FlowRadar: A Better NetFlow for Data Centers

Yuliang Li and Rui Miao, ***University of Southern California;*** **Changhoon Kim,** ***Barefoot Networks;*** **Minlan Yu,** ***University of Southern California***

https://www.usenix.org/conference/nsdi16/technical-sessions/presentation/li-yuliang

This paper is included in the Proceedings of the 13th USENIX Symposium on Networked Systems Design and Implementation (NSDI '16).

March 16–18, 2016 • Santa Clara, CA, USA

ISBN 978-1-931971-29-4

Open access to the Proceedings of the 13th USENIX Symposium on Networked Systems Design and Implementation (NSDI '16) is sponsored by USENIX.

FlowRadar: A Better NetFlow for Data Centers

Yuliang Li* Rui Miao* Changhoon Kim† Minlan Yu*
*University of Southern California †Barefoot Networks

Abstract

NetFlow has been a widely used monitoring tool with a variety of applications. NetFlow maintains an active working set of flows in a hash table that supports flow insertion, collision resolution, and flow removing. This is hard to implement in merchant silicon at data center switches, which has limited per-packet processing time. Therefore, many NetFlow implementations and other monitoring solutions have to sample or select a subset of packets to monitor. In this paper, we observe the need to monitor all the flows without sampling in short time scales. Thus, we design FlowRadar, a new way to maintain flows and their counters that scales to a large number of flows with small memory and bandwidth overhead. The key idea of FlowRadar is to encode per-flow counters with a small memory and constant insertion time at switches, and then to leverage the computing power at the remote collector to perform network-wide decoding and analysis of the flow counters. Our evaluation shows that the memory usage of FlowRadar is close to traditional NetFlow with *perfect hashing*. With FlowRadar, operators can get better views into their networks as demonstrated by two new monitoring applications we build on top of FlowRadar.

1 Introduction

NetFlow [4] is a widely used monitoring tool for over 20 years, which records the flows (e.g., source IP, destination IP, source port, destination port, and protocol) and their properties (e.g., packet counters, and the flow starting and finish times). When a flow finishes after the inactive timeout, NetFlow exports the corresponding flow records to a remote collector. NetFlow has been used for a variety of monitoring applications such as accounting network usage, capacity planning, troubleshooting, and attack detection.

Despite its wide applications, the key problem to implement NetFlow in hardware is how to maintain an active working set of flows using a data structure with low time and space complexity. We need to handle collisions during flow insertion and remove old flows to make room for new ones. These tasks are challenging given the limited per-packet processing time at merchant silicon.

To handle this challenge, today's NetFlow is implemented in two ways: (1) Using complex custom silicon that is only available at high-end routers, which is too expensive for data centers; (2) Using software to count sampled packets from hardware, which takes too much CPU resources at switches. Because of the lack of usable NetFlow in data centers, operators have to mirror packets based on sampling or matching rules and analyze these packets in a remote collector [26, 40, 44, 34]. It is impossible to mirror all the packets because it takes too much bandwidth to mirror the traffic, and too many storage and computing resources at the remote collector to analyze every packet. (Section 2)

However, in data centers, there is an increasing need to have visibility of the counters for all the flows all the time. We need to cover all the flows to capture those transient loops, blackholes, and switch faults that only happen to a few flows in the Network and to perform fine-grained traffic analysis (e.g., anomaly detection). We need to cover these flows all the time to identify transient losses, bursts, and attacks in a timely fashion. (Section 3)

In this paper, we propose FlowRadar, which keeps counters for all the flows with low memory overhead and exports the flow counters in short time scales (e.g., 10 ms). The key design of FlowRadar is to identify the best division of labor between cheap switches with limited per-packet processing time and the remote collector with plenty of computing resources. We introduce *encoded flowsets* that only require simple constant-time instructions for each packet and thus are easy to implement with merchant silicon at cheap switches. We then decode these flowsets and perform network-wide analysis across time and switches all at the remote collector. We make

the following key contributions in building FlowRadar:

