Catching the Microburst Culprits with Snappy

Xiaoqi Chen
Princeton University
xiaoqic@cs.princeton.edu

Shir Landau Feibish
Princeton University
sfeibish@cs.princeton.edu

Yaron Koral
AT&T Labs
yk216h@att.com

Jennifer Rexford
Princeton University
jrex@cs.princeton.edu

Ori Rottenstreich
Technion
or@cs.technion.ac.il

ABSTRACT

Short-lived traffic surges, known as microbursts, can cause periods of unexpectedly high packet delay and loss on a link. Today, preventing microbursts requires deploying switches with larger packet buffers (incurring higher cost) or running the network at low utilization (sacrificing efficiency). Instead, we argue that switches should detect microbursts as they form, and take corrective action before the situation gets worse. This requires an efficient way for switches to identify the particular flows responsible for a microburst, and handle them automatically (e.g., by pacing, marking, or rerouting the packets). However, collecting fine-grained statistics about queue occupancy in real time is challenging, even with emerging programmable data planes. We present Snappy, which identifies the flows responsible for a microburst in real time. Snappy maintains multiple snapshots of the occupants of the queue over time, where each snapshot is a compact data structure that makes efficient use of data-plane memory. As each new packet arrives, Snappy updates one snapshot and also estimates the fraction of the queue occupied by the associated flow. Our simulations with data-center packet traces show that Snappy can target the flows responsible for microbursts at the sub-millisecond level.

CCS CONCEPTS

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

ACM Reference Format:
Xiaoqi Chen, Shir Landau Feibish, Yaron Koral, Jennifer Rexford, and Ori Rottenstreich. 2018. Catching the Microburst Culprits with Snappy. In *SelfDN 2018: ACM SIGCOMM 2018 Afternoon Workshop on Self-Driving Networks , August 24, 2018, Budapest, Hungary*. ACM, New York, NY, USA, 7 pages. https://doi.org/10.1145/3229584.3229586

1 INTRODUCTION

Queue utilization in network switches remains a major concern for network administrators. Large queues cause packet loss and delay, leading to performance degradation. Even on a link with low

SelfDN 2018, August 24, 2018, Budapest, Hungary

ACM ISBN 978-1-4503-5914-6/18/08…$15.00
https://doi.org/10.1145/3229584.3229586

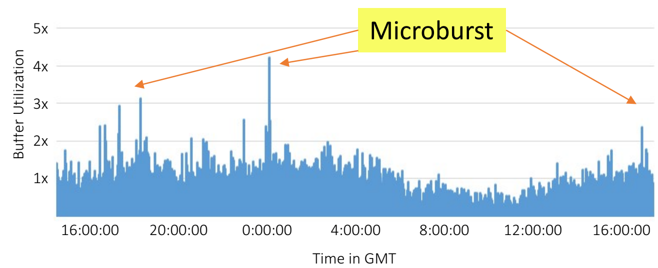

Figure 1: Carrier grade network router buffer utilization measurements. The y-axis indicates the increase in buffer utilization compared to the average usage.

average utilization, a large queue can arise due to a *microburst*—a short-lived spike of legitimate or adversarial traffic that exceeds the average volume by several orders of magnitude. In data-center networks, microbursts quickly cause queues to become fully utilized, leading to immediate packet loss [5]. Microbursts also pose challenges for network planning in carrier networks. While router buffers are extremely underutilized most of the time, and studies show that shorter buffers should be sufficient [3, 14], the long-tail nature of the traffic still introduces significant microbursts. Figure 1 shows an example of router buffer utilization measurements over a 24-hour time period in a carrier network. As can be seen, some of the bursts cause a 4x increase in buffer utilization, compared to the average traffic volume, whereas most of the time buffer utilization does not surpass a factor of 2x.

To maintain high quality of service during microbursts, administrators are forced to deploy equipment with larger buffers and run their networks at lower utilization, hence incurring higher cost. While *preventing* microbursts is the obvious goal, even *detecting* them in time poses a significant challenge. Today's state-of-the-art commercial network equipment reports traffic statistics at the scale of minutes or at best seconds, while observing a microburst requires monitoring at the scale of microseconds. These measurement techniques rely on exporting raw, predefined measurements from the data plane, to be analyzed in the control plane. Exporting information at the millisecond timescale requires moving tremendous amounts of data, which is very expensive and harms the network performance. Furthermore, the short time scale of microbursts may sometimes make controller assisted remediation less adequate.

