packet coin flips. Finally, uniform sampling entails an increase in the convergence time before the sketches can produce useful and accurate results.

NitroSketch builds on insights from analyzing these two basic building blocks but avoids their shortcomings. Our first insight is that retaining the multiple independent hash arrays is vital for keeping a small cache-resident memory footprint in the memory-accuracy trade-off. Given this observation, our second insight is a more effective sampling strategy to reduce the number of per-packet operations. Rather than sample packets to process, we sample the counter arrays which need to be updated, which we show to be more efficient than packet sampling. To avoid the per-packet coin flip cost, we use a simple-yet-effective geometric sampling optimization [30, 50]. An additional benefit of these optimizations is that it incurs fewer updates to the flow-key data structure. Finally, while the convergence issue is an unavoidable concern for any sampling solution, we develop rigorous ways for *adapting* the sampling rate to always produce accurate results.

NitroSketch is a general framework that applies to *any* sketch structure that follows a canonical workflow of using multiple independent hashes and counter arrays. Indeed, we support a number of popular sketching algorithms atop NitroSketch (e.g., [17, 27, 51, 55]) and boost their packet processing performance without sacrificing their theoretical robustness guarantees.

We implement a NitroSketch prototype and integrate it with popular software switches, including Open vSwitch-DPDK (OVS-DPDK) [67], FD.io-VPP [34], and BESS [39]. We implement additional system optimizations such as buffered counter updates and SIMD-style parallelism [45]. We evaluate NitroSketch on OVS-DPDK, VPP, and BESS using a range of traces [11, 14, 15, 58] on commodity servers with 40GbE NICs. We demonstrate that NitroSketch handles 40GbE on software switches with a single thread. Compared to NetFlow/sFlow [21, 71], NitroSketch achieves better accuracy and uses significantly less memory when evaluated with the same sampling rate. Compared with SketchVisor [43], NitroSketch runs dramatically faster in CPU (e.g., >83Mpps vs. <7Mpps), or uses significantly less CPU to achieve the same throughput and yields more accurate results after convergence. Our in-memory benchmarks also suggest that NitroSketch is a promising option for future virtual switches handling >40Gbps line rates.

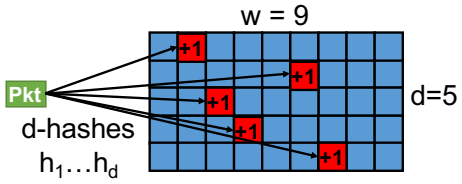


Figure 1: Count-Min Sketch example.

We begin with background and related work in the next section. We then present the bottleneck analysis of sketches in §3, the design that addresses the bottlenecks in §4, and the analysis of NitroSketch algorithms in §5. After implementation (§6) and evaluation (§7), we summarize NitroSketch and conclude in §8.

2 RELATED WORK AND MOTIVATION

We begin by providing background on *sketching algorithms* and motivate the need for performance improvements in software switches. We then discuss previous attempts to improve the performance of software sketch implementations.

Sketches. Sketches are useful for many network measurement tasks such as: (1) Heavy Hitter Detection to identify flows that consume more than a threshold α of the total traffic (e.g., based on the packet or byte counts) [17, 27, 55, 61, 69]; (2) Change Detection to identify flows that contribute more than a threshold of the total capacity change over two consecutive time intervals [51, 55, 68]; (3) Cardinality Estimation to estimate the number of distinct flows in the traffic [7, 55]; (4) Entropy Estimation to approximate the entropy of different header distributions (e.g., [52]); and (5) Attack Detection to identify a destination host that receives traffic from more than a threshold number of source hosts [76]. More recent work such as UnivMon [55] also suggests extensions to support a range of these tasks, rather than a specialized sketch per task.

At a high-level, sketches are approximate data structures to estimate various statistics of a given streaming workload. A sketch data structure typically consists of some arrays of counters, and uses a set of independent hash functions to update these counters. To illustrate the common structure of sketches, we consider the Count-Min Sketch (CMS) [27] as an example. As depicted in Figure 1, CMS maintains d arrays of w counters each. On its *Update* procedure, it computes d independent hash values from the packet's flow identifier (e.g., 5-tuple). Each such hash provides an offset within one of the d arrays, and CMS then increases the corresponding counters. This data structure can then be used for a number of estimation tasks. Specifically, the estimate for a given flow's size is the minimum value among the corresponding counters for this flow. Using these

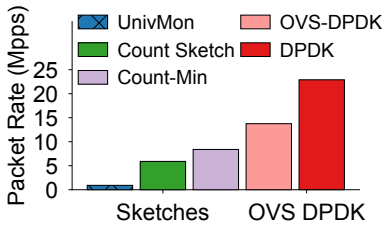


Figure 2: Packet rates of Sketches, OVS, and DPDK.

per-flow estimates, we can compute other derived statistics such as quantiles or other traffic summaries.

Sketches are typically designed to reduce the memory usage of measurement tasks and to achieve guaranteed fidelity for the estimated statistics. Space efficiency is crucial in hardware as high-speed memory (e.g., SRAM) is at a premium [41, 57, 72] and line-rate processing is guaranteed once data structure fits in the memory.

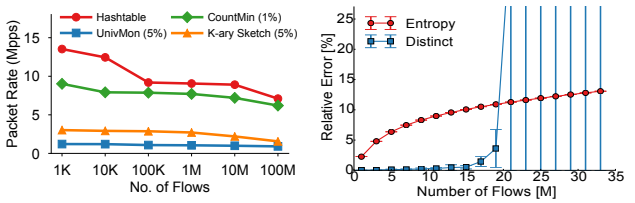
Sketch performance in software switches. In a software switch context, however, memory may be a less severe constraint. A more critical goal is to support high line-rates with a low computation footprint (e.g., using single CPU core). This low footprint is vital to ensure that other concurrent services (e.g., virtual machine instances) can make maximal use of the available resources.

To analyze the performance, we profiled a single-thread OVS-DPDK and the result shows that the CPU room for running added measurement algorithms is limited. Figure 2 shows the measured performance of software implementation of sketches atop OVS-DPDK. (We provide more details on our testbed setup in Section 7.) We configure the memory allocation of the sketches as recommended in the respective papers. For Count-Min Sketch, we set 5 rows of 1000 counters; for UnivMon, its Count Sketch components have five rows of 10000 counters each. We observe that sketches impose significant overhead and cannot meet 10Gbps line-rate with a single CPU core for a worst-case workload (i.e., 14.88Mpps with 64B packets). Even the light-weight Count-Min Sketch [27] is far away from line rate processing.

Existing solutions. Indeed, we are not alone (nor the first) in pointing out these limitations, and these results mirror the measurements from other efforts [1, 2, 43, 73].

To tackle this performance issue, prior efforts have taken one of two approaches: (1) eschewing sketches altogether in favor of simpler data structures or (2) heuristic fixes to sketches. Both approaches have fundamental limitations and do not meet our goals of performance, robustness, and generality.

Alipourfard et al. [1, 2] suggest that small hash tables can suffice for software switches as in skewed workloads (i.e., a small number of flows carry most of the traffic volume) we can accurately monitor all flows. Indeed, we can expect higher accuracy when the traffic is only composed of a small number of elephant flows. However, this small hash table approach is not robust since it relies on the skewness of workloads to achieve good performance. Skewness is a strong assumption since traffic distributions are sometimes heavy-tailed (i.e., there are a significant number of flows with non-trivial volumes). Further, the traffic distribution may vary at times, such as a port scan. Even if we accept these limitations, updating a hash table still requires a per-packet hash computation and counter update.



(a) Throughput vs. #flows on 1 core OVS-DPDK.

(b) ElasticSketch (2.7MB) Accuracy vs. #flows. in a malware trace [58].

Figure 3: Prior approaches are not performant or robust to a large number of flows.

The performance of such actions is problematic once the table does not fit in the last level cache.

SketchVisor [43] uses a *fast path* to expedite the packet processing of the measurement on software switches when there is a queue buildup before the sketches that process packets on a *normal path*. The fast path maintains a hash table of k entries and each entry has three different counters that are used for deciding whether to run an update or a kick-out operation. The normal path contains standard, slower sketches. Although the fast path can achieve higher packet rate, it is less precise than the normal path as accuracy degrades significantly when the majority of packets go to the fast path. Therefore, it is not a robust solution.

ElasticSketch [73] splits the packet processing into two parts: a *heavy part* and a *light part*. The heavy part is a table of hash buckets, where the heavy flows are stored and evicted to the light part when necessary; and the light part is a Count-Min Sketch (CMS), which tracks the mice flows. The difference from SketchVisor is that ElasticSketch has an eviction policy that further reduces the worst-case packet operations. This design works well when the number of flows is not large but using CMS solely in the light part loses the generality for some measurement tasks that cannot be handled with L_1[1] guarantee from CMS. As depicted in Figure 3, these approaches cannot maintain acceptable accuracy and throughput when the number of flow increases (e.g., for 20M flows the hash table's throughput reduces to less than 10Mpps and the error of ElasticSketch exceeds 100% due to the overflow on its linear counting when estimating the distinct flows). As depicted in Figures 11 and 12 in the evaluation, the relative errors of the sketches that come with better-than-L_1 guarantee will not increase significantly.

R-HHH [8] reduces the update time of a deterministic Hierarchical Heavy Hitters algorithm [64] to O(1) by choosing one random prefix per-packet to update. Although their algorithm is robust for a specific measurement task, it does not support other measurement tasks and is therefore not general.

In summary, existing solutions trade off robustness or generality for improved performance as summarized in Table 1. Our goal is to improve the performance without losing the robustness or generality.

3 BOTTLENECK ANALYSIS

We start by systematically understanding the bottlenecks of running sketching inside software switches before we design optimizations. For this analysis, we use the same testbed as described in Section 7

[1] $L_1 \triangleq \sum f_x$ refers to the first norm of the flow frequency vector of the workload.

Func/Call Stack	Description	CPU Time
xxhash32	hash computations	37.29%
__memcpy	memcpy and counter update	15.91%
heap_find	heap operation	10.71%
univmon_proc	packet copy and cache	8.02%
heapify	heap maintenance	4.91%
miniflow_extract	retrieve miniflow info	2.93%
recv_pkts_vecs	dpdk packet recv	2.71%

Table 2: CPU hotspots on UnivMon with OVS-DPDK.

with min-sized packets as a worst-case scenario to stress the throughput of software sketch implementations. We observe similar trends with other workloads varying packet size distributions.

For the following analysis, we instrument a single thread OVS-DPDK with UnivMon sketch as a representative example sketch. We use Intel VTune Amplifier [46] to identify performance bottlenecks. Our observations are qualitatively similar for other sketches as well, but not shown for brevity. Table 2 summarizes the results.

Bottleneck 1: ***Sketches perform multiple independent hash computations per packet.***

From the profiling results in Table 2, we see an obvious bottleneck due to hash function calls, which consumes $\approx$ 40% of the CPU. For the ease of comparison later, if we denote the cost of each hash operation as H, for a sketch using d_1 hash functions, there is a $d_1 \cdot H$ computation cost we incur.

Bottleneck 2: ***Sketches entail multiple counter updates with memory copy and arithmetic operations.***

The second significant bottleneck after the hashing we see in the table is the memory operations. More specifically, sketches also update a number of (e.g., 3 to 10) counters based on the computed hash values, which in turn entail memory copy operations. If we denote the cost of the counter update operation as C and there are d_2 counter updates per packet, we can define this bottleneck as $d_2 \cdot C$.

Bottleneck 3: ***Sketches maintain extra data structures for tracking "heavy" flows that incur expensive per packet operations.***

Along with the counters arrays, sketches use additional data structures for bookkeeping. For instance, when tracking top K heavy hitters, users may want to know largest flows. In such cases, we can use a heap to store the flow keys. As shown in Table 2, maintaining a heap of top keys is not cheap. Let the cost of the per-packet operation of updating such a heap be denoted as P.

Summary. Based on these observations, a typical sketch implementation entails $\Theta(d_1 \cdot H + d_2 \cdot C + P)$ *per-packet operations*. Given this, we revisit prior attempts to improve the performance of the sketches and see how they tackle these costs.

SketchVisor [43] partially addresses the bottlenecks by proposing an improved Misra-Gries algorithm [63] to simplify the per-packet per packet operations in their front-end. SketchVisor reduces the amortized per-packet cost to $1H, 1C, 1P$ ($d_1H, d_1C, 1P$ in worst case), which can be much better than the original sketch implementations. However, as we will see, with min-sized packets even this is a high cost and cannot achieve a 10Gbps line rate. Perhaps more importantly, the speedup comes at the cost of robustness when the majority of packets are processed in the front-end, where the front-end Misra-Gries algorithm is not as accurate as other sketches.

ElasticSketch [73] separates the processing of elephant and mice flows, similarly as SketchVisor. Their approach mitigates the bottleneck operations to $1H, 1C, 1P$ per-packet even in worst-case, which improves performance. However, this per-packet complexity is still significant for software switches. Moreover, their back-end structure is a Count-Min Sketch, which only provides L_1 (norm) accuracy guarantees. This implies that the accuracy guarantees do not hold for complex functions such as entropy and L_2.

4 NITROSKETCH DESIGN

We first discuss strawman alternatives and their limitations. We then explain the key design ideas underlying NitroSketch.

4.1 Strawman Solutions and Lessons

As described in Section 3, the performance bottlenecks arise from hash calculations (H), counter updates (C), and priority queue (or heap) updates (P). We discuss the following strawman ideas to address these bottlenecks:

Strawman1: Reduce the number of hash functions and arrays. To reduce the number of bottleneck operations to $1H, 1C, 1P$, we can consider forming a one-array sketch. However, to retain the original accuracy guarantee, we need to increase the number of counters exponentially. For example, Count Sketch requires $O(\epsilon^{-2} \log \delta^{-1})$ counters to provide an ϵ additive error with $1 - \delta$ probability. This approach requires $O(\epsilon^{-2}\delta^{-1})$ to match this accuracy. In practice, when $\delta = 0.01$, this suggestion increases memory by $\approx$ 50×. Second, this approach may not be fast enough as we still perform one hash calculation, one counter copy operation, and one heap update operation per packet as $1H, 1C, 1P$. Our evaluation shows that doing so achieves 10G line-rate only when the sketch is Last Level Cache (LLC) resident. However, large memory increase implies that the sketch's LLC residency is affected and the performance degrades.