Capture encoded flow counters with constant time for each packet at switches: We introduce encoded flowsets, which is an array of cells that encode the flows (5 tuples) and their counters. Encoded flowsets ensure constant per-packet processing time by *embracing* rather than handling hash collisions. It maps one flow to many cells, allows flows to collide in one cell, but ensure each cell has constant memory usage. Since encoded flowsets are small, we can afford to periodically export the entire flowsets to the remote collector in short time scales. Our encoded flowset data structure is an extension of Invertible Bloom filter Lookup Table (IBLT), but provides better support for counter updates.

Network-wide decoding and analysis at a remote collector: While each switch independently encodes the flows and counters, we observe that most flows traverse multiple switches. By leveraging the redundancies across switches, we make the encoded flowsets more compact. We then propose a network-wide decoding scheme to decode the flows and counters across switches. With the network-wide decoding, our encoded flowsets can reduce the amount of memory needed to track 100K flows by 5.6% compared to an ideal (and hence impractical) implementation of NetFlow with *perfect hashing* (i.e., no collisions) while providing 99% decoding success rate[1]. (Section 4 and 5)

FlowRadar can support a wide range of monitoring applications including both existing monitoring applications on NetFlow, and new ones that require monitoring all the flows all the time. As demonstrations, we design and build two systems on top of FlowRadar: one that detects transient loops and blackholes using a network-wide flow analysis and another that provides a per-flow loss map using temporal analysis (Section 6).

We discuss the implementation issues in Section 7, compare with related work in Section 8, and conclude in Section 9.

2 Motivation

In this section, we discuss the key challenges of implementing NetFlow. We then describe three alternative monitoring solutions (Table 1): NetFlow in high-end routers with custom silicon, NetFlow in cheap switches with merchant silicon, and selective mirroring. To address the limitations of these approaches, we present FlowRadar architecture, which identifies a good division of labor between the switches and the remote collector.

[1]The decode success rate is defined as the probability of successfully decoding all the flows.

2.1 Key challenges of supporting NetFlow

Since NetFlow has been developed for over 20 years, there have been many implementations and extensions of NetFlow in routers and switches. We cannot capture all the NetFlow solutions here, and in fact many solutions are proprietary information. Instead, we focus on the basic function of NetFlow: storing the flow fields (e.g., 5 tuples) and the records (e.g., packet counter, flow starting time, the time that the flow is last seen, etc.) in a hash table. The key challenge is how to maintain the active working set of flows in the hash table given the limited packet processing time.

Maintain the active working set of flows: There are two key tasks in maintaining the active working set of flows:

(1) How to handle hash collisions during flow insertion? When we insert a new flow, it may experience collisions with existing flows. One solution is to store multiple flows in each cell in the hash table to reduce the chances of overflow (e.g., d-left hashing [14, 38]), which requires atomic many-byte memory accesses. Another solution to move existing flows around to make room for new flows (e.g., Cuckoo hashing [33]), which requires multiple, non-constant memory accesses per packet in the worst case. Both are very challenging to implement on merchant silicon with high line rate. The detailed challenges are discussed in Section 8.

(2) How to remove an old flow? We need to periodically remove old flows to make room for new flows in the hash table. If a TCP flow receives a FIN, we can remove it from the table. However, in data centers there are many persistent connections reused by multiple requests/responses or messages. To identify idle flows, NetFlow keeps the time a flow is last seen and periodically scan the entire hash table to check the inactive time of each flow. If a flow is inactive for more than the inactive timeout, NetFlow removes the flow and exports its counters. The inactive timeout can only be set between 10 and 600 seconds with a default value of 15 seconds [1]. When the hash table is large, it takes a significant time and switch CPU resources to scan the table and clean up the table entries.