Microbursts have been defined in a number of ways, based on the congestion [11] or loss [6] that they cause. We focus on the length of the queue as the backlog builds; here, a microburst is a group of packets that consume a significant fraction of the traffic in the queue when the queue has passed a given threshold length. This allows us to detect microbursts as they form, therefore allowing the network device to quickly take action to mitigate them, before the queue is full and packet loss is inevitable. We focus on identifying the weight of the individual flows in the queuing buffer, and particularly heavy-hitter flows that consume a significant fraction of the queuing buffer during a microburst. These weights could be used to mitigate microbursts by, say, dropping a packet with probability proportionally to its flow's contribution to the queue length. Alternatively, we could mark ECN flags only on those flows contributing to a significant fraction of queue during congestion. With the new capabilities provided by programmable switches, detecting these forming bursts is now possible, and we present a mechanism which does so quickly, right in the data plane. Such timely detection of microbursts could be especially useful for detecting different types of Denial of Service (DoS) attacks, such as low-rate TCP-targeted DoS attacks [16].

A straightforward approach for detecting the significant flows causing the microbursts would require tracking the volume of each flow in the queue. This in turn requires maintaining per-flow state and updating the information on packet arrivals and departures. This approach is not realistic even with programmable switches, as we will discuss in Section 2.2. Fortunately, we can exploit three relaxations to the general problem of measuring queue occupancy, i.e., measuring the exact volume of each flow in the queue:

(1) **Perform detection only when the queue is long**: This allows us to use approximation techniques. In particular, we divide groups of incoming packets into snapshots, and estimate the queue's content by looking some number of snapshots. This may cause large relative error when the queue is very short, but can yield a reasonable approximation when the queue is sufficiently long.
(2) **Target only the heavy flows**: Since we are only concerned with detecting the large flows, we can use sketches or other approximation techniques, removing the need to keep per-flow state. Hence, our snapshots are sketches of the sizes of the flows in the queue.
(3) **Take action directly in the data plane**: The switch acts on incoming packets (e.g., by marking, dropping, or rate limiting) that belong to heavy flows. Since we only need to identify heavy flows when an associated packet arrives, we do not need to store and report flow identifiers.

Based on these insights, we present Snappy, a scalable framework for detecting microbursts quickly, within the data plane. The Snappy framework periodically records queue snapshots with incoming packets. These snapshots consist of sketches of part of the queue, and allow us to effectively estimate the queue's content when the queue is experiencing an ongoing build-up due to a burst of traffic. By using approximate snapshots, the detection algorithm is scalable and highly efficient even for high capacity routers. Our technique can run on commodity programmable switches, as we explain in detail in Section 2.

We evaluate Snappy via simulation with real packet traces. Snappy is capable of reacting to sub-millisecond queue buildup, and can capture several types of microburst culprit flows such as flows that surpass a certain threshold or heavy hitters that consist of a certain fraction of the queue. Simulation evaluation using data center network trace shows Snappy achieves high accuracy (≥ 90% precision and recall) when identifying culprit flows during microbursts, using 10 snapshots each consuming less than 1 KB of stateful memory, a reasonable resource consumption in programmable switches.

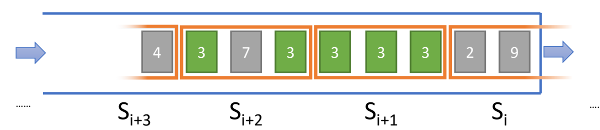

Figure 2: Three-packet snapshots of queue occupants.

2 SNAPPY FRAMEWORK

Our discussion assumes one link with a single FIFO queue with capacity (maximum queue length) C. To simplify the discussion, we assume unit-sized packets; it is straightforward to extend our solution to variable packet sizes.

2.1 Heavy Hitters With Subtraction

To answer which flow is occupying significant queuing buffer space is essentially solving the Heavy Hitters Detection problem, but *with subtractions*: a packet's size is added to its flow's size when it enters the queue, and a packet exiting the queue should be *subtracted* from its flow size. Subsequently, we can identify which flow is occupying a significant fraction of the entire queue.