Lessons learned: We learn two things from Approach #1: (1) In software implementations, the memory usage of sketches should be moderate to fit into LLC to gain the best performance; (2) Even updating a single counter with one hash per-packet (i.e., $1H, 1C, 1P$) is non-trivial overhead under high packet rates (e.g., >15 Mpps), and we need to optimize it further.

Strawman2: Run sketch only over sampled packets. Uniform sampling is popular in estimating statistics in a database [13, 25, 47]. All sampling-based methods only provide accuracy guarantees once the measurement is long enough. We define the waiting time until accurate results are achieved as "convergence time". By using packet sampling, the number of packets that go to the sketch is reduced, which reduces the number of per-packet operations.

However, this approach comes with the following limitations: (a) First, we need a more accurate (larger) sketch to compensate for the accuracy loss that results from the sampling. As shown in Appendix B, if we set the error ϵ, sampling rate p, measurement length m, and $1 - \delta$ probability, the memory usage is $\Omega\Big(\epsilon^{-2}p^{-1} \log \delta^{-1} + \epsilon^{-2}p^{-1.5}m^{-0.5} \log^{1.5} \delta^{-1}\Big)$. In practice, this results in $60MB$+ memory, which likely results in a loss of cache residency. (b) Second,

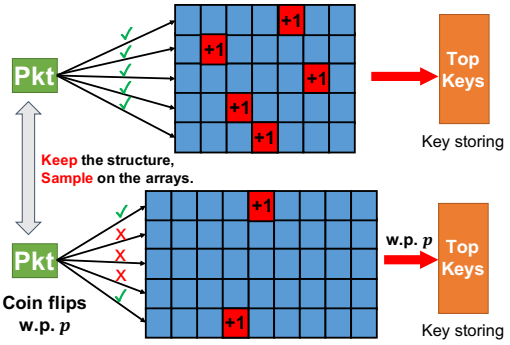


Figure 4: Idea A: Counter array sampling to reduce per-packet bottleneck operations.

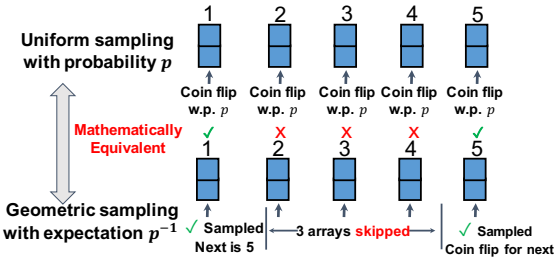


Figure 5: Idea B: Sampling the counter arrays with geometric sampling to avoid per packet PRNG. In this way, we don't need "coin flips" every time.

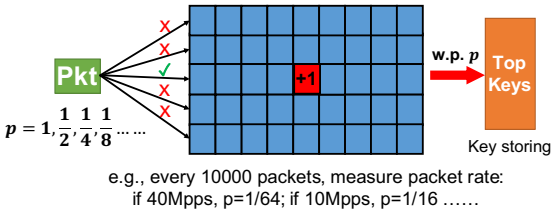


Figure 6: Idea C: Adjusting geometric sampling rates based on (estimated) packet arrival rate for faster convergence.

flipping a coin requires a random number generation and doing it for each packet has considerable overhead. Based on our test, a single coin flip per-packet still prevents us from supporting 40Gbps (≈ 60Mpps). (c) Third, small packet sampling rates result in long convergence time, which is a potential issue for short-time measurements [60].

Lessons learned: Sampling offers the potential for significant speedup as it reduces the per-packet computation to less than $1H, 1C, 1P$. However, it still has overheads such as random number generation and cache misses which prevent it from scaling to higher line rates (e.g., 40G with min-sized packets). Furthermore, there is a trade-off between the sampling probability and the convergence time.

4.2 Key Ideas

In designing NitroSketch, we use the lessons learned from the bottleneck analysis and the strawman solutions. We now describe our key ideas and highlight how they improve the performance.

Idea A: Keep the multi-array structure as original sketches and sample on the independent counter arrays.
Sketches use multiple high-quality and independent (usually require pair-wise or even four-wise independent) hash functions that amplify the success probability and play a significant role in their accuracy guarantees. Thus, we retain the same multi-array structure and employ the same set of hash functions as the underlying sketch. For each packet, we flip a coin *for each array* to decide if we need to update a counter in that array, as shown in Figure 4. Doing so reduces the per-packet hashes and counter updates to less than one by using appropriate sampling rate p (e.g., $p < \frac{1}{5}$ in Figure 4). Compared with Strawman2, this idea also needs to increase the number of counters but it is more space efficient in order to be cache resident, as we will show in Section 5. However, this idea of counter array sampling requires even more coin flips than packet sampling. Thus, Idea B refines this idea to reduce the number of coin flips.

Idea B: Avoid coin flips by drawing a single geometric sample for all the counter arrays.
The straightforward realization of Idea A needs one coin flip per counter array when processing each packet. To avoid this, we consider a simple but effective implementation that draws samples from a geometric distribution to decide (i) which counter array will be updated next (ii) how many packets to skip until that update. We realize these two decisions by randomizing how many "arrays" to skip until the next update.

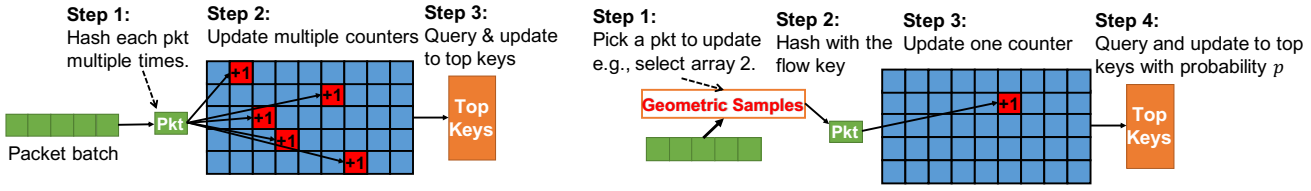
We illustrate Idea B in Figure 5. The sketch has five counter arrays, and let us assume that Array 1 was just updated. We draw a geometric variable, with success probability p that tells us how many arrays to skip. If the sample tells the next update is four arrays away, we skip the following three arrays and update Array 5 (with the same packet). If the randomization tells us to skip six arrays, then we will skip further updates for the current packet and update Array 3 during the next packet. Thus, we use a single geometric variable for each sampled counter array and minimize the coin-toss overheads.

Idea C: Adaptive sampling based on packet arrival rate to reduce convergence time.
There is a trade-off between convergence time and the packet update speed. The more packets we skip, the longer we need to wait before providing an accuracy guarantee. When using a static counter array sampling probability, we should determine p to be sufficiently small for the highest possible packet rate. However, in that case, we needlessly suffer a long convergence time as most workloads have a lower average packet rate and occasional traffic bursts. When the packet arrival rate is low, we do not need to statically skip many packets (to achieve this rate) and we can thus enlarge p to sample more packets for the sketch, as shown in Figure 6.

We propose two adaptive approaches:

(1) AlwaysLineRate. We dynamically set the counter array sampling probability p to be inversely proportional to the current packet arrival rate. We use a large p when the packet rate is low and reduce p as it increases. This mode performs on average the same number of operations within a time unit regardless of the packet rate.

(2) AlwaysCorrect. In this mode, we start with $p = 1.0$ (same as the original sketches), and switch to AlwaysLineRate once we can guarantee convergence. Doing so allows us to maintain the accuracy guarantees throughout the measurement. This is required to maintain the fidelity of composite sketches such as UnivMon.



(a) Original sketch. (b) Sketch with NitroSketch

Figure 7: (a) Before using NitroSketch, each packet goes through multiple hash computations (e.g., $O(\log \delta^{-1})$), update multiple counters, and query and update to a top-k flow key storage (e.g., heap). (b) After applying NitroSketch, only a small portion of packets (say 5% by geometric sampling) need to go through one hash computation, update to one row of counters (instead of all rows) and occasionally to a top-k structure. Therefore, the CPU cost is significantly reduced.

In practice, the choice of mode depends on the use cases. The AlwaysLineRate mode usually reduces the convergence time to <1s in a 10Gbps+ link but there is still a delay on the result. For typical measurement statistics and their higher-layer management applications, such as heavy hitters for load balancing [48, 54], flow size distribution for cache admission/eviction [42], and entropy for anomaly detection [52, 66], we recommend using this mode as this delay is tolerable. The AlwaysCorrect mode avoids the convergence time but there is a delay on the performance speedup. For fine-grained network management and security applications such as detection of dynamic DDoS attacks [33] that need visibility at finer granularities (e.g., 100ms to 10s), the AlwaysCorrect mode is more suitable.

Idea D: Buffer counter updates and use SIMD parallelism. Since sketch operations are independent, after applying the ideas mentioned above, we can buffer the sampled counters and their pending flow keys to compute the hash functions and counter updates in Single Instruction Multiple Data (SIMD) mode. For best compatibility with morden CPUs, we use AVX [45] (Intel and AMD CPUs support AVX since 2011) in our implementation. We also include non-SIMD version in the implementation to avoid potential compatibility issues with other software platforms.

4.3 End-to-End view of NitroSketch

We now describe the end-to-end view of our approach. In NitroSketch, we have a data plane component to process packets with sketching and a control plane component to estimate the statistics from the sketch data structure. We split the data plane packet processing into a *pre-processing* stage and a *sketch-updating* stage. NitroSketch is illustrated in Figure 7(b), and its pseudocode is available in Algorithm 1. We now describe the two stages.

Pre-processing stage. Once packet batches arrive (e.g., from DPDK Polling Mode Driver), NitroSketch only "selects" the packets that need to update a counter array. As in ideas A and B, we use a single geometric variable to sample the packets and counter arrays which implies that the majority of packets (say, 95%) are "skipped".

Sketch-updating stage. NitroSketch only delivers the sampled packets to this stage which minimizes the overheads. Sampled packets from the pre-processing stage update one (or more) counter/s in one (or more) array/s based on the updating functions of different sketches. For instance, in NitroSketch with Count Sketch, each sampled packet will update one or more counters with $\{+p^{-1}, -p^{-1}\}$ where p is the geo-sampling probability.

Algorithm 1 NitroSketch Data-plane

Input: Packet stream $D(m,n) = a_0, \ldots, a_{m-1} \in [n]^{[m]}$

1: **procedure Initialization**
2: Generate pairwise independent hash functions:
$\{h_i{:}[n] \to [w]; \quad g_i{:}[n] \to \{-1,+1\}$ or $\{+1\}\}_{i\in[d]}$
3: ▷ g_i is either ±1 getting an L_2 guarantee or +1 for an L_1 guarantee.
4: $r \leftarrow (-1)$, $j \leftarrow 0$ ▷ The next row (r) and packet (j) to select
5: **procedure AlwaysLineRate**
6: **while** $j \le m$ **do** ▷ Continue to process packets
7: **Update**(p) ▷ Update each row w.p. p
8: **if** 100ms passed **then** ▷ Adjust p's value
9: Set p inversely proportional to traffic rate
10: **procedure AlwaysCorrect**
11: $T \leftarrow 121(1+\epsilon\sqrt{p})\epsilon^{-4}p^{-2}$ ▷ Convergence threshold
12: **while** $j \le m$ **do** ▷ Continue to process packets
13: **Update**(1) ▷ Before convergence, we always update
14: **if** $(j \mod Q) = 0 \wedge (\text{median}_{i\in[d]} \sum_{y=1}^{w} C_{i,y}^2) > T$ **then**
15: Switch to using **AlwaysLineRate**
16: **procedure Update**(p)
17: r += $Geo(p)$ ▷ Geometric variable
18: j += $\lfloor r/d \rfloor$ ▷ Skip packets if needed
19: $r \leftarrow r \mod d$ ▷ The row to update
20: $C_{r,h_r(a_j)}$ += $p^{-1} \cdot g_r(a_j)$
21: **function Query**(x)
22: **return** $\text{median}_{i\in[d]}\{C_{i,h_i(x)} \cdot g_i(x)\}$

With the two stages above, we design two adaptive modes in the following for various user requirements. The AlwaysLineRate mode offers quick convergence time and a fixed amount of bottleneck operations per packet batch. Alternatively, the AlwaysCorrect mode does not require convergence time but performs the same amount of packet operations as vanilla sketches in the beginning of the measurement.

AlwaysLineRate mode. In this mode, NitroSketch adapts to the packet arrival rate. We increase the sampling probability p when the arrival rate is low, and we lower p when the arrival rate is high to avoid packet drops.

This mode considers choosing the sampling probability $p \in \{1, 2^{-1}, 2^{-2}, \ldots, 2^{-7}\}$ and updating the counters by p^{-1} in the sketch. In Algorithm 1, we monitor the number of processed packets within fixed time epochs (e.g., 100ms) by measuring the received packet timestamps (Line 8). In Line 8, we adjust the sampling probability

p to keep the number of operations (roughly) fixed for each epoch. We do so by setting p to be inversely proportional to the packet rate. Finally, this mode is allocated with the space required for sampling with probability $p_{\min} = 2^{-7}$. This choice assures the theoretical guarantees of Theorem 2 hold and provides better accuracy in practice compared to sampling at a fixed rate of $p_{\min}$.