Limited per-packet processing time at merchant silicon: It is hard to maintain the active working set of flows at the merchant silicon—the commodity switch design in data centers. The key constraint of the merchant silicon is the limited time we can spend on each packet. Suppose a switch has 40Gbps per port, which means 12ns per packet processing time for 64 Byte packets[2]. Let's assume the entire 12 ns can be dedicated to NetFlow by performing perfect packet pipelining and

[2]This becomes worse when datacenters move to 100Gbps.

	Hardware-based NetFlow in custom silicon	Sampled software-based NetFlow in merchant silicon	sFlow [40], EverFlow [44]	FlowRadar
Division of labor				
state in switch hardware	active working set of flows	none	none	encoded flows and counters
state in switch software	none (or some active flows)	active working set of flows	none	none
data exported to collector	flow records after termination	flow records after termination	Selected pkts and timestamps	periodic encoded flow records
Coverage of traffic info				
Temporal coverage	No	No	No (if select control packets)	**Yes (milliseconds)**
Flow coverage	**All** or sampled packets	sampled packets	sampled or selected packets	**All**

Table 1: Comparing FlowRadar with hardware-based NetFlow in custom silicon, sampling-based software NetFlow in merchant silicon, and sFlow/EverFlow

allocating all other packet processing functions (packet header parsing, Layer 2/3 forwarding, ACLs, etc.) to other stages. Yet inside NetFlow, one needs to calculate the hash functions, look up SRAM, run a few ALU operations, and write back to the SRAM. Even with on-chip SRAM which has roughly 1 ns access time, to finish all these actions in 12 ns is still a challenge. (Similar arguments are made in [23] about the difficulties of implementing data streaming at routers.)

2.2 Alternative monitoring solutions

Due to the limited per-packet time in merchant silicon, one cannot process complex and non-constant time insertion and deletion actions as required in NetFlow. Therefore, there are three alternatives (Table 1):

Hardware-based NetFlow in custom silicon: One solution is to design custom silicon to maintain the active working set of flows in switch hardware. We can cache popular flow entries in on-chip SRAM, but the rest in off-chip SRAM or DRAM. We can also combine SRAM with expensive and power-hungry TCAM to support parallel lookup. Even with the expensive custom silicon, the test of Cisco high-end routers (Catalyst series) [18, 12] shows that there is still around 16% switch CPU overhead for storing 65K flow entries in hardware. Cisco highly recommends NetFlow users to choose sampling to reduce the NetFlow overhead on these routers [18].

Sampled software-based NetFlow in merchant silicon: Another solution is to sample packets and mirror them to the switch software, and maintain the active working set of flows in software. This solution works with cheap merchant silicon, but takes even more CPU overhead than hardware-based NetFlow in high-end routers. To reduce the switch CPU overhead of NetFlow and avoid interrupting other processes (e.g., OSPF, rule updates) in CPU, operators have to set sampling rate low enough (e.g., down to 1 in 4K). With such low sampling rate, operators cannot use NetFlow for fine-grained traffic analysis (e.g., anomaly detection) or capturing those events that only happen to some flows (e.g., transient loops or blackholes).

Selective mirroring (sFlow [40], EverFlow [44]): The final solution data center operators take today is to only sample packets or select packets based on match-action rules, and then mirror these packets to a remote collector. The remote collector extracts per flow information and performs detailed analysis. This solution works with existing merchant silicon, and best leverages the computing resources in the cloud. However, it takes too much bandwidth overhead to transfer all the packets to the collector and too much storage and computing overhead at the collector [44]. Therefore, operators can only get a partial view from the selected packets.

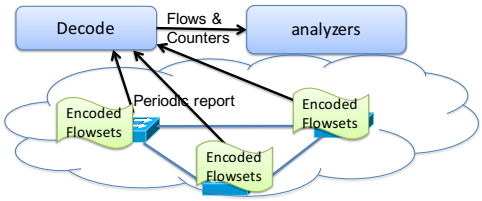

Figure 1: FlowRadar architecture

2.3 FlowRadar architecture

Instead of falling back to sampling in existing monitoring solutions, we aim at providing full visibility to all the flows all the time (see example use cases in Section 3). To achieve this, we propose to best leverage the capabilities at both the merchant silicon at switches and the computing power at the remote collector (Figure 1).