We first present an ideal algorithm to answer this problem. The ideal algorithm maintains a key-value table mapping flow IDs to flow size counters. Whenever a packet p of flow f enters the queue, we increment its appropriate counter: $count[f]$+=1. At the other end of the queue, for each departing packet p' of flow f', we decrease its counter: $count[f']$-=1. If the current queue length l exceeds a threshold, we would like to find all flows occupying more than (say) 1% of the buffer space (i.e. flows with $count[f] \geq 1\% \times l$).

However, the ideal algorithm requires simultaneous update to the data structure from both ends of the queue, as packets are constantly entering and leaving the queue. Such simultaneous update to the data structure is impractical to implement in any of today's high throughput switches.

2.2 PISA Constraints

A Protocol Independent Switch Architecture (PISA) switch is composed of a pipeline of stages, and each stage consists of a match-action table and a fixed amount of state. In order to maintain line-speed packet processing, the amount of work that can be performed at each stage is limited. A typical high-performance PISA switch may have $4-32$ hardware stages, each with access to $O(10MB)$ stateful memory.

The PISA architecture poses many constraints for algorithm implementation. We highlight some of these constraints, which are important for understanding the design challenges and decisions in Snappy. PISA allows only constant-time actions at each stage, and the number of hardware stages is limited. Furthermore, the overall amount of memory is limited, thus making it impractical to maintain accurate per-flow counters within the data plane.

Additionally, in order to prevent concurrent memory access to a single memory location, each stateful memory location may only be accessed from one particular stage of the packet processing pipeline. Due to this constraint, PISA does not allow access to the same memory (or data) twice in a pipeline. Therefore, as a packet traverses the pipeline, it may only access a register in a single stage of the pipeline, meaning that a register cannot be accessed both when a packet enters the queue and as it leaves the queue. We remove the second memory access (and eliminate the need for simultaneous updates to the data structure) by introducing snapshots in Section 2.3. We address other hardware limitations mentioned above in Section 2.4 and 2.5.

Additionally, current P4 specifications [8, 9] are quite vague regarding the structure and behaviour of the queuing mechanism. Queuing dynamics information such as queue capacity and utilization are not necessarily accessible from either ingress or egress pipeline. However, from our study of current implementations, we observed that often a packet does not have access to the queue length during the ingress pipeline. This is because routing decisions are finalized only after the packet has pass through the entire ingress pipeline. Meanwhile, in commodity programmable switches, current queue length and packet queue-arrival time are generally available at the egress pipeline. Therefore, our algorithm should be performed in the egress pipeline to allow access to this crucial information.

2.3 Queue Snapshots for Batch Subtraction

When a packet arrives in a constant-throughput FIFO queue, we know the time it will exit the queue, based on current queue length l. Similarly, given current queue length l, we can observe the content of the queue by looking into l most recently arrived packets. Although arbitrary access to exactly l past packet arrivals may be unfeasible, we can approximate this by partitioning the arriving packet stream into snapshot windows. We present the first key component of Snappy framework, *snapshots*, illustrated in Figure 2, as a solution to avoid concurrent memory update while observing queue occupancy. Instead of maintaining one data structure and subtracting packets from it, we maintain many snapshots, each capturing a window of w bytes of traffic. When a packet arrives, we add it to the most recent snapshot; after w bytes of traffic have arrived, we advance to a new snapshot. We exploit the FIFO property, which guarantees that packets exit the queue in the same order in which they enter the queue.

We denote $[f]$ as rounding value f to the nearest integer. When the queue length is l, we can combine the most recent $[\frac{l}{w}]$ snapshots to approximate the content of the queue and find heavy flows. When the queue is longer due to more severe congestion, we look at more snapshots. Combining snapshots inevitably causes some rounding errors near the head of the queue. If the queue length is shorter than one snapshot, the relative rounding error can become large; however, since we focus on microburst-caused congestion, the queue is rather long, and the rounding error is less significant.

An old snapshot is simply ignored after all its packets have left the queue, equivalent to batch-subtracting those packets from the estimate. Thus, we avoid the need to update any part of the data structure twice.

2.4 Approximate Snapshot Data Structure

PISA switches do not support maintaining per-flow counters for all flows directly in the data plane. Fortunately, to catch microburst culprits, accurately estimating the size of heavy flows would suffice. We can use an approximate data structure to estimate flow statistics while satisfying architectural limitation on memory access. To enable actionable mitigation during microbursts, all we need is to recognize that an arriving packet belongs to a heavy flow in the queue.