SketchVisor and ElsaticSketch [43, 73] also have mechanisms that adapt the running speed to the line rate. However, their approach uses a binary fast/slow path approach. In addition, SketchVisor requires a merge operation by the control plane (e.g., SDN controller) which is computationally intensive, and ElasticSketch loses its accuracy guarantee once the light part is used. In comparison, our approach offers a spectrum of eight different processing speeds and performs a roughly fixed amount of work in each packet batch. It retains the accuracy guarantees without controller merging.

AlwaysCorrect mode. This mode provides accuracy guarantees starting with the first packet, but only provides a speedup once enough packets are processed (converged). Taking a Count Sketch as an example, NitroSketch periodically estimates the L_2 norm to determine when it can justify sampling. That way, sampling starts once the measurement is large enough to converge. The pseudocode of this mode is given in Algorithm 1. Observe that we now update rows with different probabilities (initially as 1 and then the varying p), and we update the counters with the inverse sampling probabilities (initially 1 and then p^{-1}).

Formally, the sum of squared counters in each row i is a $(1+\epsilon\sqrt{p})$-multiplicative estimator for the stream's L_2^2 with probability $3/4$, and the rows' median with a probability of $1-\delta$. We perform this computation once per Q (e.g., $Q = 1000$) packets which reduces the overheads and ensures that sampling starts at most Q packets late. We use the union bound to get an overall error probability of 2δ – with probability $\leq \delta$ we start sampling too early and with probability $\leq \delta$ the sketch's error exceeds ϵL_2.

Scope of NitroSketch. Similar as existing sketch solutions [43, 55, 73], NitroSketch relies on the control plane to query the measurement results. The data plane component of NitroSketch is responsible for updating the sketch data structure but not obtaining the traffic statistics from the sketch. Thus, any applications (e.g., connectivity checking [53] and fast rerouting [40]) that require computation entirely in the data plane are not targeted use cases of NitroSketch. For instance, fast data plane processing is needed to measure and react to short-lived traffic surges (microbursts). In handling such microbursts, a programmable hardware switch can be a vantage point, as suggested in [18, 19].

5 ANALYSIS

We now show the theoretical guarantees of NitroSketch. To analyze the worst-case guarantee, we assume that all packets are sampled with probability $p_{\min}$ and denote $p \equiv p_{\min}$. We consider two variants; first, combining the Count-Min Sketch with Nitro for achieving an ϵL_1 guarantee, and second using NitroSketch for an ϵL_2 approximation. Here, $L_k \triangleq \sqrt[k]{F_k} = \sqrt[k]{\sum_{x\in\mathcal{U}} f_x^k}$ is the k-th norm of the frequency vector f (i.e., f_x is the size of flow x) and $\mathcal{U}$ is the set of all possible flows (e.g., all 2^{32} possible source IPs). Specifically, L_1 is simply the number of packets in the measurement.

Supported sketches. Intuitively, NitroSketch improves the processing performance of sketches that normally calculate multiple hash values, and update multiple counter arrays. This criterion includes but not limited to Count-Min Sketch [27], Count Sketch [17], UnivMon [55], and K-ary [51]. Moreover, NitroSketch can further accelerate the slower light part (Count-Min Sketch) of ElasticSketch [73]. Formally, we show that NitroSketch provides ϵL_1 and ϵL_2 accuracy guarantees which is compatible with most sketches.

ϵL_1 and ϵL_2 Guarantees. The ϵL_1 guarantee follows from the analysis in [8]. We show the ϵL_2 guarantee in the following theorems.

THEOREM 1. *Let $d \triangleq \log_2 \delta^{-1}$ and $w \triangleq 4\epsilon^{-1}$. For streams in which $L_1 \geq c \cdot \left(\epsilon^{-2}p^{-1}\sqrt{\log \delta^{-1}}\right)$ for a sufficiently large constant c, NitroSketch + Count-Min Sketch satisfies:* $\Pr\left[|\widehat{f_x} - f_x| \geq \epsilon L_1\right] \leq \delta$ *where f_x is the real frequency of flow x, and $\widehat{f_x}$ is the return value of Query(x) in Algorithm 1.*

Next, we state Theorem 2 and Theorem 5 that establish the correctness of both modes of NitroSketch.

THEOREM 2. *Let $w = 8\epsilon^{-2}p^{-1}, d = O(\log \delta^{-1})$. AlwaysLineRate NitroSketch requires $O(\epsilon^{-2}p^{-1}\log\delta^{-1})$ space, operates in amortized $O(1+dp)$ time (constant for $p = O(1/d)$), and provides the following guarantee:* $\Pr\left[\left|f_x - \widehat{f_x}\right| > \epsilon L_2\right] \leq \delta$ *for streams in which $L_2 \geq 8\epsilon^{-2}p^{-1}$.*

PROOF. We consider the sequence of packets that was sampled for each of the rows. That is, let $S_i \subseteq S$ be the subset of packets that updated row i (for $i \in \{1,\dots d\}$). Further, we denote by $f_{i,x} \triangleq |\{j \mid (x_j \in S_i) \wedge (x_j = x)\}|$ the frequency of x within S_i. That is, $f_{i,x}$ the number of times a packet arrived from flow x and we updated row i. Let $L_2 \triangleq \sqrt{\sum_{x\in\mathcal{U}} f_x^2}$ denote the second norm of the frequency vector of S and similarly $L_{2,i} \triangleq \sqrt{\sum_{x\in\mathcal{U}} f_{i,x}^2}$ denote that of S_i. Clearly, we have $L_{2,i} \leq L_2$ for any row $i \in \{1,\dots,d\}$. Recall that we assume that the sampling probability is fixed at $p = p_{\min}$; if the actual probability for some packets is higher it only decreases the counter variances and therefore the error.

We proceed with a simple lemma that bounds $\mathbb{E}\left[L_{2,i}^2\right]$ as a function of L_2^2. Observe that $f_{i,x} \sim \text{Bin}(f_x, p)$ and thus $\text{Var}[f_{i,x}] = f_x p(1-p)$ and $\mathbb{E}[f_{i,x}] = f_x p$.

LEMMA 3. $\mathbb{E}\left[L_{2,i}^2\right] \leq 2pL_2^2$.

PROOF.

$$\mathbb{E}\left[L_{2,i}^2\right] = \sum_{x\in\mathcal{U}} \mathbb{E}\left[f_{i,x}^2\right] = \sum_{x\in\mathcal{U}} \text{Var}[f_{i,x}] + (\mathbb{E}[f_{i,x}])^2$$
$$= \sum_{x\in\mathcal{U}} f_x p(1-p) + (f_x p)^2 \leq \sum_{x\in\mathcal{U}} 2pf_x^2 = 2pL_2^2. \quad \square$$

Next, we bound the variance of $\left(C_{i,h_i(x)}g_i(x) - p^{-1}f_{i,x}\right)$ – the noise that other flows add to x's counter on the i'th row.

LEMMA 4. $\text{Var}\left[C_{i,h_i(x)}g_i(x) - p^{-1}f_{i,x}\right] \leq 2p^{-1}L_2^2/w$.

PROOF. We have $C_{i,h_i(x)} = p^{-1}\sum_{x'\in\mathcal{U}|h_i(x)=h_i(x')} f_{i,x'}g_i(x')$.

That is, the value of x's counter, $C_{i,h_i(x)}$, is affected by all $x' \in \mathcal{U}$ such that $h_i(x) = h_i(x')$, and the contribution of each such x' is $p^{-1} f_{i,x'} g_i(x')$.

Next, notice that since $\mathbb{E}[g_i(x')] = 0$ and as g_i is two-way independent, we have that $\mathbb{E}\left[C_{i,h_i(x)} \cdot g_i(x)\right] =$
$p^{-1} \sum_{x' \in \mathcal{U}: h_i(x') = h_i(x)} \mathbb{E}\left[f_{i,x'} \cdot g_i(x') \cdot g_i(x)\right] = p^{-1}\mathbb{E}[f_{i,x}] = f_x$.

Now, as h_i is pairwise independent, we have that for any $x' \in \mathcal{U} \setminus \{x\}$: $\Pr[h_i(x) = h_i(x')] = 1/w$. We are now ready to prove the lemma:

$$\begin{aligned}
\mathrm{Var}\left[C_{i,h_i(x)} g_i(x) - p^{-1} f_{i,x}\right] &= \mathbb{E}\left[(C_{i,h_i(x)} g_i(x) - p^{-1} f_{i,x})^2\right] \\
&= \mathbb{E}\left[(C_{i,h_i(x)} g_i(x))^2 - 2p^{-1} C_{i,h_i(x)} g_i(x) f_{i,x} + p^{-2} f_{i,x}^2\right] \\
&= \mathbb{E}\left[\left(p^{-1} \sum_{x' \in \mathcal{U} | h_i(x) = h_i(x')} f_{i,x'} g_i(x') g_i(x)\right)^2\right. \\
&\quad \left. - 2p^{-1}\left(p^{-1} \sum_{x' \in \mathcal{U} | h_i(x) = h_i(x')} f_{i,x'} g_i(x') g_i(x) f_{i,x}\right) + p^{-2} f_{i,x}^2\right] \\
&= p^{-2}\mathbb{E}\left[\left(\sum_{x' \in \mathcal{U} | h_i(x) = h_i(x')} f_{i,x'}^2\right) - f_{i,x}^2\right] \\
&= p^{-2}\mathbb{E}\left[\sum_{x' \in \mathcal{U} \setminus \{x\} | h_i(x) = h_i(x')} f_{i,x'}^2\right] \le p^{-2}\mathbb{E}[L_{2,i}^2]/w \le 2p^{-1}L_2^2/w,
\end{aligned}$$

where the last inequality is correct per Lemma 3. □

We denote $A \equiv C_{i,h_i(x)} g_i(x)$ and $B \equiv p^{-1} f_{i,x}$ (note that since $\mathrm{Var}[f_{i,x}] = f_x p(1-p)$, $\mathrm{Var}[B] = f_x p^{-1}(1-p)$). Our goal is to bound the variance of A and use Chebyshev's inequality.

$$\begin{aligned}
A - B &= \left(p^{-1} \sum_{x' \in \mathcal{U} | h_i(x) = h_i(x')} f_{i,x'} g_i(x') g_i(x)\right) - p^{-1} f_{i,x} \\
&= p^{-1} \sum_{x' \in \mathcal{U} \setminus \{x\} | h_i(x) = h_i(x')} f_{i,x'} g_i(x') g_i(x).
\end{aligned}$$

Notice that $A - B$ (the change caused by all flows but x) is independent from B (just flow x), and thus:

$$\begin{aligned}
\mathrm{VAR}[A] &= \mathrm{VAR}[(A-B) + B] = \mathrm{VAR}[A-B] + \mathrm{VAR}[B] \\
&\le p^{-2} E[L_{2,i}^2]/w + f_x p^{-1}(1-p) \le 2p^{-1}L_2^2/w + f_x p^{-1}.
\end{aligned}$$

We denote by $\widehat{f_x(i)} \triangleq A = C_{i,h_i(x)} g_i(x)$ the estimation for x's frequency provided by the i'th row. Then according to Chebyshev's inequality (where $\sigma(A) = \sqrt{\mathrm{Var}[A]}$):

$$\begin{aligned}
\Pr\left[\left|\widehat{f_x(i)} - f_x\right| \ge \epsilon L_2\right] &= \Pr\left[\left|C_{i,h_i(x)} g_i(x) - \mathbb{E}\left[C_{i,h_i(x)} g_i(x)\right]\right| \ge \epsilon L_2\right] \\
&= \Pr[|A - \mathbb{E}[A]| \ge \epsilon L_2] \le \Pr\left[|A - \mathbb{E}[A]| \ge \frac{\sigma(A) \cdot \epsilon L_2}{\sqrt{2p^{-1}L_2^2/w + f_x p^{-1}}}\right] \\
&\le \frac{2p^{-1}L_2^2/w + f_x p^{-1}}{(\epsilon L_2)^2} = \frac{2L_2^2/w + f_x}{p(\epsilon L_2)^2} = \frac{2/w}{p\epsilon^2} + \frac{f_x}{p(\epsilon L_2)^2} \le \frac{2/w}{p\epsilon^2} + \frac{1}{p\epsilon^2 L_2}.
\end{aligned}$$

We want a constant probability of the error exceeding ϵL_2 in each row, so that the median of the rows will be correct with probability $1-\delta$. Therefore, by demanding $L_2 \ge 8p^{-1}\epsilon^{-2}$ and $w \ge 8p^{-1}\epsilon^{-2}$ we get that the error probability is

$$\Pr\left[\left|\widehat{f_{x,i}} - f_x\right| \ge \epsilon L_2\right] \le \frac{2/w}{p\epsilon^2} + \frac{1}{p\epsilon^2 L_2} \le 3/8.$$

As the $d = O(\log \delta^{-1})$ rows are independent, the algorithm's estimate, $\widehat{f_x} = \mathrm{median}_{i \in \{1,\ldots d\}} \widehat{f_x(i)}$, is correct with a probability of at least $1-\delta$ using a standard Chernoff bound. Specifically, we showed the correctness of Theorem 2. □

The formal proof of Theorem 5 is deferred to Appendix A.