Capturing encoded flow counters at switches: FlowRadar chooses to encode flows and their counters into small fixed memory size that can be implemented in merchant silicon with constant flow insertion time. In this way, we can afford to capture all the flows without sampling, and periodically export these encoded flow counters to the remote collector in short time scales.

Decoding and analyzing flow counters at a remote collector: Given the encoded flows and counters exported from many switches, we can leverage the computing power at the remote collector to perform network-wide decoding of the flows, and temporal and flow space analysis for different monitoring applications.

3 Use cases

Since FlowRadar provides per flow counters, it can easily inherit many monitoring applications built on NetFlow such as accounting, capacity planning, application monitoring and profiling, and security analysis. In this section, we show that FlowRadar provides better monitoring support than sampled NetFlow and sFlow/EverFlow in two aspects: (1) Flow coverage: count all the flows without sampling; and (2) Temporal coverage: export these counters for each short time slot (e.g., 10 ms).

3.1 Flow coverage

Transient loop/blackhole detection: Transient loops and blackholes are important to detect, as they could cause packet loss. Just a few packet losses can cause significant tail-latency increase and throughput drops (especially because TCP treats losses as congestion signals) [31, 10], leading to violations of service level agreements (SLAs) and even a decrease of revenue [19, 39]. However, transient loops and blackholes are difficult to detect, as they may only affect a few packets during a very short time period. EverFlow or sampled NetFlow only select a few packets to monitor, and thus may miss most of the transient loops and blackholes. In addition, the transient loops and blackholes may only affect a certain kind of flows, so probing methods like Pingmesh [25] may not even notice the existence of them. Instead, if we can capture all the packets in each flow and maintain a corresponding counter in real time at every switch, we can quickly identify flows that are experiencing loops or blackholes (see Section 6).

Errors in match-action tables: Switches usually maintain a pipeline of match-action tables for packet processing. Data centers have reported table corruptions when switch memory experiences soft errors (i.e., bit flips) and these corruptions can lead to packet losses or incorrect forwarding for a small portion of the traffic [25, 44][3]. Such corruptions are hard to detect using network verification tools because they cannot see the actual corrupted tables. They are also hard to detect by sampled NetFlow or EverFlow because we cannot pre-decide the right set of packets to monitor. Instead, since FlowRadar can monitor all the packets, we can see problems when they happen (Section 6).

Fine-grained traffic analysis: Previous research has shown that packet sampling is inadequate for many fine-grained monitoring tasks such as understanding flow size distribution and anomaly detection [22, 20, 30]. Since FlowRadar monitors all the packets, we can provide more accurate traffic analysis and anomaly detection.

[3]For example, the L2 forwarding table gets corrupted. The packet that matches the entry can be flooded or mis-forwarded, leading to transient blackholes or loops before the entry is relearnt and corrected.

3.2 Temporal coverage

Per-flow loss map: Packet losses can be caused by a variety of reasons (e.g., congestion, switch interface bug, packet corruptions) and may have significant impact on applications. Although each TCP connection can detect its own losses (with sequence numbers or with switch support [17]), it is hard for the operators to understand where the losses happen inside the network, how many flows/applications are affected by such loss, and how the number of losses changes over time. NetFlow with low sampling rates cannot capture losses that happened to flows that are not sampled; and even for those sampled flows, we cannot infer losses from estimated flow counters. EverFlow can only capture control packets (e.g., NACK (Negative Acknowledgment)) to infer loss and congestion scenarios. Instead, if we can deploy FlowRadar at switches, we can directly get an overall map of the per-flow loss rate for all the flows soon after a burst of packets passes by (see Section 6).