One popular option to use in PISA is the Count-Min Sketch (CMS) [10]. A CMS maintains r rows with b counter buckets in each row. For each packet being added to CMS, the packet ID is hashed by r different hash functions to locate one bucket at each row, and its size is added to those buckets. To estimate flow size given a flow ID, we gather the value of those buckets and compute their minimum.

We implement each snapshot as a Count-Min Sketch. When a new packet enters the queue, it is added to the CMS of the current snapshot, and also looks up estimated flow size from the CMS of previous snapshots. Based on the estimated flow size, we can decide if this packet belongs to a heavy flow in the queue.

We note that the CMS does not keep state-per-flow and therefore does not maintain the flow IDs. The ID of a heavy flow is extracted from subsequent packets of that flow, when they arrive at the switch. These packets are used to "report" their own flow as being a heavy flow. If no subsequent packets of a heavy flow arrive, no action can be taken against this flow. However, this is the desired behavior of our system, since, in this case, such a flow does not continue to contribute to congestion.

CMS may incur overestimation, and the approximation error for a given flow ID depends on the number of hash collisions at buckets it hashed to. Following the analysis presented in [10], if we want to identify all flows that take up at least $\frac{1}{k}$ of the snapshot window, achieving an ϵ-error in flow size estimates with probability $1-\delta$, then we need a CMS with $ln(\frac{1}{\delta})$ rows and $\frac{e}{\epsilon}$ buckets per row. Therefore, for example for $\epsilon = \frac{1}{2k}$ and $k = 10$, using $64 > \frac{e}{\epsilon}$ buckets per row and four rows gives an $\epsilon < 0.1$ and $\delta < 0.02$, which is sufficient for our purposes. Further insight on the selected size of the CMS can be seen in the evaluation in Section 3.

2.5 Round-Robin Rotation of Snapshots

Maintaining infinitely many old snapshots is impractical and unnecessary. With queue capacity C, we need to look into at most $[\frac{C}{w}]$ most recent snapshots. This leads to the second core component of Snappy, using *Round-Robin* on a finite number of snapshots, clearing old snapshots to make space for new ones.

We maintain h snapshots in total, and use them in a Round-Robin fashion, as shown in Figure 3. Every packet entering the queue is added to the "current" snapshot, and the size of its flow is read from several most recent snapshots. We define a snapshot window to be w bytes. The role of these snapshots are rotated after every w bytes that enter the queue. Since $l \leq C$, as long as $h-2 \geq [\frac{C}{w}]$,we have a sufficient number of recent snapshots to read from.

To illustrate the idea further, let us assign h variable indexes to indicate which snapshot to read, write, or clean: I^w is the write index, I^c is the clean index and $I^r_1, \cdots, I^r_{h-2}$ are the read indexes.

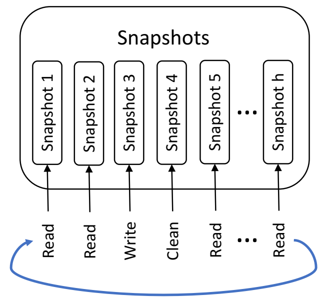

Figure 3: Round-Robin between Snapshots

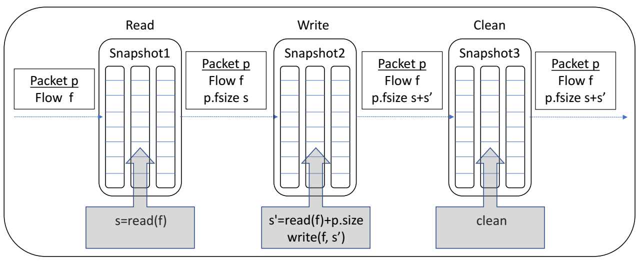

Figure 4: Snapshots in PISA Pipeline

Within a snapshot window, for each packet p of flow f that arrives at the switch, the following is performed:

(1) In snapshot I^w we increment the count of flow f by 1.
(2) To extract the estimated flow size in the queue for f, we first decide to read the $n = [\frac{l}{w}]$ most recent snapshots based on the queue length l when p enters. Subsequently, we sum the estimated flow sizes reported by $I_1^r, \cdots, I_n^r$.
(3) Memory area of snapshot I^c is cleared for future use.