THEOREM 5. *Let $w = 11\epsilon^{-2}p^{-1}$ and $d = O(\log \delta^{-1})$; AlwaysCorrect NitroSketch guarantees:*

$$\Pr\left[|\widehat{f_x} - f_x| > \epsilon L_2\right] < 2\delta.$$

Interpretation of main theorems. Intuitively, the main theorems prove that NitroSketch trades space for throughput while retaining the same asymptotic error bounds as original sketches. Specifically, it requires an $O(p^{-1})$ factor more space than Count Sketch for the same accuracy. Compared to Strawman Approach #1 (One-Array-Count-Sketch), it provides faster updates and has lower space requirements. The improvement in update time comes from reducing the number of hash computations. While One-Array-Count-Sketch computes two hashes per packet, NitroSketch only does so for each *sampled* counter array. As the expected number of sampled counter arrays per packet is $dp = o(1)$, NitroSketch significantly reduces the processing overheads. Space-wise, NitroSketch only requires $O(\epsilon^{-2}\log\delta^{-1}p^{-1})$ space compared to the $O(\epsilon^{-2}\delta^{-1})$ memory of One-Array-Count-Sketch and is therefore more cache resident. For example, we can set $p = d^{-2} = O(\log\delta^{-2})$ to get a space of $O(\epsilon^{-2}\log\delta^{-3})$. In summary, One-Array-Count-Sketch makes two hash computations per packet while NitroSketch makes just $o(1)$ computations. We also formally compare NitroSketch with Strawman Approach #2 in Appendix B.

Convergence time in practice. We use real Internet traces to estimate the convergence time. For example, the first 10M source IPs of the CAIDA 2016 [14] trace has a second norm of $L_2 \approx 1.28 \cdot 10^6$ while 100M packets gives $L_2 \approx 1.03 \cdot 10^7$. This means that if we fix $p_{\min} = 2^{-7}$, we get guaranteed convergence for $\epsilon \ge 2.9\%$ after 10M packets and $\epsilon \ge 1\%$ after 100M. In practice, we observe that the actual error is much lower which suggests that our analysis is just an upper bound and one can use smaller ϵ values as well. Finally, we note that extending the counter array sizes further allows faster convergence of the algorithm.

6 NITROSKETCH IMPLEMENTATION

We have implemented NitroSketch in C and integrated it with Open vSwitch (OVS-DPDK), FD.io/Vector Packet Processing (VPP), and BESS. In each platform, we perform the measurements using a single thread measurement daemon. We integrate four sketches with NitroSketch: UnivMon [55], Count-Min Sketch [27], Count Sketch [17], and K-ary Sketch [51]. In each sketch implementation, we use the xxHash library's [23] hash function, and Intel's AVX [45] to parallelize the computation. The source code is available at [32].

At a high level, NitroSketch includes two modules: a data-plane **Sketching** module and a control-plane **Estimation** module.The Sketching module maintains the sketch data structure, and the Estimation module fetches the data from the Sketching module. We now describe the Sketching module.

OVS-DPDK Integration. OVS-DPDK enables the packet processing entirely in user space, the user space `vswitchd` thread has a three-tier look-up cache hierarchy. The first-level table works as an

Exact Match Cache (EMC) and has the fastest look-up speed. If a packet misses in EMC, it goes through the second-level classifier as a Tuple Space Search and may trigger the third-level table managed by an OpenFlow-compliant controller. For efficiency, we integrate the sketching module with the OVS-DPDK's EMC module in dpif-netdev. We provide implementations for varying performance requirements:

(1) All-in-one version (AIO). In this version, the Sketching module works as a sub-module of the EMC module inside an OVS vswitchd-PMD thread. For each packet batch received from DPDK PMD, NitroSketch decides which packet header is measured without affecting the packet batch, as described in Section 4.2. This extension incurs a small processing overhead to the EMC module but dedicates the CPU to all the tasks inside OVS-DPDK, such as DPDK, table look-up, and measurement. The Sketching module in this version is processed entirely in the OVS data plane (`vswitchd`).

(2) Separate-thread version. In this version the Sketching module works as a separate thread alongside the OVS vswitchd thread. For each batch of packets, the extended logic in EMC module (preprocessing stage) in vswitchd decides which packets' headers to add into a shared buffer (modified from [16]). A separate NitroSketch thread (sketch-updating stage) concurrently empties the buffer and updates the sketch.

VPP and BESS Integration. VPP is a modular, flexible, and extensible platform that runs entirely on the user-space. VPP is based on a "packet processing graph", where each node is a module and packets are processed node by node. For instance, in a simple VPP based L3 vSwitch, VPP first fetches packets from the network I/O as a batch. VPP then sends the packet batch to the Ethernet-input module (L2), and then through IP4-input and IP4-lookup modules (L3). We implemented a measurement module in VPP 18.02 and added it to the packet processing graph after the VPP IP stack. This module runs both stages of NitroSketch in a dedicated thread, minimizing the impact on other VPP plugins. Similarly, BESS is a light-weight modular software switch, and we implement the sketching module of NitroSketch as a plugin in the data plane processing pipeline.

Control Plane Module The control plane module uses the monitoring system to (i) periodically (at the end of each epoch) receive sketching data from the data plane module through a 1GbE link connected to the virtual switch; (ii) assign the sketching data to the corresponding measurement tasks based on user definitions; (iii) calculate the estimated results. For example, it can use UnivMon [55] to calculate HH, Change detection, or traffic Entropy.

7 EVALUATION

Our evaluation demonstrates that NitroSketch: (a) can match 40Gbps with a single core for measurement; (b) runs on software switches with small CPU overheads; (c) provides accurate results once converged; (d) has higher throughput ($> 7.6\times$ faster) and better accuracy once converged when compared to SketchVisor [43], and (e) is more accurate and requires less memory than NetFlow [21] and sFlow [71].

Testbed. We evaluate NitroSketch on a set of 4 commodity servers running Ubuntu 16.04.03, each of which has an Intel Xeon E5-2620 v4 CPU@3.0Ghz, 128GB DDR4 2400Mhz memory, two Broadcom BCM5720 1GbE NICs, and an Intel XL710 Ethernet NIC with two 40-Gigabit ports. Our testbed has three hosts as the data plane directly connected through 40Gbps links. The control is connected through a 1Gbps link. Each virtual switch is configured with two forwarding rules for bidirectional packet forwarding.

Workloads. We use four types of workloads: (a) *CAIDA*: 10 one-hour public CAIDA traces from 2016 [14] and 2018 [15] each containing 1 to 1.9 billion packets; (b) *Min-sized*: simulated traffic with min-sized packets for stress testing; (c) *Data center*: data center traces UNI1 and UNI2 from [11]; (d) *Cyber attack*: DDoS attack traces from [58]. The average packet sizes in the CAIDA, Cyber attack, and data center traces are 714, 272, and 747 bytes respectively. To minimize confounding effects of overheads, we modify the MAC addresses of packets to avoid cache misses on the Exact-Match Cache of OVS-DPDK. We use MoonGen [31] to replay traces and to generate random 64B packets.

Sketches and metrics. We evaluate NitroSketch with four popular sketches Count-Min Sketch [27], Count-Sketch [17], UnivMon [55], and K-ary Sketch [51]. We use 5-tuple as the flow key and consider the following performance metrics:

- *Throughput:* in gigabits per second (Gbps) of traffic.
- *Packet Rate:* Millions of transmitted packets per second (Mpps). For 64B packets, 10Gbps throughput is equivalent to 14.88Mpps, and 40Gbps equals to 59.53Mpps.
- *CPU Utilization:* percentage of the CPU time spent on each module/function, measured by Intel VTune Amplifier [46].
- *Accuracy:* the accuracy of three measurement tasks: Heavy Hitter (HH), Change Detection (Change), and Entropy Estimation (Entropy). For HH and Change, we set a threshold 0.05% and estimate the mean relative errors on the detected heavy flows. We report *relative error*= $\frac{|t-t_{real}|}{t_{real}}$, where t_{real} is the ground truth of a task and t is the measured value. For each data point, we run 10 times independently and report the *median* and the *standard deviation*. Also, the *recall rate* is defined as the ratio of true instances found.

Parameters. By default, we select parameters based on a 5% accuracy guarantee. Note that this is a theoretical guarantee for parameter selection purposes and NitroSketch achieves higher fidelity in practice (e.g., $< 1\%$ errors). For throughput evaluation, we set a fixed $p = 0.01$ geometric sampling rate for NitroSketch and allocate the memory based on the precision guarantee. We evaluate four sketches in NitroSketch. (a) UnivMon: we allocate 4MB, 2MB, 1MB, 500KB for the first HH sketches, and 250KB for the rest of sketches. (b) Count-Min: we use 200KB memory for 5 rows of 10000 counters. (c) Count Sketch: we allocate 2MB for 5 rows of 102400 counters. (d) K-ary Sketch: we utilize 2MB for 10 rows of 51200 counters.

7.1 Throughput

Throughput with AIO version. We evaluate the throughput of the all-in-one version in Figure 8(a) with 1h CAIDA traces and 1h datacenter traces (looped). All original sketches implemented with OVS-DPDK suffer from significant throughput degradation. Among the four sketches, UnivMon achieves 2.1Gbps and the faster Count-Min only reaches 5.5Gbps. With NitroSketch, all sketches achieve 10G and 40G line rates under CAIDA and datacenter traces, without adding an extra thread. We observe that inside this vswitchd thread,

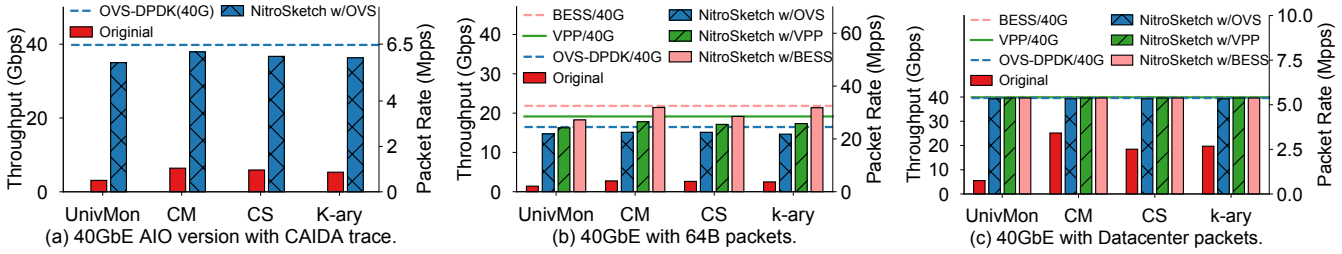


Figure 8: Throughput (left y-axis) and packet rates (right y-axis) on OVS-DPDK, VPP, and BESS. In (a), OVS-DPDK uses single-thread inline version while in (b) and (c) use a single-thread NitroSketch and another two threads for the switches.

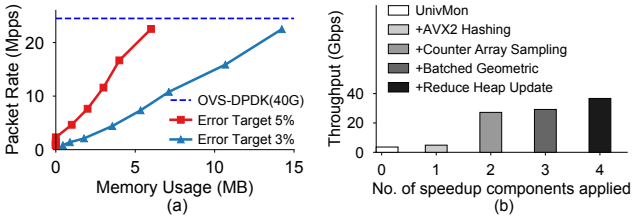


Figure 9: (a) Throughput vs. memory for varying error targets. (b) Throughput with different NitroSketch components applied (Setting: one vswitchd thread with 40GbE NIC).

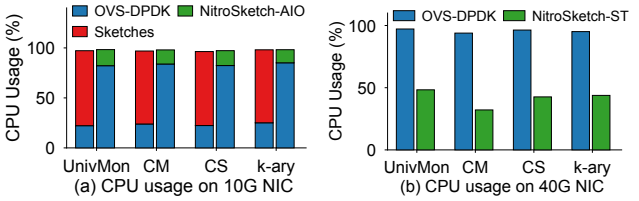


Figure 10: CPU usage of the all-in-one version (NitroSketch-AIO) and the separate-thread version (NitroSketch-ST).

DPDK, OVS, and NitroSketch modules use all the potential of a single core.

Throughput with separate-thread. Figures 8(b,c) show the throughput of the separate-thread version. It is already difficult for virtual switches to achieve 10G line-rate on a single core with 64B packets. For 40G, even vanilla DPDK does not reach the line rate with 64B packets due to the hardware limitation in Intel XL710 NIC [29]. This means that OVS-DPDK, VPP, and BESS cannot reach this line rate under 64B packet traces. In Figure 8(b), we see that NitroSketch has a negligible throughput impact on the virtual switches. That is, it achieves 20Gbps+ line rate under any workload. As is evident from Figure 8(b) and (c), NitroSketch is not the bottleneck in achieving 40G line rates for 64B packets and for data center workloads.

Throughput vs. Memory. To guarantee an error budget ϵ (for any distribution), the sampling probability p in the pre-processing stage depends on the amount of allocated memory. To illustrate this trade-off, we set error guarantees 3% and 5% for UnivMon with NitroSketch. Figure 9(a) shows that NitroSketch copes with 40G OVS-DPDK with an acceptable increase in memory.

Improvement breakdown. While implementing NitroSketch, we used multiple optimization techniques. Therefore, we evaluate the gains of each optimization separately for UnivMon with NitroSketch. Figure 9(b) confirms that the counter array sampling technique offers the most significant speedup.

Throughput with AlwaysCorrect NitroSketch. To evaluate the convergence time in this mode, we implement AlwaysCorrect NitroSketch with Count-Sketch and UnivMon in OVS-DPDK with the AIO version. In Figure 11(c), we report the measured throughput every 0.1sec (extra measurement overhead added) under 40GbE. We see that it needs about 0.6s for Count-Sketch and 0.8s for UnivMon to reach full speed.

7.2 CPU Utilization

A single DPDK PMD thread is continuously polling packets from NIC. It "saturates" a core and utilizes 100% CPU reported from a universal process viewer (e.g., htop). Therefore, we profile the CPU time of each module.

CPU Time in all-in-one. We measure the CPU time in the same setting as in Figure 8(a). As shown in Figure 10(a), when vanilla sketches are running, most of the CPU time is spent on sketching, and the overall switching performance drops. After applying NitroSketch-AIO, the switch achieves line-rate while keeping the NitroSketch's CPU < 20%.

CPU Time in separate-thread. Figure 10(b) compares the CPU time between OVS-DPDK and NitroSketch-Separate Thread, in a setting as in Figure 8(b). When the switch is saturated with min-sized packets (~22Mpps), the cores for packet switching are running at nearly 100% while NitroSketch is not running at full-speed and would handle higher packet rates, if the virtual switch supports.