Debugging ECMP load imbalance: ECMP load imbalance can lead to inefficient bandwidth usage in network and can significantly hurt application performance [11]. Short-term load imbalance can be caused by either (1) the network (e.g., ECMP not hashing on the right flow fields) or (2) the application (e.g., the application sends a sudden burst). If operators can quickly distinguish the two cases, they can make quick reactions to either reconfigure the ECMP functions for the network problem or to rate limit a specific application for the application problem.

EverFlow can diagnose some load imbalance problems by mirroring all the SYN and FIN packets and count the number of flows on each ECMP paths. However, it cannot diagnose either of the two cases above because it does not have detailed packet counters for each flow and does not know the traffic changes for these flows over time. Traditional NetFlow has similar limitations (i.e., no track of flows over time).

Timely attack detection: Some attacks exhibit specific temporal traffic patterns, which are hard to detect if we just count the number of packets per flow as NetFlow, or just capture the SYN/FIN packets as EverFlow. For example, TCP low-rate attacks [29] send a series of small traffic bursts that always trigger TCPs retransmission timeout, which can throttle TCP flows to a small fraction of the ideal rate. With per-flow counters at small time scale, we can not only detect these attacks by temporal analysis, but also report these attacks quickly (without waiting for the inactive timeout in NetFlow).

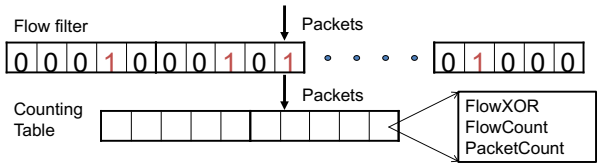

Figure 2: IBLT based flow counters

4 FlowRadar Design

The key design in FlowRadar is an encoding scheme to store flows and their counters in a small fixed-size memory, that requires constant insertion time at switches and can be decoded fast at the remote collector. When there is a sudden burst of flows, we can leverage network-wide decoding to decode more flows from multiple encoded flowsets. We also analyze the tradeoff between memory usage and decoding success rates.

4.1 Encoded Flowsets

The key challenge for NetFlow is how to handle flow collisions. Rather than designing solutions to *react* to flow collisions, our design focuses on how to *embrace* collisions: We allow flows to collide with each other without extra memory usage, and yet ensure we can decode individual flows and their counters at the collector.

There are two key designs that allow us to embrace collisions: (1) First, we hash the same flow to multiple locations (like Bloom filters). In this way, the chance that one flow collide with other flows in one of the bins decreases. (2) When multiple flows fall in the same cell, it is expensive to store them in a linked list. Instead, we use a XOR function to the packets of these flows without using extra bits. In this way, FlowRadar can work with a fixed-size memory space shared among many flows and has constant update and insertion time for all the flows.

Based on the two designs, the *encoded flowset* data structure is shown in Figure 2, which includes two parts: The first part is the *flow filter*. The flow filter is just a normal Bloom filter with an array of 0's and 1's, which is used for testing if a packet belongs to a new flow or not. The second part is the *counting table* which is used to store flow counters. The counting table includes the following fields:

- *FlowXOR:* which keeps the XOR of all the flows (defined based on 5 tuples) mapped in the bin
- *FlowCount:* which keeps the number of flows mapped in the bin
- *PacketCount:* which keeps the number of packets of all the flows mapped in the bin

As indicated in Algorithm 1, when a packet arrives, we first extract the flow fields of the packet, and check the flow filter to see if the flow has been stored in the flowset or not. If the packet comes from a new flow, we update the counting table by adding the packet's flow fields to FlowXOR and incrementing FlowCount and PacketCount at all the k_c locations. If the packet comes from an existing flow, we simply increment the packet counters at all the k_c locations.

Algorithm 1: FlowRadar packet processing

```
1  if ∃ i ∈ [1,k_f], s.t. FlowFilter[H_i^F(p.flow)]==0 then
2      FlowFilter.add(p.flow);
3      for j= 1..k_c do
4          l = H_j^C(p.flow);
5          CountTable[l].FlowXOR =
           CountTable[l].FlowXOR ⊕ p.flow;
6          CountTable[l].FlowCount ++;
7      end
8  end
9  for j= 1..k_c do
10     CountTable[H_j^C(p.flow)].PacketCount ++;
11 end
```

Each switch sends the flowset to the collector every a few milliseconds, which we defined as *time slots*. In the rest of the paper, we set the value of the time slot to 10ms, unless explicitly setting it to other values in the context.