For the structure depicted in Figure 3, without loss of generality we may assume these indexes are initialized to be $I_1^r = 1, \cdots, I_{h-2}^r = h-2$, $I^w = h-1$ and $I^c = h$. Every time w (more) bytes have entered the queue, these indexes are cyclically incremented by 1. For example, after the first w bytes, we cycle indexes to $I_1^r = 2, \cdots, I_{h-2}^r = h-1$, $I^w = h$ and $I^c = 1$. After cycling 4 times, we have $I^w = 3, I^c = 4$, as shown in Figure 3.

As depicted in Figure 4, in a practical implementation on a PISA switch, we maintain snapshots at different stages, implement CMS using stateful memory, and utilize the match-action table to select the appropriate action to read from, write to, or clean the snapshot data structures.

In the data plane, the programmable switch cannot clear out a large chunk of memory at once. Therefore, we use the ongoing traffic to help us clear the oldest snapshot, using each packet to clear one index of memory.

In the illustrated example, we have $h = 3$ snapshots, each snapshot (with its Count-Min Sketch data structure) spans across 3 stages, with CMS using 3 rows and 8 buckets per row. Different snapshots reside in different set of stages across the pipeline.

The first snapshot is currently used for reading. The rules in the match-action table hash the flow ID f to locate counter buckets, then estimate flow size s based on counter values. This estimation is kept as metadata inside the packet.

The second snapshot is in the writing role and accumulates the size of the incoming packet $p.size$. The packet size is added to the appropriate buckets, based on hashing flow ID, and the latest estimated flow size s' is also put in the packet metadata. In this manner, the total estimated size of the flow $p.fsize$ reflects the packets in the latest snapshot window as well.

Finally, the third snapshot is currently being cleaned. Each packet traversing the switch is assigned a single memory index to clear, in a round-robin fashion.

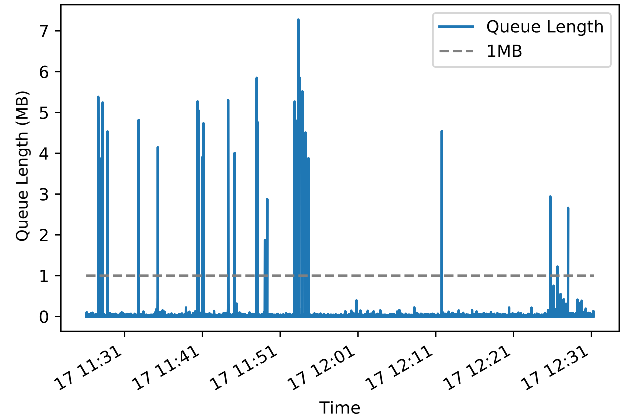

Figure 5: Queue buildup on the UW Trace using throughput 200Mbps.

3 PERFORMANCE EVALUATION

In this section, we evaluated Snappy using realistic data center network trace. We first analyze the trace empirically and show characteristics of microbursts. Subsequently, we show Snappy can achieve high accuracy when identifying culprit flows, using a reasonable amount of hardware resource. Finally, we also demonstrate Snappy can yield good estimate size for both small and large flow, producing an accurate flow size distribution.

3.1 Characterizing Microbursts

We evaluate our solution on the publicly available University of Wisconsin Data Center Measurement trace *UNI1* (UW trace) [5]. We expose the underlying burstiness of its traffic to cause queue buildup, by letting all packets go through a single FIFO queue. In our simulation, packets enter the queue based on their timestamp in the trace file, and depart from the queue with a constant throughput. In this manner, when packets arrive faster than they depart, the queue grows longer; when packets arrive slower, the queue becomes empty. We note that in a real-world scenario each output port has its own queue, our evaluation simulates a single port queue.

Figure 5 shows the queue buildup when running the above simulation on the UW trace, which appears to have a similar bursty pattern as the carrier grade network traffic shown in Figure 1. Using a throughput of 200Mbps the queue builds up to at most ~7MB, albeit having a relatively low average link utilization (26Mbps, 13%) and low average queue utilization (50KB). Most of the time the

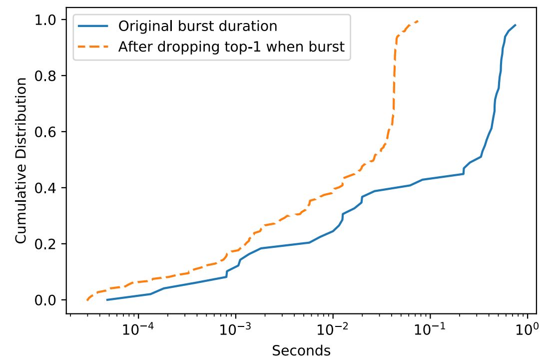

Figure 6: Cumulative distribution of burst duration.