7.3 Accuracy and Convergence Time

We evaluate the accuracy of UnivMon, CMS, Count Sketch and K-ary in NitroSketch with different sized epochs and report in Figures 11 and 12. Our experiments show that NitroSketch converges to similar accurate results as vanilla sketches after receiving enough packets. As depicted in Figure 11(a) and (b), with fixed 0.1 and 0.01 sampling rates, NitroSketch with UnivMon can achieve a similar accuracy as the vanilla UnivMon after receiving 8M packets. As shown in Figure 12(a) and (b), tested sketches with NitroSketch can achieve better-than-guaranteed results (< 5% error) after seeing 2-4M packets and converge to similar accuracy after receiving 8-16M packets. An interesting finding here is that Count-Min Sketch achieves better-than-original results when NitroSketch is enabled and converged. We believe this is because Count-Min Sketch overestimates the counts (i.e., always +1) and produced "biased" estimates and NitroSketch's

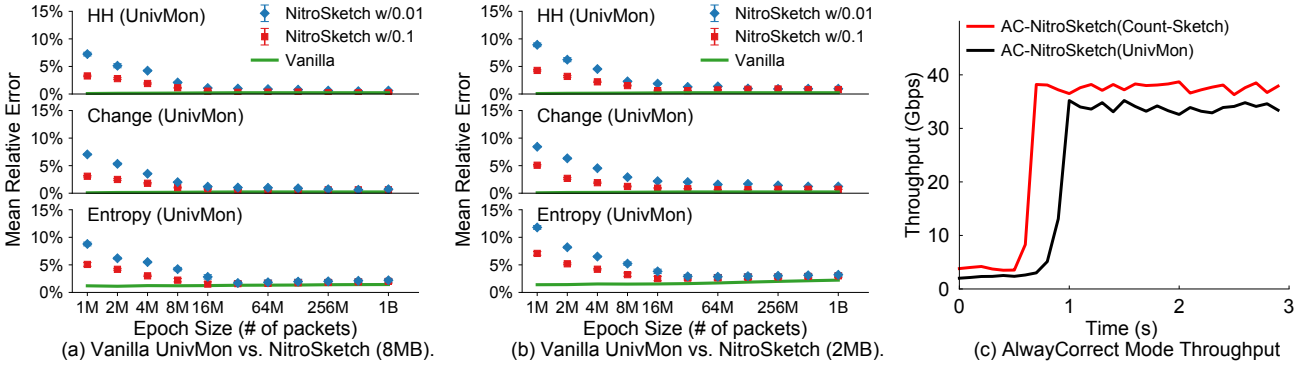


Figure 11: (a),(b) Error rates of vanilla UnivMon and NitroSketch with different fixed sampling rates p (0.1 and 0.01) and memory settings (8MB and 2MB). (c) Throughput over time for AlwaysCorrect NitroSketch with different sketches.

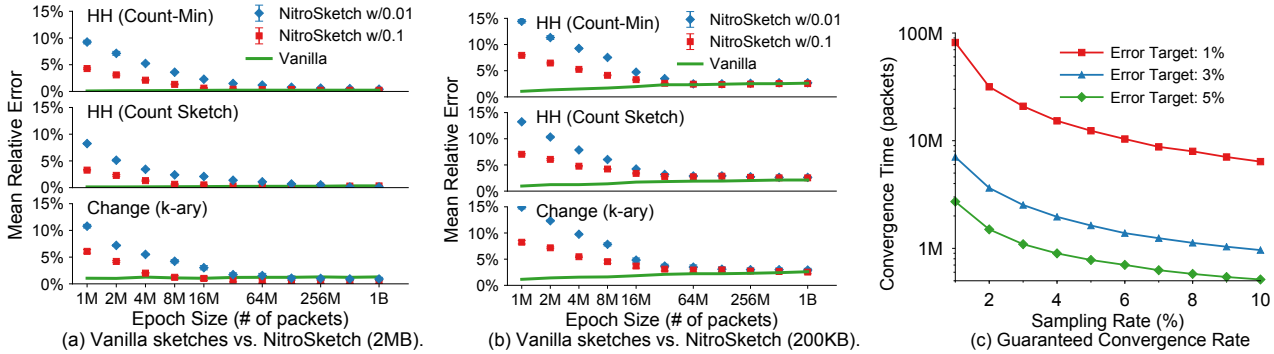


Figure 12: (a),(b) Error rates of vanilla sketches and NitroSketch with different fixed sampling rates p (0.1 and 0.01) and memory settings (2MB and 200KB). (c) Proven convergence time on CAIDA traces.

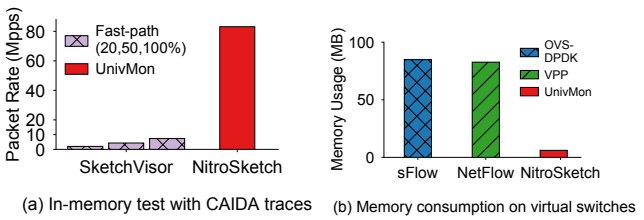


Figure 13: (a) Throughput: SketchVisor vs. NitroSketch. (b) Memory usage: NetFlow vs. NitroSketch.

sampling procedure actually corrects such an overestimation. Finally, it is worth noting that the error results are collected based on fixed-rate NitroSketch. When adopting AlwaysLineRate Mode with adaptive sampling rates on real-world traffic, NitroSketch will achieve better accuracy and faster convergence.

Since NitroSketch uses sampling to select packets, it requires a convergence time to produce a guaranteed accurate result (analyzed in section 5). For different error targets on CAIDA traces, we study the trade-off between geo-sampling rate p and the convergence time (in terms of the number of packets) and report in Figure 12(c). Further, NitroSketch is expected to converge faster on data center traffic due to their more skewed workload and expected larger L_2 value establishment.

7.4 Comparison with Other Solutions

SketchVisor accelerates sketches using a fast path algorithm in its front-end. Since the source code of SketchVisor [43] on Open vSwitch is not publicly available we implement its fast-path algorithm in C and carefully integrate it with UnivMon on OVS-DPDK using the same FIFO buffer as NitroSketch [16]. SketchVisor's performance depends on the portion of the traffic that is processed by the fast path. Thus we evaluate the throughput based on in-memory testing with manually injecting 20%, 50%, 100% of traffic into the fast path. We allocate memory for SketchVisor and NitroSketch to detect top 100 HHs, we use 900 counters for the fast-path and set a 5% error guarantee on UnivMon.

As reported in Figure 13(a), the throughput of SketchVisor improves when the percentage of traffic handled by the fast-path increases. When the fast-path processes 20% of the traffic, it achieves 2.12Mpps. SketchVisor reaches its maximum packet rate of 6.11Mpps when 100% traffic goes into the fast-path. Meanwhile, NitroSketch runs at a dramatically faster speed of 83Mpps. Unsurprisingly, this explains the situation that SketchVisor uses 100% CPU (not shown in the figure) while NitroSketch requires less than 50% (shown in Figure 10(b)) when running in a separate thread on OVS-DPDK.

We observe that to cope with the full 10G speed and avoid packet drops, the fast-path has to handle 100% of the packets. For a fair comparison on OVS, we prevent packet drop by using a very large

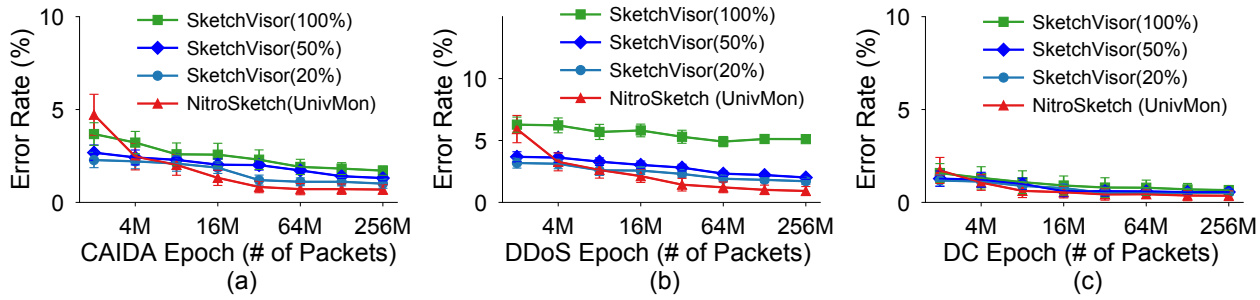


Figure 14: HH errors of SketchVisor and NitroSketch, in CAIDA, DDoS, and datacenter traces.

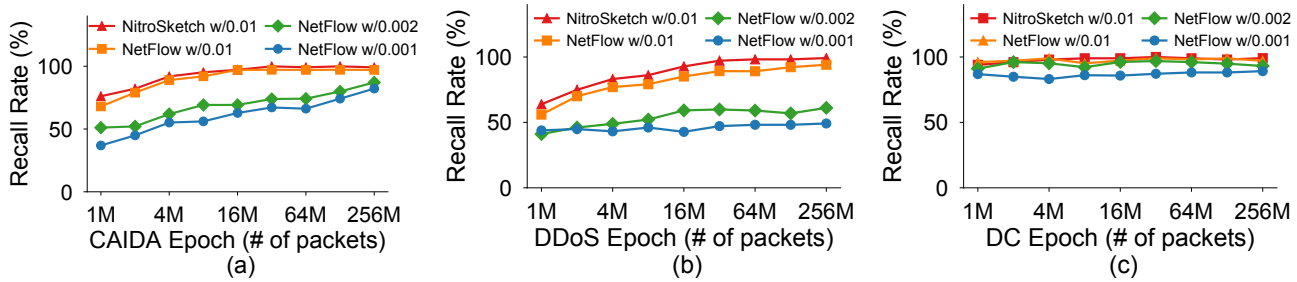


Figure 15: HH recalls of NetFlow with different sampling rates vs. NitroSketch with 0.01, using CAIDA, DDoS, and datacenter traces.

buffer. We manually redirect 20%, 50%, and 100% of the packets to the fast-path. Figure 14(a), (b) and (c) report relative errors on HH in the three traces. We can see that NitroSketch has larger errors before convergence ($<$ 3.61M packets) but is more accurate than SketchVisor after convergence. In a 10G OVS-DPDK switch, this stabilization time can be as little as 0.24 seconds. Here, SketchVisor is inaccurate in the CAIDA and DDoS trace in Figure 14(a) and (b) and is relatively accurate in the data center trace [11]. In contrast, NitroSketch achieves good accuracy on all traces.

Comparison with NetFlow/sFlow. On OVS-DPDK and VPP, NetFlow/sFlow are default monitoring tools. We configure OVS-DPDK to enable sFlow and VPP to enable NetFlow. We set a polling interval of 10 seconds with sampling rates of 0.001, 0.002, and 0.01 for NetFlow. For fairness, we configured NitroSketch with a sampling probability of 0.01. In practice, it is often unreasonable to configure NetFlow with higher sampling rates because a large sampling rate can potentially incur huge memory consumption in high line-rate switches. On the controller, we collect the sampled packets/reports with Wireshark [24] directly from the port. Figure 13(b) indicates that NetFlow consumes much more memory with 0.01 sampling rate. In NetFlow (as in Figure 15), we observe that the recall rates of 100 HHs are low in the CAIDA and DDoS traces and are relatively good in the UNI2 datacenter trace [11]. This is because UNI2 is quite skewed while CAIDA and DDoS are heavy tailed. In contrast, NitroSketch achieves high recall rates in all cases.

8 CONCLUSIONS AND DISCUSSION

Sketching continues to be a promising direction in network measurement. However, its current performance on software switches is far from ideal to serve as a viable line-rate and low CPU consumption option. We identify the key bottlenecks and optimizations for software sketches. Our optimization is encapsulated into NitroSketch, an open source high-performance software sketching framework [32]. NitroSketch supports a variety of measurement tasks and provides accuracy guarantees. Our evaluation shows that NitroSketch achieves the line rate using multiple software switches, and offers competitive accuracy compared to the alternatives.

Interestingly, by replacing each Count Sketch instance in UnivMon with AlwaysCorrect NitroSketch, we get an optimized solution that can provide a $(1+\epsilon)$-approximation for measurement tasks which are known to be *infeasible* to estimate accurately from a uniform sample [60]. Specifically, count distinct cannot be approximated better than a $\Omega(1/\sqrt{p})$ factor while Entropy does not admit any constant factor approximation even if $p = 1/2$!

In Appendices, this paper has supporting material that has not been peer reviewed. Finally, we can state that this work does not raise any ethical issues.

9 ACKNOWLEDGEMENTS

We would like to thank the anonymous SIGCOMM reviewers and our shepherd Alex C. Snoeren for their thorough comments and feedback that helped improve the paper. We thank Omid Alipourfard, Sujata Banerjee, Minlan Yu, and Intel SPAN center for their helpful suggestions. This work was supported in part by CONIX Research Center, one of six centers in JUMP, a Semiconductor Research Corporation program sponsored by DARPA, NSF grants CNS-1565343, CNS-1700521, NSF CAREER-1652257, ONR Award N00014-18-1-2364, Israeli Science Foundation grant 1505/16, the Lifelong Learning Machines program from DARPA/MTO, the Technion HPI research school, the Zuckerman Foundation, the Technion Hiroshi Fujiwara Cyber Security Research Center, the Israel Cyber Directorate, the Cyber Security Research Center and the Lynne and William Frankel Center for Computing Science at Ben-Gurion University.

REFERENCES

[1] Omid Alipourfard, Masoud Moshref, and Minlan Yu. 2015. Re-evaluating Measurement Algorithms in Software. In *Proc. of ACM HotNets*.

[2] Omid Alipourfard, Masoud Moshref, Yang Zhou, Tong Yang, and Minlan Yu. 2018. A Comparison of Performance and Accuracy of Measurement Algorithms in Software. In *Proc. of ACM SOSR*.