When FlowRadar collector receives the encoded flowset, it can decode the per flow counters by first looking for cells that include just one flow in it (called *pure cell*). For each flow in a *pure cell*, we perform the same hash functions to locate the other cells of this flow and remove it from all the cells (by XORing with the FlowXOR fields, subtracting the packet counter, and decrementing the flow counter). We then look for other *pure cells* and perform the same for the flows in each *pure cell*. The process ends when there are no *pure cells*. The detailed procedure is illustrated in Algorithm 3 in the appendix.

4.2 Network-wide decoding

Operators can configure the encoded flowset size based on the expected number of flows. However, there can be a sudden burst in terms of the number of flows. In that case, we may fail to decode some flows, when we do not have any cell with just one flow in the middle of the SingleDecode process. To handle a burst of flows, we propose a network-wide decoding scheme that can correlate multiple encoded flowsets at different switches to decode more flows. Our network-wide decoding process has two steps: decoding flows across switches and decoding counters inside a single switch.

FlowDecode across switches: The key observation is that if we use different hash functions at different switches, and if we cannot decode one flow in one encoded flowset, it is likely that we may be able to de-

code the flow at another encoded flowset at a different switch the flow traverses. For example, suppose we collect flowsets at two neighboring switches A_1 and A_2. We know that they have a common subset of flows from A_1 to A_2. Some of these flows may be single-decoded at A_1 but not A_2. If they match A_2's flow filter, we can remove these flows from A_2, which may lead to more one-flow cells. We can run SingleDecode on A_2 again.

Algorithm 2: FlowDecode

```
1  for i=1..N do
2  |  S_i = SingleDecode(A_i);
3  end
4  finish = false;
5  while not finish do
6  |  finish = true;
7  |  foreach A_i,A_j are neighbor do
8  |  |  foreach flow in S_i − S_j do
9  |  |  |  if A_j.FlowFilter.contains(flow) then
10 |  |  |  |  S_j.add(flow);
11 |  |  |  |  for p=1..k_c do
12 |  |  |  |  |  l = H_p^{j,C}(flow);
13 |  |  |  |  |  A_j.CountTable[l].FlowXOR =
   |  |  |  |  |  A_j.CountTable[l].FlowXOR ⊕ flow;
14 |  |  |  |  |  A_j.CountTable[l].FlowCount -= 1;
15 |  |  |  |  end
16 |  |  |  end
17 |  |  end
18 |  |  foreach flow in S_j − S_i do
19 |  |  |  Update S_i and A_i same as S_j and A_j
20 |  |  end
21 |  end
22 |  for i=1..N do
23 |  |  result = SingleDecode(A_i);
24 |  |  if result ≠ ∅ then
25 |  |  |  finish = false;
26 |  |  end
27 |  |  S_i.add(result);
28 |  end
29 end
```

The general process of FlowDecode is described in Algorithm 2. Suppose we have the N encoded flowsets: $A_1..A_N$, and the corresponding sets of flows we get from SingleDecode $S_1..S_N$. For any two neighboring A_i and A_j, we check the all the flows we can decode from A_i but not A_j (i.e., $S_i - S_j$) to see if they also appear at A_j's flow filter. We remove those flows that match A_j's flow filter from A_j. We then run SingleDecode for all the flowsets again, get the new groups of $S_1..S_N$ and continue checking the neighboring pairs. We repeat the whole process until we cannot decode any more flows in the network.

Note that if we have the routing information of each packet, FlowDecode can speed up, because for one decoded flow at A_i, we only check the previous hop and next hop of A_i instead of all neighbors.