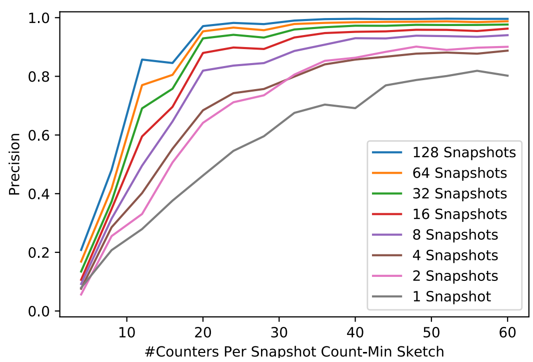

Figure 7: Precision vs. snapshot data structure size.

queue length is relatively short, but on rare occasions when traffic bursts, the queue quickly builds up, then quickly shrinks back down. We varied queue throughput from 200Mbps to 500Mbps and observed similar bursty patterns. Setting throughput close to 1Gbps causes no buildup since the incoming rate never exceeds 1Gbps, while throughput as low as 100Mbps causes the queue to grow excessively long in certain parts of the trace.

For the rest of our evaluation we use throughput 200Mbps and queue capacity $C = 8$MB. Modern shallow-buffer commodity switches typically have a buffer size of several MB. We arbitrarily choose $\alpha C = 1$MB as a congestion threshold, and define a burst be any period that the queue is longer than threshold. Once the threshold is passed, a practical switch should start to react to queue buildup by dropping or marking new packets.

As shown in the lower curve in Figure 6, the duration of these bursts vary greatly, ranging from a fraction of millisecond to almost a second. Figure 6 also shows that if Snappy performs a draconian evasive action to start dropping subsequent packets of the heaviest flow (with the largest estimated flow size) in the queue when the queue length exceeds a threshold, it can effectively reduce the burst duration by an order of magnitude. Although such evasive action is quite primitive, it does illustrate the potential of microburst suppression by targeting at individual bursty flows. We note that different traffic patterns will likely exhibit different flow size distribution during microbursts, and hence may require taking action that is tailored to the different weights of the flows.

Figure 8: Recall vs. snapshot window size.

3.2 Accuracy for Limited Memory and Stages

We evaluate the accuracy of Snappy by testing if it can correctly identify the microburst culprit, using a practical amount of resource in programmable switch. We define a microburst to occur when the queue length is $\geq$ 1MB, at which point Snappy starts to decide which incoming packets belong to culprit flows, defined as the flows occupying at least 1% of the queue length. Snappy is evaluated by the accuracy of its estimated culprit flow set, in terms of Precision and Recall. Precision refers to the number of actual culprit flows identified out of all flows identified by the system. Recall is the number of culprits identified out of all the actual culprits.

In the design space of Snappy there are two primary design choices, the memory size allocated for the snapshot data structure and the snapshot traffic window size. Using more memory to construct a larger Count-Min Sketch (CMS) data structure reduces collision and improve accuracy, but stateful memory is a scarce resource on programmable switches. Using a smaller window provides better granularity when approximating the queue's boundary, at the cost of using more pipeline stages, which is also scarce in hardware.

We first evaluate the memory needed to achieve adequate accuracy. In each snapshot, we use a 4-row CMS to record and estimate the total flow size for each flow during each snapshot window. When memory is insufficient, CMS suffers from hash collisions and over-estimate the size of flows, reporting more false positives and lowering Precision (but Recall doesn't change since CMS produces no false negatives). Figure 7 shows the effect of varying the total number of counters in the CMS on Precision. The Precision plateaus at 24~32 counters (6 to 8 columns per row) with diminishing returns for allocating additional counters. The trace simulation has an average of 56 distinct flows in the queue during microbursts, with an average of 3 heavy flows.