[3] Mohammad Alizadeh, Tom Edsall, Sarang Dharmapurikar, Ramanan Vaidyanathan, Kevin Chu, Andy Fingerhut, Vinh The Lam, Francis Matus, Rong Pan, Navindra Yadav, and George Varghese. 2014. CONGA: Distributed Congestion-aware Load Balancing for Datacenters. In *Proc. of ACM SIGCOMM*.

[4] Mohammad Alizadeh, Shuang Yang, Milad Sharif, Sachin Katti, Nick McKeown, Balaji Prabhakar, and Scott Shenker. 2013. pFabric: Minimal Near-optimal Datacenter Transport. In *Proc. of ACM SIGCOMM*.

[5] Noga Alon, Yossi Matias, and Mario Szegedy. 1996. The Space Complexity of Approximating the Frequency Moments. In *Proc. of ACM STOC*.

[6] Eran Assaf, Ran Ben-Basat, Gil Einziger, and Roy Friedman. 2018. Pay for a sliding bloom filter and get counting, distinct elements, and entropy for free. In *Proc. of IEEE INFOCOM*.

[7] Ziv Bar-Yossef, T. S. Jayram, Ravi Kumar, D. Sivakumar, and Luca Trevisan. 2002. Counting Distinct Elements in a Data Stream. In *Proc. of RANDOM*.

[8] Ran Ben Basat, Gil Einziger, Roy Friedman, Marcelo Caggiani Luizelli, and Erez Waisbard. 2017. Constant Time Updates in Hierarchical Heavy Hitters. In *Proc. of ACM SIGCOMM and CoRR/1707.06778*.

[9] Ran Ben Basat, Gil Einziger, Roy Friedman, Marcelo Caggiani Luizelli, and Erez Waisbard. 2018. Volumetric Hierarchical Heavy Hitters. In *Proc. of IEEE MASCOTS*.

[10] Ran Ben-Basat, Xiaoqi Chen, Gil Einziger, and Ori Rottenstreich. 2018. Efficient Measurement on Programmable Switches Using Probabilistic Recirculation. In *Proc. of IEEE ICNP*.

[11] Theophilus Benson, Aditya Akella, and David A. Maltz. 2010. Network Traffic Characteristics of Data Centers in the Wild. In *Proc. of ACM IMC*.

[12] Theophilus Benson, Ashok Anand, Aditya Akella, and Ming Zhang. 2011. MicroTE: Fine Grained Traffic Engineering for Data Centers. In *Proc. of ACM CoNEXT*.

[13] Supratik Bhattacharyya, Andre Madeira, S. Muthukrishnan, and Tao Ye. 2007. How to Scalably and Accurately Skip Past Streams. In *Proc. of IEEE ICDE*.

[14] CAIDA. 2016. The CAIDA UCSD Anonymized Internet Traces equinix-chicago. http://www.caida.org/data/passive/passive_2016_dataset.xml

[15] CAIDA. 2018. The CAIDA UCSD Anonymized Internet Traces equinix-chicago. http://www.caida.org/data/passive/passive_dataset.xml

[16] Cameron. 2015. Fast Concurrent Queue. https://github.com/cameron314/readerwriterqueue

[17] Moses Charikar, Kevin Chen, and Martin Farach-Colton. 2002. Finding Frequent Items in Data Streams. In *Proc. of ICALP*.

[18] Xiaoqi Chen, Shir Landau Feibish, Yaron Koral, Jennifer Rexford, and Ori Rottenstreich. 2018. Catching the Microburst Culprits with Snappy. In *Proc. of SelfDN Workshop*.

[19] Xiaoqi Chen, Shir Landau Feibish, Yaron Koral, Jennifer Rexford, Ori Rottenstreich, Steven A. Monetti, and Wang Tzuu-Yi. 2019. Fine-Grained Queue Measurement in the Data Plane. In *Proc. of ACM CoNEXT*.

[20] Kenjiro Cho. 2017. Recursive Lattice Search: Hierarchical Heavy Hitters Revisited. In *Proc. of ACM IMC*.

[21] Cisco. 2012. Introduction to Cisco IOS NetFlow. https://www.cisco.com/c/en/us/products/collateral/ios-nx-os-software/ios-netflow/prod_white_paper0900aecd80406232.html

[22] Cisco. 2015. Cisco Nexus 1000V Switch. https://www.cisco.com/c/en/us/products/switches/nexus-1000v-switch-vmware-vsphere/index.html

[23] Yann Collet. 2016. xxHash Library. http://www.xxhash.com/

[24] Gerald Combs. 1998. Wireshark. https://www.wireshark.org

[25] Graham Cormode and Minos Garofalakis. 2007. Sketching Probabilistic Data Streams. In *Proc. of ACM SIGMOD*.

[26] Graham Cormode, Flip Korn, S. Muthukrishnan, and Divesh Srivastava. 2008. Finding Hierarchical Heavy Hitters in Streaming Data. *ACM Trans. Knowl. Discov. Data* (2008).

[27] Graham Cormode and S. Muthukrishnan. 2005. An Improved Data Stream Summary: The Count-Min Sketch and Its Applications. *J. Algorithms* (2005).

[28] Andrew R. Curtis, Jeffrey C. Mogul, Jean Tourrilhes, Praveen Yalagandula, Puneet Sharma, and Sujata Banerjee. 2011. DevoFlow: Scaling Flow Management for High-performance Networks. In *Proc. of ACM SIGCOMM*.

[29] Intel Ethernet Networking Division. 2018. Intel Ethernet Controller 710 Series Datasheet. https://www.intel.com/content/dam/www/public/us/en/documents/datasheets/xl710-10-40-controller-datasheet.pdf

[30] Rick Durrett. 2010. *Probability: Theory and Examples* (4th ed.). Cambridge University Press.

[31] Paul Emmerich, Sebastian Gallenmüller, Daniel Raumer, Florian Wohlfart, and Georg Carle. 2015. MoonGen: A Scriptable High-Speed Packet Generator. In *Proc. of ACM IMC*.

[32] Zaoxing Liu et al. 2019. NitroSketch Source Code. https://github.com/zaoxing/NitroSketch

[33] Seyed K. Fayaz, Yoshiaki Tobioka, Vyas Sekar, and Michael Bailey. 2015. Bohatei: Flexible and Elastic DDoS Defense. In *Proc. of USENIX Security*.

[34] FD.io. 2018. Vector Packet Processing. https://fd.io/technology/

[35] William Feller. 1943. Generalization of a Probability Limit Theorem of Cramér. *Trans. Amer. Math. Soc.* (1943).

[36] Pedro Garcia-Teodoro, Jesus E. Diaz-Verdejo, Gabriel Macia-Fernandez, and E. Vazquez. 2009. Anomaly-Based Network Intrusion Detection: Techniques, Systems and Challenges. *Computers and Security* (2009).

[37] Robert D Gordon. 1941. Values of Mills' Ratio of Area to Bounding Ordinate and of the Normal Probability Integral for Large Values of the Argument. *The Annals of Mathematical Statistics* (1941).

[38] Arpit Gupta, Rob Harrison, Marco Canini, Nick Feamster, Jennifer Rexford, and Walter Willinger. 2018. Sonata: Query-Driven Streaming Network Telemetry. In *Proc. of ACM SIGCOMM*.

[39] Sangjin Han, Keon Jang, Aurojit Panda, Shoumik Palkar, Dongsu Han, and Sylvia Ratnasamy. 2015. *SoftNIC: A Software NIC to Augment Hardware*. Technical Report.

[40] Thomas Holterbach, Edgar Costa Molero, Maria Apostolaki, Alberto Dainotti, Stefano Vissicchio, and Laurent Vanbever. 2019. Blink: Fast Connectivity Recovery Entirely in the Data Plane. In *Proc. of USENIX NSDI*.

[41] Nan Hua, Bill Lin, Jun (Jim) Xu, and Haiquan (Chuck) Zhao. 2008. BRICK: ANovel Exact Active Statistics Counter Architecture. In *Proc. of ACM/IEEE ANCS*.

[42] Qi Huang, Ken Birman, Robbert van Renesse, Wyatt Lloyd, Sanjeev Kumar, and Harry C. Li. 2013. An Analysis of Facebook Photo Caching. In *Proc. of ACM SOSP*.

[43] Qun Huang, Xin Jin, Patrick P. C. Lee, Runhui Li, Lu Tang, Yi-Chao Chen, and Gong Zhang. 2017. SketchVisor: Robust Network Measurement for Software Packet Processing. In *Proc. of ACM SIGCOMM*.

[44] Qun Huang, Patrick PC Lee, and Yungang Bao. 2018. SketchLearn: Relieving User Burdens in ApproximateMeasurement with Automated Statistical Inference. In *Proc. of ACM SIGCOMM*.

[45] Intel. 2012. Intel Advanced Vector Extensions. https://software.intel.com/en-us/isa-extensions/intel-avx

[46] Intel. 2018. Intel VTune Amplifier. https://software.intel.com/en-us/vtune

[47] T. S. Jayram, Andrew McGregor, S. Muthukrishnan, and Erik Vee. 2007. Estimating Statistical Aggregates on Probabilistic Data Streams. *Proc. of ACM PODS* (2007).

[48] Xin Jin, Xiaozhou Li, Haoyu Zhang, Robert Soulé, Jeongkeun Lee, Nate Foster, Changhoon Kim, and Ion Stoica. 2017. NetCache: Balancing Key-Value Stores with Fast In-Network Caching. In *Proc. of ACM SOSP*.

[49] Abdul Kabbani, Mohammad Alizadeh, Masato Yasuda, Rong Pan, and Balaji Prabhakar. 2010. AF-QCN: Approximate Fairness with Quantized Congestion Notification for Multi-tenanted Data Centers. In *Prof. of IEEE HOTI*.

[50] Maurice George Kendall, Alan Stuart, and Keith Ord. 1987. *Kendall's Advanced Theory of Statistics*. Oxford University Press, Inc.

[51] Balachander Krishnamurthy, Subhabrata Sen, Yin Zhang, and Yan Chen. 2003. Sketch-based Change Detection: Methods, Evaluation, and Applications. In *Proc. of ACM IMC*.

[52] Ashwin Lall, Vyas Sekar, Mitsunori Ogihara, Jun Xu, and Hui Zhang. 2006. Data Streaming Algorithms for Estimating Entropy of Network Traffic. In *Proc. of ACM SIGMETRICS/Performance*.

[53] Junda Liu, Aurojit Panda, Ankit Singla, Brighten Godfrey, Michael Schapira, and Scott Shenker. 2013. Ensuring Connectivity via Data Plane Mechanisms. In *Proc. of USENIX NSDI*.

[54] Zaoxing Liu, Zhihao Bai, Zhenming Liu, Xiaozhou Li, Changhoon Kim, Vladimir Braverman, Xin Jin, and Ion Stoica. 2019. DistCache: Provable Load Balancing for Large-Scale Storage Systems with Distributed Caching. In *Proc. of USENIX FAST*.

[55] Zaoxing Liu, Antonis Manousis, Gregory Vorsanger, Vyas Sekar, and Vladimir Braverman. 2016. One Sketch to Rule Them All: Rethinking Network Flow Monitoring with UnivMon. In *Proc. of ACM SIGCOMM*.

[56] Zaoxing Liu, Greg Vorsanger, Vladimir Braverman, and Vyas Sekar. 2015. Enabling a "RISC" Approach for Software-Defined Monitoring Using Universal Streaming. In *Proc. of ACM HotNets*.

[57] Yi Lu, Andrea Montanari, Balaji Prabhakar, Sarang Dharmapurikar, and Abdul Kabbani. 2008. Counter Braids: A Novel Counter Architecture for Per-Flow Measurement. In *Proc. of ACM SIGMETRICS*.

[58] MACCDC. 2012. Capture Traces from Mid-Atlantic CCDC. http://www.netresec.com/?page=MACCDC

[59] Jiri Matousek and Jan Vondrak. 2008. The Probabilistic Method-Lecture Notes. http://www.cs.cmu.edu/~15850/handouts/matousek-vondrak-prob-ln.pdf

[60] Andrew McGregor, A Pavan, Srikanta Tirthapura, and David P. Woodruff. 2016. Space-Efficient Estimation of Statistics Over Sub-Sampled Streams. *Algorithmica* (2016).