CounterDecode at a single switch: Although we can easily decode the flows using FlowDecode, we cannot decode the counters of them. This is because the counters at A and B for the same flow may not be the same due to the packet losses and on-the-fly packets (e.g. packets in A's output queue). Fortunately, from the FlowDecode process, we may already know all the flows in one encoded flowset. That is, at each cell, we know all the flows that are in the cell and the summary of these flows' counters. Formally, we know $CountTable[i].PacketCount = \sum_{\forall f,\exists j,H_j^C(f)=i} f.PacketCount$ for each cell i. Suppose the flowset has m_c cells and n flows, we have a total of m_c equations and n variables. This means we need to solve $MX = b$, where X is the vector of n variables and M and b are constructed from the above equations. We show how to construct M and b in Algorithm 4 in the Appendix.

Solving a large set of sparse linear equations is not easy. With the fastest solver lsqr (which is based on iteration) in Matlab, it takes more than 1 minute to get the counters for 100K flows. We speed up the computation from two aspects. First, we provide a close approximation of the counters, so that the solver can start from the approximation and reach the result fast. As the counters are very close across hops for the same flow, we can get the approximated counters during the FlowDecode. That is, when decoding A_i with the help of A_j's flows (Algorithm 2 line 7 to 21), we treat the counter from A_j as the counter in A_i for the same flow. We feed the approximated counters to the solver as initial values to start iteration, so that it can converge faster. Second, we use a loose stopping criterion for the iteration. As the counter is always an integer, we stop the iteration as long as the result is floating within a range of ± 0.5 around an integer. This significantly reduces the rounds of iteration. By these two optimizations, we reduce the computation time by around 70 times.

4.3 Analysis of decoding errors

SingleDecode: We now perform a formal analysis of the error rate in an encoded flowset. Suppose the flow filter uses k_f hash functions and m_f cells; and the counting table has k_c hash functions and m_c cells with s_c bits per cell. The total memory usage is $m_c \cdot s_c + m_f$. Assume there are n flows in the encoded flowset. For the flow filter, the false positive for a single new flow (i.e., the new flow being treated as an existing flow) is $(1 - e^{-k_f n/m_f})^{k_f}$. Thus the chance that none of the n flows experience false positives is $\prod_{i=1}^{n-1}(1 - (1 - e^{-k_f i/m_f})^{k_f})$. When the flow filter has a false positive, we can detect it by checking if there are non-zero PacketCounts after decoding. In this case the counters are not trustful, but we still get all the flows.

For the counting table, the decoding success rate of SingleDecode (i.e., the chance we can decode *all* the flows) is proved to be larger than $O(1 - n^{-k_c+2})$, if $m_c > c_{k_c} n$, where c_{k_c} is a constant associates with k_c [24]. When we fail to decode some flows in the counting table, the already decoded flows and their counters are correct.

We choose to use separated flow filter and counting table rather than a combined one (i.e. the counting table also serves as a bloom filter to test new flow), because a combined one consumes much more memory. For a combined one, for each packet, we check the k_c cells it is hashed to, and view this flow as a new flow if and only if at least one of these k_c cells' FlowCount is 0. However, this solution requires far more memory than the separated solution. This is because for the counting table, a good parameter setting is about k_c = 3 and $m_c = 1.24n$ when n is larger than 10K based on the guidelines in [24] and our experiences in Section 5. In such a parameter setting, when we treat the counting table as a Bloom filter, the false positive rate for a new flow is $(1 - e^{-k_c n/m_c})^{k_c}$ is larger than 99.9%. To keep the false positive rate low enough for all the n flows, we would have to significantly increase k_c and m_c.

NetDecode: We discuss FlowDecode and CounterDecode separately. For FlowDecode, we first consider a simple *pair-decode* case, where we run NetDecode between two nodes with the same set of flows. This can be viewed as decoding n flows in a large counting table with $2k_c$ hashes and $2m_c$ cells. This means we will need only half of the number of cells of the counting table with $2k_c$ hashes with SingleDecode. In our experiment, we only need $m_c = 8K$ for decoding 10K flow appear at both sides, which is even fewer than the number of flows.