Next, we evaluate the effect of snapshot window granularity on accuracy. We focus on improving Recall in this evaluation, since Figure 7 already demonstrated that the estimation yields high Precision when given enough memory. The multiple curves in Figure 8 overlap, as providing more than enough memory has no impact on Recall. Increasing the number of snapshots (therefore using a shorter window per snapshot) improves Snappy's approximation of

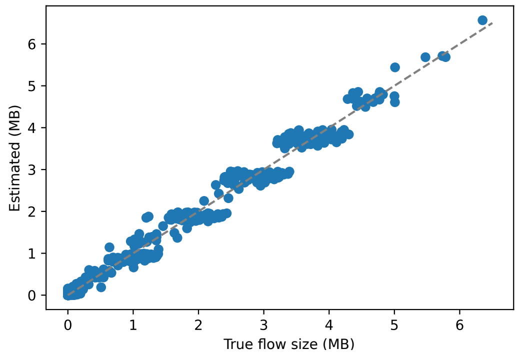

Figure 9: Estimated vs. actual flow sizes in the queue.

the end of the queue. Using fewer snapshots (and a larger window) causes the heavy flows in the offset period near the end of queue (but not actually in the queue) to be erroneously reported, lowering the Recall. In the worst case, Snappy can only look at one snapshot and cannot adapt to changing queue length, therefore reporting only conventional link-level heavy hitters. As shown in Figure 8, by aggregating a maximum of 4 to 8 snapshots each spanning w =1 to 2 MB of traffic, we can achieve a high Recall, and have diminishing return afterwards. As can be seen, adding more memory yields negligible difference in Recall.

3.3 Estimating the Flow Size Distribution

The Count-Min Sketch produces flow weight estimates for all flows, not necessarily the largest ones. Thus, we can use the snapshots to report an in-queue flow size distribution. A network operator may use such a distribution to gain insights on the nature of microburst in a specific switch, and decide on the most appropriate action. For example, if there's usually only one large flow occupying 90% of the queue, then it may be sensible to mark or drop the heaviest flow.

We evaluate the accuracy of this estimation by comparing estimated versus actual size for all flows present in the queue when burst happens. In this evaluation, shown in Figure 9, we use parameters derived from previous experiments to achieve high accuracy using minimal resources: maintaining $h = 8 + 2$ snapshots (read ≤ 8), each accumulating traffic in a window of w =1MB, and each using 32-counter CMS. Since the heavy flows occupy most of the queue, their estimated size are close to integer multiples of snapshot window, causing the "staircase" like graph that is seen. For the smaller flows, a small absolute error is normally achieved. The mean estimation error is 6.2KB while median estimation error is 0.24 KB, implying the estimation is relatively accurate for a majority of flows.

4 RELATED WORK

Existing solutions such as Fastpass [12] offer a centralized traffic orchestration approach for treating queue buildup using scheduling methods. These attempts are too slow for detecting microbursts, as most of the damage is already done by the time high delay or loss can be detected centrally. Other solutions, such as DRILL [11] and CONGA [2], take action to disperse the load within the data plane using load balancing. General solutions such as routing changes or load balancing may disrupt the well-behaved flows, not just the culprits. Instead, solving the problem requires a better understanding of the nature of a microburst as opposed to just detecting it. For instance, finding out that a microburst consists of a single flow or of a certain application opens the opportunity for a targeted remediation such as marking packets, rate limiting or selective dropping. Zhang et al. [15] implemented a high-precision microburst measurement framework in data-center networks and analyzed the duration and inter-arrival time of microbursts. However, the system provides limited insight into the contents of the bursts, such as flow-size distribution and the ID of most significant flows.

An alternative method to prevent bursty flows from affecting other traffic is to use fair queuing. Sharma et al. [13] recently proposed an approximate per-flow fair queuing mechanism using programmable switches. While they present an innovative solution, their method relies on using multiple-FIFO queues per port and quickly rotating their priority. Not all target switches support such functionality and therefore this method may not be deployable in such targets.

We note that our proposed framework continues a series of works which present streaming algorithms for identifying heavy hitters in a sliding window [1, 4, 7]. Previous work mainly focused on a constant-sized window. However, the context of our work deals with a dynamic queue length. This requires detection of heavy flows in varying lengths of the queue history and therefore requires lookup of heavy flows within a variable-length window. Additionally, as far as we know, ours is the first solution provided which has been adapted to the computational constraints posed by programmable switches.

5 CONCLUSION

We present Snappy, a novel way to gain visibility into queue buildups caused by microbursts, based on round-robin snapshots of incoming packets using programmable data plane switches. Evaluation using data-center traces shows that Snappy can achieve good accuracy estimating the heaviest queue occupant flows during microbursts, and can yield a good approximation of flow size distribution in the queue, using a reasonable amount of hardware resources.