[61] Ahmed Metwally, Divyakant Agrawal, and Amr El Abbadi. 2005. Efficient Computation of Frequent and Top-k Elements in Data Streams. In *Proc. of ICDT*.
[62] Microsoft. 2016. Hyper-V Virtual Switch Overview. https://technet.microsoft.com/en-us/library/hh831823.aspx
[63] Jayadev Misra and David Gries. 1982. *Finding Repeated Elements*. Technical Report.
[64] M. Mitzenmacher, T. Steinke, and J. Thaler. 2012. Hierarchical Heavy Hitters with the Space Saving Algorithm. In *Proc. of ALENEX*.
[65] Srinivas Narayana, Anirudh Sivaraman, Vikram Nathan, Prateesh Goyal, Venkat Arun, Mohammad Alizadeh, Vimalkumar Jeyakumar, and Changhoon Kim. 2017. Language-Directed Hardware Design for Network Performance Monitoring. In *Proc. of ACM SIGCOMM*.
[66] George Nychis, Vyas Sekar, David G. Andersen, Hyong Kim, and Hui Zhang. 2008. An Empirical Evaluation of Entropy-based Traffic Anomaly Detection. In *Proc. of ACM IMC*.
[67] Ben Pfaff, Justin Pettit, Teemu Koponen, Ethan Jackson, Andy Zhou, Jarno Rajahalme, Jesse Gross, Alex Wang, Joe Stringer, Pravin Shelar, Keith Amidon, and Martin Casado. 2015. The Design and Implementation of Open vSwitch. In *Proc. of USENIX NSDI*.
[68] Robert Schweller, Ashish Gupta, Elliot Parsons, and Yan Chen. 2004. Reversible Sketches for Efficient and Accurate Change Detection over Network Data Streams. In *Proc. of ACM IMC*.
[69] Vibhaalakshmi Sivaraman, Srinivas Narayana, Ori Rottenstreich, S. Muthukrishnan, and Jennifer Rexford. 2017. Heavy-Hitter Detection Entirely in the Data Plane. In *Proc. of ACM SOSR*.
[70] Eric V Slud. 1977. Distribution inequalities for the binomial law. *The Annals of Probability* (1977).
[71] Mea Wang, Baochun Li, and Zongpeng Li. 2004. sFlow: Towards Resource-Efficient and Agile Service Federation in Service Overlay Networks. In *Proc. of IEEE ICDCS*.
[72] Li Yang, Wu Hao, Pan Tian, Dai Huichen, Lu Jianyuan, and Liu Bin. 2016. CASE: Cache-assisted Stretchable Estimator for High Speed Per-flow Measurement. In *Proc. of IEEE INFOCOM*.
[73] Tong Yang, Jie Jiang, Peng Liu, Qun Huang, Junzhi Gong, Yang Zhou, Rui Miao, Xiaoming Li, and Steve Uhlig. 2018. Elastic Sketch: Adaptive and Fast Network-wide Measurements. In *Proc. of ACM SIGCOMM*.
[74] Lei Ying, R. Srikant, and Xiaohan Kang. 2015. The Power of Slightly More than One Sample in Randomized Load Balancing. In *Proc. of IEEE INFOCOM*.
[75] Da Yu, Yibo Zhu, Behnaz Arzani, Rodrigo Fonseca, Tianrong Zhang, Karl Deng, and Lihua Yuan. 2019. dShark: A General, Easy to Program and Scalable Framework for Analyzing In-network Packet Traces. In *Proc. of USENIX NSDI*.
[76] Minlan Yu, Lavanya Jose, and Rui Miao. 2013. Software Defined Traffic Measurement with OpenSketch. In *Proc. of USENIX NSDI*.

A ANALYSIS OF ALWAYSCORRECT NITROSKETCH

We now formally analyze the accuracy guarantees of AlwaysCorrect NitroSketch (Algorithm 1). We start with Lemma 6 that shows that once AlwaysCorrect NitroSketch converges (see Line 14), the L_2 is large enough to justify sampling with probability $p_{\min}$. We analyze the worst case scenario here where once starting to sample we always use the smallest probability and for convenience denote $p \equiv p_{\min}$.

LEMMA 6. *When AlwaysCorrect NitroSketch starts sampling:*

$$\Pr\left[L_2 \geq 11\epsilon^{-2}p^{-1}\right] \geq 1-\delta.$$

PROOF. Since L_2 grows monotonically with the number of packets, it is enough to show that the condition of Line 14 implies the lower bound on the L_2 value. Namely, we assume that

$$\text{median}_{i\in[d]} \sum_{y=1}^{w} C_{i,y}^2 > 121(1+\epsilon\sqrt{p})\epsilon^{-4}p^{-2}. \tag{1}$$

It is known that given a Count Sketch that is configured for a (ϵ', δ)-guarantee, it is possible to compute a $(1+\epsilon')$-approximation of the L_2 with probability $1-\delta$ [5]. Specifically, as throughout the processing of $\overline{S}$ our sketch is identical to a Count Sketch (for $\epsilon' = \epsilon\sqrt{p}$), we have that:

$$\Pr\left[\left|\left(\text{median}_{i\in[d]} \sum_{y=1}^{w} C_{i,y}^2\right) - L_2^2\right| > \epsilon' L_2^2\right] \leq \delta.$$

Combining this with (1), the lemma follows. □

In AlwaysCorrect NitroSketch, there are $d = O(\log \delta^{-1})$ rows, each having $w = 11\epsilon^{-2}p^{-1}$ counters. As long as the sketch has not 'converged' (see Line 14), it is indistinguishable to a Count Sketch [17] with a guarantee of $\epsilon' \triangleq \epsilon\sqrt{p}$. Thus, given a flow x, if $converged = 0$ then Algorithm 1 guarantees:

$$\Pr\left[|\widehat{f_x} - f_x| \leq \epsilon L_2'\right] \leq \delta.$$

As $\epsilon' = \epsilon\sqrt{p} \leq \epsilon$ the algorithm provides the desired accuracy guarantee prior to convergence. Henceforth, we assume that we converged and show that the error is still at most ϵL_2.

We denote by u the index of the packet that during its processing the condition in Line 14 was satisfied and the sketch converged. That is, packets $a_i, \ldots, a_u$ were processed using a UPDATE(1), while $a_{u+1}, \ldots, a_m$ followed a UPDATE(p). Further, we denote by $\overline{S} \triangleq a_1, \ldots, a_u$ the substream of the first u packets, by $\ddot{S} \triangleq a_{u+1}, \ldots, a_m$ the remaining substream, and for a flow x we use $\overline{f_x}$ and $\ddot{f}_x$ to denote its frequency in $\overline{S}$ and $\ddot{S}$. Note that the overall frequency of x is $f_x = \overline{f_x} + \ddot{f}_x$. Additionally, we denote the number of times a packet that belongs to a flow x in $\ddot{S}$ was sampled by the i'th row as $\ddot{f}_{x,i}$. Similarly to the analysis of Theorem 2, we first analyze the guarantee provided by a single row. Namely, fix some flow $x \in \mathcal{U}$ and a row $i \in \{1, \ldots, d\}$; the counter associated with x on this row is $C_{i,h_i(x)}$. Observe that we can express the value of the i'th estimator as:

$$C_{i,h_i(x)}g_i(x) = \sum_{y:h_i(y)=h_i(x)} \overline{f_y} g_i(x)g_i(y) + p^{-1} \cdot \sum_{y:h_i(y)=h_i(x)} \ddot{f}_{y,i} g_i(x)g_i(y). \tag{2}$$

That is, every flow y that is mapped to the same counter as x (i.e., $h_i(y) = h_i(x)$) changes the estimation by $\overline{f_y}g_i(x)g_i(y) + p^{-1}\ddot{f}_{y,i}g_i(x)g_i(y)$ – every packet of y in $\overline{S}$ surely adds $g_i(y)$ to the counter (Algorithm 1, Line 13), while every *sampled* packet in $\ddot{S}$ modifies the counter by $p^{-1}g_i(y)$ (Algorithm 1, Line 20).

Next, we denote $A \triangleq \sum_{y:h_i(y)=h_i(x)} \overline{f_y} g_i(x)g_i(y)$ and $B \triangleq p^{-1} \cdot \sum_{y:h_i(y)=h_i(x)} \ddot{f}_{y,i} g_i(x)g_i(y)$ (i.e., $C_{i,h_i(x)}g_i(x) = A + B$). We note that A and B are independent and that $\mathbb{E}\left[C_{i,h_i(x)}g_i(x)\right] = \mathbb{E}[A] + \mathbb{E}[B] = \overline{f_x} + \ddot{f}_x = f_x$. That is, the resulting estimator for row i is unbiased.

We now turn to bound the variance of the estimator by bounding $\text{Var}[A - \overline{f_x}]$ and $\text{Var}[B - p^{-1}\ddot{f}_{x,i}]$. First, since

$$\Pr[h_i(x) = h_i(y)] = 1/w$$

for $x \neq y$, observe that:

$$\text{Var}[A - \overline{f_x}] = \text{Var}\left[\sum_{y:h_i(y)=h_i(x)} \overline{f_y} g_i(x)g_i(y) - \overline{f_x}\right]$$

$$= \mathrm{Var}\left[\sum_{y\neq x: h_i(y)=h_i(x)} \overline{f_y} g_i(x) g_i(y)\right]$$
$$= \mathbb{E}\left[\sum_{y\neq x: h_i(y)=h_i(x)} \overline{f_y}^2\right] \leq 1/w \sum_{y\in\mathcal{U}} \overline{f_y}^2. \quad (3)$$

Next, let us analyze the variance of $B - p^{-1}\ddot{f}_{x,i}$:

$$\mathrm{Var}[B - p^{-1}\ddot{f}_{x,i}]$$
$$= \mathrm{Var}\left[p^{-1}\cdot \sum_{y: h_i(y)=h_i(x)} \ddot{f}_{y,i} g_i(x) g_i(y) - p^{-1}\ddot{f}_{x,i}\right]$$
$$= \mathrm{Var}\left[p^{-1}\cdot \sum_{y\neq x: h_i(y)=h_i(x)} \ddot{f}_{y,i} g_i(x) g_i(y)\right]$$
$$= \mathbb{E}\left[p^{-2}\cdot \sum_{y\neq x: h_i(y)=h_i(x)} \ddot{f}_{y,i}^2\right] \leq p^{-2}/w \cdot \mathbb{E}\left[\sum_{y\in\mathcal{U}} \ddot{f}_{y,i}^2\right]. \quad (4)$$

Similarly to Lemma 3, we have that $\mathbb{E}\left[\sum_{y\in\mathcal{U}} \ddot{f}_{y,i}^2\right] \leq 2p\sum_{y\in\mathcal{U}} \ddot{f}_y^2$, which allows reduce (4) to

$$\mathrm{Var}[B - p^{-1}\ddot{f}_{x,i}] \leq 2p^{-1}/w \cdot \sum_{y\in\mathcal{U}} \ddot{f}_y^2. \quad (5)$$

Recall that during the processing of $\ddot{S}$, every packet is sampled with probability p and thus $\ddot{f}_{x,i} \sim Bin(\ddot{f}_x, p)$. Putting everything together we get:

$$\mathrm{Var}\left[C_{i,h_i(x)} g_i(x) - f_x\right] = \mathrm{Var}[A + B - f_x]$$
$$= \mathrm{Var}[(A - \overline{f_x}) + (B - \ddot{f}_x)]$$
$$= \mathrm{Var}[(A - \overline{f_x}) + (B - p^{-1}\ddot{f}_{x,i}) + (p^{-1}\ddot{f}_{x,i} - \ddot{f}_x)]$$
$$= \mathrm{Var}[(A - \overline{f_x})] + \mathrm{Var}[(B - p^{-1}\ddot{f}_{x,i})] + \mathrm{Var}[(p^{-1}\ddot{f}_{x,i} - \ddot{f}_x)]$$
$$\leq 1/w \sum_{y\in\mathcal{U}} \overline{f_y}^2 + 2p^{-1}/w \cdot \sum_{y\in\mathcal{U}} \ddot{f}_y^2 + \ddot{f}_x p^{-1}$$
$$\leq \frac{L_2^2\cdot\left(1+2p^{-1}\right)}{w} + \ddot{f}_x p^{-1} \leq \frac{L_2^2\cdot\left(3p^{-1} + p^{-1}w/L_2\right)}{w}. \quad (6)$$

Here, the last inequality follows as $\ddot{f}_x \leq f_x \leq L_2$. We now use Lemma 6 to get that with a very high probability, $L_2 > w$. Intuitively, this follows from our convergence criteria (Algorithm 1, Line 14). This means that *conditioned on* $L_2 > w$ (which happens with probability $1-\delta$), we have that

$$\mathrm{Var}\left[C_{i,h_i(x)} g_i(x) - f_x\right] \leq \frac{L_2^2\cdot\left(3p^{-1} + p^{-1}w/L_2\right)}{w}$$
$$\leq \frac{L_2^2\cdot 4p^{-1}}{w} \leq 3L_2^2\epsilon^2/8. \quad (7)$$

We now use Chebyshev's inequality to conclude that the estimator of the i'th row, $\widehat{f_x}(i)$, satisfies

$$\Pr\left[|\widehat{f_x}(i) - f_x| \geq \epsilon L_2\right] = \Pr\left[|C_{i,h_i(x)} g_i(x) - f_x| \geq \epsilon L_2\right]$$
$$\leq \frac{\mathrm{Var}\left[C_{i,h_i(x)} g_i(x) - f_x\right]}{(\epsilon L_2)^2} \leq 3/8. \quad (8)$$

That is, the probability that each row estimates the frequency of x with an error no larger than $L_2\epsilon$ is at least $5/8$. Finally, the standard use of Chernoff's inequality shows that $d = O(\log\delta^{-1})$ (independent) rows are required for their median to amplify the probability to $1-\delta$. Taking the union bound over the events of sampling too early and having an error in the row's median, we have an error probability no larger than 2δ. This concludes the proof of Theorem 5.

B COMPARISON TO UNIFORM SAMPLING

Our sketch updates each row, for every packet, with probability p. An alternative approach, *uniform sampling*, would be updating *all rows* with probability $1/p$. We note that the two approaches make the same number of hash computations in expectation. Here, we claim that our approach is superior to that of uniform sampling.

Intuitively, our sketch uses the fact that for each row i, with probability 3/4 we have $L_{2,i} = O(\sqrt{p}L_2)$. This reduction in the second norm allows one to increase the row width by a factor of p (compared to Count Sketch) to make up for the extra error introduced by the sampling. We now show that uniform sampling requires asymptotically more space as the second norm of the sampled substream is expected to be $\Omega\left(L_2\sqrt{p\log\delta^{-1}}\right)$ with probability $\Omega(\delta^{-1})$. Since Count Sketch is known to have an error of $\Omega\left(\overline{L_2}/\sqrt{w}\right)$ for streams with a second norm of $\overline{L_2}$, we get that for an error of ϵL_2 one would need to use more counters per row, or wait longer for the algorithm to converge. That is, a uniform sampling with the same update rate would require a multiplicative $\Omega\left(\log\delta^{-1}\right)$ more space.