We are currently exploring extensions to the model we have discussed, including a multi-queue scenario or non-FIFO queues. Furthermore, we may extend Snappy to identify rapid changes in individual flow throughput, which can help us better understand the dynamics of microbursts. Meanwhile, we are considering how better remediation schemes can be realized using in-queue flow size estimates. We also plan to perform further testing to exhibit how Snappy can bring real world performance improvement.

6 ACKNOWLEDGMENTS

This research is supported by NSF grant CCF-1535948, gifts from AT&T, Intel, and The Eric and Wendy Schmidt Fund for Strategic Innovation. We thank the anonymous reviewers of SelfDN'18 for their helpful comment and suggestions. In addition, we would like to thank Vladimir Braverman, Alan Liu, and Praveen Tammana for their feedback during the writing of this paper.

REFERENCES

[1] Yehuda Afek, Anat Bremler-Barr, Shir Landau Feibish, and Liron Schiff. 2018. Detecting Heavy Flows in the SDN Match and Action Model. *Computer Networks* 136 (2018), 1–12.

[2] 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 *ACM SIGCOMM Conference*. 503–514.

[3] Guido Appenzeller, Isaac Keslassy, and Nick McKeown. 2004. Sizing router buffers. In *ACM SIGCOMM Conference*. 281–292.

[4] Ran Ben-Basat, Gil Einziger, Roy Friedman, and Yaron Kassner. 2016. *Heavy Hitters in Streams and Sliding Windows*. Technical Report CS-2016-01. Computer Science, Technion.

[5] Theophilus Benson, Aditya Akella, and David A. Maltz. 2010. Network traffic characteristics of data centers in the wild. In *ACM SIGCOMM Internet Measurement Conference*. 267–280.

[6] Theophilus Benson, Ashok Anand, Aditya Akella, and Ming Zhang. 2010. Understanding data center traffic characteristics. *ACM SIGCOMM Computer Communication Review* 40, 1 (2010), 92–99.

[7] Vladimir Braverman, Ran Gelles, and Rafail Ostrovsky. 2014. How to catch L_2-heavy-hitters on sliding windows. *Theoretical Computer Science* 554 (2014), 82–94.

[8] The P4 Language Consortium. 2018. $P4_{16}$ Language Specifications. (2018). https://p4.org/p4-spec/docs/P4-16-v1.0.0-spec.pdf

[9] The P4 Language Consortium. 2018. $P4_{16}$ Portable Switch Architecture. (2018). https://p4.org/p4-spec/docs/PSA-v1.0.0.pdf

[10] Graham Cormode and S. Muthukrishnan. 2005. An improved data stream summary: The count-min sketch and its applications. *Journal of Algorithms* 55, 1 (2005), 58–75.

[11] Soudeh Ghorbani, Zibin Yang, Philip Brighten Godfrey, Yashar Ganjali, and Amin Firoozshahian. 2017. DRILL: Micro Load Balancing for Low-latency Data Center Networks. In *ACM SIGCOMM Conference*. 225–238.

[12] Jonathan Perry, Amy Ousterhout, Hari Balakrishnan, Devavrat Shah, and Hans Fugal. 2014. Fastpass: A centralized "zero-queue" datacenter network. In *ACM SIGCOMM Conference*. 307–318.

[13] Naveen Kr. Sharma, Ming Liu, Kishore Atreya, and Arvind Krishnamurthy. 2018. Approximating Fair Queueing on Reconfigurable Switches. In *USENIX Symposium on Networked Systems Design and Implementation*.

[14] Damon Wischik and Nick McKeown. 2005. Part I: Buffer Sizes for Core Routers. *ACM SIGCOMM Computer Communication Review* 35, 3 (July 2005), 75–78.

[15] Qiao Zhang, Vincent Liu, Hongyi Zeng, and Arvind Krishnamurthy. 2017. High-resolution measurement of data center microbursts. In *ACM SIGCOMM Internet Measurement Conference*. ACM, 78–85.

[16] Ying Zhang, Zhuoqing Morley Mao, and Jia Wang. 2007. Low-Rate TCP-Targeted DoS Attack Disrupts Internet Routing. In *Network and Distributed System Security Symposium*.