To begin, we first discuss a *lower bound* on the error of Count Sketch. In Count Sketch, one uses a matrix of $\overline{w}$ columns and $\overline{d} = O(\log\delta^{-1})$ to get $\Pr\left[|\widehat{f_x} - f_x| \geq L_2/\sqrt{\overline{w}}\right] \leq \delta$. For an ϵL_2 guarantee, one then sets $\overline{w} = O(\epsilon^{-2})$. We now show that this is asymptotically tight. Namely, we show that there exists a distribution for which $\Pr\left[|\widehat{f_x} - f_x| \geq \epsilon L_2\right] = \Omega(\delta)$.

To prove our result, we use the following theorem.

THEOREM 7. *([35, 59]) Let X be a binomial variable such that $\mathrm{Var}[X] \geq 40000$. Then for all $t \in [0, \mathrm{Var}[X]/100]$, we have*

$$\Pr[X \geq E[X] + t] = \Omega\left(e^{-t^2/3\mathrm{Var}[X]}\right)$$

We are now ready to show a lower bound on the error of Count Sketch.

LEMMA 8. *Let $n \geq m+1$. Consider Count Sketch allocated with $\overline{d} = O(\log\delta_1^{-1})$ rows and $\overline{w} \leq m/c'$ columns, for a sufficiently large constant[2] c'. There exists $c = \Theta(1)$, a stream $S \in [n]^{[m]}$, and an element $x \in [n]$ such that $\Pr\left[|\widehat{f_x} - f_x| \geq c\cdot L_2/\sqrt{\overline{w}}\right] \geq \delta_1$.*

PROOF. We denote by c'' the constant in the $\Omega(\cdot)$ of Theorem 7, and by $z = O(1)$ the constant such that $\overline{d} = z\ln\delta_1^{-1}$. Let $c' = \max\left\{320000, -8\ln\left(1 - e^{-1/2z}/c''\right)\right\}$ and $c = \sqrt{\frac{3}{4z}}$ be two constants. We will show that with probability of at least e^{-z}, each row has an error of at least $c\cdot L_2/\sqrt{\overline{w}}$. This would later allow us to

[2] In practice, $w \ll m$, as otherwise we have enough memory for exact counting and would not need sketches, and this trivially holds.

conclude that the estimation, which is the median row, has an error of $c \cdot L_2/\sqrt{w}$ with probability of at least δ.

Consider the stream in which all elements of $[m]$ arrive once each (and thus, $L_2 = \sqrt{m}$), and consider a query for $x \triangleq m+1$ (i.e., $f_x = 0$). Fix a row i, and let
$Q \triangleq \{j \in [m] \mid h_i(j) = h_i(x)\}$ be the elements that affect x's counter on the i'th row. Intuitively, we show that the number of items that change x's counter ($C_{i,h_i(x)}$) is $|Q| = \Omega(L_2/w)$ and then give a lower bound on the resulting value of the counter (given that some of the flows in Q increase it while others decrease). Observe that $|Q| \sim Bin(m, 1/\overline{w})$. According to Chernoff's bound:

$$\Pr\left[|Q| \leq m/2\overline{w}\right] \leq e^{-m/8\overline{w}} \leq e^{-c'/8} \leq 1 - e^{-1/2z}/c''. \quad (9)$$

Next, we denote by $X \triangleq \{j \in Q \mid g_i(j) = +1\}$ the number of elements from Q that increased the value of x's counter. Observe that $X \sim Bin(|Q|, 1/2)$ is binomially distributed and that x's counter satisfies $c_{i,h_i(x)} = 2X - |Q|$. Conditioned on the event $|Q| > m/2\overline{w}$ (which happens with constant probability as (9) shows), we have that $\mathrm{Var}[X] = |Q|/4 \geq m/8\overline{w} = c'/8 \geq 40000$. According to Theorem 7, we now have that

$$\Pr\left[X \geq \mathbb{E}[X] + t \;\middle|\; |Q| > m/2\overline{w}\right] \geq c''e^{-t^2/3\mathrm{Var}[X]} \quad (10)$$

some $c'' > 0$ and any $t \in [0, \mathrm{Var}[X]/100]$. We will now show that in each row i, $\Pr\left[c_{i,h_i(x)} \geq c \cdot L_2/\sqrt{w}\right] \geq e^{-1/z}$, for $c = \sqrt{\frac{3}{4z}}$.

$$\begin{aligned}
&\Pr\left[c_{i,h_i(x)} \geq c \cdot L_2/\sqrt{\overline{w}}\right] = \Pr\left[2X - |Q| \geq c \cdot \sqrt{m/\overline{w}}\right] \\
&= \Pr\left[X \geq (|Q| + c \cdot \sqrt{m/\overline{w}})/2\right] = \Pr\left[X \geq \mathbb{E}[X] + c \cdot \sqrt{m/4\overline{w}}\right] \\
&\geq \Pr\left[\left(X \geq \mathbb{E}[X] + c \cdot \sqrt{m/4\overline{w}}\right) \wedge \left(|Q| > m/2\overline{w}\right)\right] \\
&= \Pr\left[\left(X \geq \mathbb{E}[X] + c \cdot \sqrt{m/4\overline{w}}\right) \;\middle|\; \left(|Q| > m/2\overline{w}\right)\right] \Pr\left[|Q| > m/2\overline{w}\right] \\
&\geq \Pr\left[\left(X \geq \mathbb{E}[X] + c \cdot \sqrt{m/4\overline{w}}\right) \;\middle|\; \left(|Q| > m/2\overline{w}\right)\right] \cdot e^{-1/2z}/c''.
\end{aligned} \quad (11)$$

Setting $t \triangleq c \cdot \sqrt{m/4\overline{w}} = O(\sqrt{\mathrm{Var}[X]})$ and using (10), we get that

$$\begin{aligned}
&\Pr\left[\left(X \geq \mathbb{E}[X] + c \cdot \sqrt{m/4\overline{w}}\right) \;\middle|\; \left(|Q| > m/2\overline{w}\right)\right] \\
&\geq c''e^{-(c\cdot\sqrt{m/4\overline{w}})^2/3\mathrm{Var}[X]} \geq c''e^{-(c\cdot\sqrt{m/4\overline{w}})^2/3(m/8\overline{w})} \\
&= c''e^{-2c^2/3} = c''e^{-1/2z},
\end{aligned}$$

where the last inequality follows from $\mathrm{Var}[X] = |Q|/4 \geq m/8\overline{w}$. Plugging this back into (11) we get

$$\begin{aligned}
&\Pr\left[c_{i,h_i(x)} \geq c \cdot L_2/\sqrt{\overline{w}}\right] \\
&\geq \Pr\left[\left(X \geq \mathbb{E}[X] + c \cdot \sqrt{m/4\overline{w}}\right) \;\middle|\; \left(|Q| > m/2\overline{w}\right)\right] e^{-1/2z}/c'' \\
&\geq c''e^{-1/2z}e^{-1/2z}/c'' = e^{-1/z}.
\end{aligned}$$

Thus, we established that in each row i with a probability of at least e^{-z}, x's counter (and thus, the error) is larger than $c \cdot L_2/\sqrt{\overline{w}}$. Finally, since the rows are independent, we get that the probability of Count Sketch returning a wrong estimate is at least

$$\Pr\left[\widehat{f_x} - f_x \geq c \cdot L_2/\sqrt{\overline{w}}\right] \geq \left(e^{-1/z}\right)^{\overline{d}} = \left(e^{-1/z}\right)^{z\ln\delta_1^{-1}} = \delta_1.$$

□

To proceed, we need some inequalities that allow us to provide a lower bound on the reduction in L_2 of the sub-sampled stream. To that end, we use the following results:

THEOREM 9. *([70]) Let $X \sim Bin(n,p)$; for all k such that $np \leq k \leq n(1-p)$:*

$$\Pr[X \geq k] \geq 1 - \Phi\left(\frac{k-np}{\sqrt{np(1-p)}}\right),$$

where $\Phi(z) \triangleq \int_{-\infty}^{z} \frac{1}{\sqrt{2\pi}} e^{-z^2/2}$ is the cumulative distribution function of the normal distribution.

THEOREM 10. *([37]) For any $z > 0$:*

$$1 - \Phi(z) > \frac{z}{1+z^2}\phi(z),$$

where $\phi(z) \triangleq \frac{1}{\sqrt{2\pi}} e^{-z^2/2}$ is the density function of the normal distribution.

For convenience, we also use the following fact:

FACT 1. *For any $z \geq 2$:*

$$\frac{z}{1+z^2}\phi(z) = \frac{z}{1+z^2}\frac{1}{\sqrt{2\pi}}e^{-z^2/2} \geq e^{-z^2}$$

Next, we will provide a lower bound on the reduction in L_2 when sub-sampling a stream with probability p. Once again, we consider the stream S in which m distinct elements arrived once each (and thus its L_2 is $\sqrt{m}$).

LEMMA 11. *Let $\overline{S}$ be a substream of S such that each packet in S appears in $\overline{S}$ independently with probability $p \leq 1/2$. Denote by L_2^S the L_2 of S and by $L_2^{\overline{S}}$ the L_2 of $\overline{S}$. Then for $\delta_2 \leq 1/4$:*

$$\Pr\left[L_2^{\overline{S}} \geq \sqrt{mp + \sqrt{mp(1-p)\log\delta_2^{-1}}}\right] \geq \delta_2.$$

PROOF. Denote by J the set of sampled elements; observe that $|J| \sim Bin(m,p)$ and that $L_2^{\overline{S}} = \sqrt{|J|}$. According to Theorem 9, Theorem 10, and Fact 1, we have that:

$$\Pr\left[|J| \geq mp + \sqrt{mp(1-p)\log\delta_2^{-1}}\right] \geq \delta_2.$$

Thus, we have that:

$$\begin{aligned}
&\Pr\left[L_2^{\overline{S}} \geq \sqrt{mp + \sqrt{mp(1-p)\log\delta_2^{-1}}}\right] \\
&= \Pr\left[\sqrt{|J|} \geq \sqrt{mp + \sqrt{mp(1-p)\log\delta_2^{-1}}}\right] \\
&= \Pr\left[|J| \geq mp + \sqrt{mp(1-p)\log\delta_2^{-1}}\right] \geq \delta_2.
\end{aligned}$$

□

The above lemma shows that the L_2 of the uniformly sub-sampled stream is larger than $\sqrt{mp + \sqrt{mp(1-p)\log\delta_2^{-1}}}$ with probability $\geq \delta_2$. In contrast, in our sketch every row processes a sub-stream with an L_2 of $O(F_2\sqrt{p})$ (i.e., $O(\sqrt{mp})$ for this stream) with a constant probability, *independently from the other rows*. We now show that

in some cases (when the desired error probability is small), the dependence between the rows in the case of uniform samples requires asymptotically more space than our sketch, for the same error guarantee. Therefore, we claim that our sketch has clear advantages over uniform sampling.

THEOREM 12. *Let $\overline{S}$ be a substream of S such that each packet in S appears in $\overline{S}$ independently with probability p. There exists a stream S such that Count Sketch with $d = \Theta(\log \delta^{-1})$ rows applied on $\overline{S}$ requires*

$$w = \Omega\left(\epsilon^{-2}p^{-1} + \epsilon^{-2}p^{-1.5}m^{-0.5}\sqrt{\log \delta^{-1}}\right)$$

counters per row to provide (with probability $1-\delta$) an ϵL_2 error for S.

PROOF. We set $\delta_1 = \delta_2 = \sqrt{\delta}$ (and thus $\log \delta_1^{-1}, \log \delta_2^{-1} = \Theta\left(\log \delta^{-1}\right)$). According to Lemma 8, we have that there exists $c = \Theta(1)$ such that:

$$\Pr\left[|\widehat{f_x} - f_x| \geq c \cdot L_2^{\overline{S}}/\sqrt{w}\right] \geq \delta_1$$

Next, we use Lemma 11 to obtain

$$\Pr\left[L_2^{\overline{S}} \geq \sqrt{mp + \sqrt{mp(1-p)\log \delta_2^{-1}}}\right] \geq \delta_2.$$

Since the Count Sketch uses randomization that is independent from the stream sampling, we have that

$$\Pr\left[\left(L_2^{\overline{S}} \geq \sqrt{mp + \sqrt{mp(1-p)\log \delta_2^{-1}}}\right) \wedge \left(|\widehat{f_x} - f_x| \geq c \cdot L_2^{\overline{S}}/\sqrt{w}\right)\right] \geq \delta_1\delta_2 = \delta. \quad (12)$$

Thus, with probability of at least δ, the error of the Count sketch is

$$\Omega\left(\sqrt{\frac{mp + \sqrt{mp(1-p)\log \delta^{-1}}}{w}}\right).$$

Next, recall that to estimate the frequencies in the original stream S, one needs to divide the Count Sketch estimate by p. Thus, if we denote the resulting estimation error by $\mathfrak{E}_x$ we have that

$$\Pr\left[\mathfrak{E}_x = \Omega\left(p^{-1}\sqrt{\frac{mp + \sqrt{mp(1-p)\log \delta^{-1}}}{w}}\right)\right] \geq \delta.$$

To provide an $\epsilon L_2 = \epsilon\sqrt{m}$ guarantee, uniform sampling Count Sketch needs to set w such that $\Pr\left[\mathfrak{E}_x \geq \epsilon\sqrt{m}\right] \leq \delta$. Demanding

$$p^{-1}\sqrt{\frac{mp + \sqrt{mp(1-p)\log \delta^{-1}}}{w}} = \epsilon\sqrt{m}$$

the bound follows. □

We therefore conclude that while our sketch requires $O(\epsilon^{-2}p^{-1}\log \delta^{-1})$ counters overall, inserting a uniform sample into Count Sketch, for the same sampling probability p and error guarantee, requires at least

$$\Omega\left(\epsilon^{-2}p^{-1}\log \delta^{-1} + \epsilon^{-2}p^{-1.5}m^{-0.5}\log^{1.5} \delta^{-1}\right).$$