For the more general network-wide FlowDecode, if all nodes in the network have more flows than expected and require FlowDecode, the decode success rate is similar to the *pair-decode* case. This is because for each node A, decoding its flows is similar to decoding the pair of A's flowset and the sum of flowsets from all the neighbors containing A's flows. However, it is more likely that only a portion of the nodes have more flows than expected, and the rest can SingleDecode. In this case, the decode success rate is higher than the *pair-decode* case.

For CounterDecode, we need at least the same number of linear equations as the number of variables (per flow counters). Because we have one equation per cell, we need the number of cells m_c to be at least the number of variables n. In practice, m_c should be slightly larger than the n, to obtain a high enough chance of having n linearly independent equations.

The complete NetDecode process is bottlenecked by CounterDecode not FlowDecode. This is because CounterDecode requires more memory and takes more time to decode. Since CounterDecode only runs on a single node, the memory usage and decoding speed of NetDecode at a node mostly depends on the number of flows in its own decoded flowset, rather than the number of other flowsets that contain similar flows.

5 Evaluation

In this section, we demonstrate that FlowRadar can scale to many flows and large networks with limited memory, bandwidth, and computing overhead, through simulations on FatTree topologies.

5.1 Scale to many flows

Parameter settings We set up a simulation network of FatTree with $k = 8$ (80 switches). We set the number of flows on each switch in 10 ms from 1K to 1000K. We generate an equal number of flows between each inter-Pod ToR pair. We then equally split these flows among ECMP paths. In this way, each switch has the same number of flows. We set the flow filter to ensure that the probability that one of the n flows experiences a false positive is 1/10 of the SingleDecode failure rate of the counting table. We set the optimal k_f and m_f according to the formulas in Section 4.3. We set $k_c = 4$ because it is the best for NetDecode. We select m_c based on the guidelines in [24]. We set the size of FlowCounter according to the expected number of flows. We conservatively set both NetFlow and FlowRadar packet counters as 4 Bytes, although in FlowRadar we collect statistics in a short time scale and thus would see much fewer packets and needs fewer bytes for the packet counter. Since our results are only related to the number of flows but not the packets, we generate a random set of flows as input.

We run decoding on 3.60GHz CPU cores, and parallelize decoding different flowsets on multiple cores.

The memory usage of FlowRadar is close to NetFlow with a perfect hash table: We first compare the memory usage between NetFlow and FlowRadar. As discussed in Section 2, it is almost impossible in merchant silicon to implement a hash-based design that handles flow insertions and collisions within the per packet time budget. If we implement a simple hash table, it would take 8.5TB to store 100K flows to ensure a 99% chance that there are no collisions. The actual data structure used in custom silicon would be proprietary information. Therefore, we compare with the best possible case for NetFlow—a perfect hash table without any collisions.

Even with a perfect hash table, NetFlow still needs to store in each cell the starting time of a flow and the time the flow is last seen for calculating inactive timeout (4 Bytes each). However, in FlowRadar, we do not need to keep timestamps in hardware because we use frequent

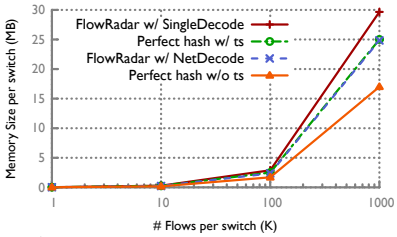

Figure 3: Memory usage per switch

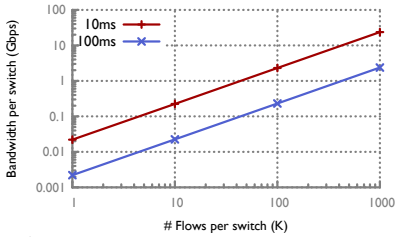

Figure 4: Bandwidth usage per switch

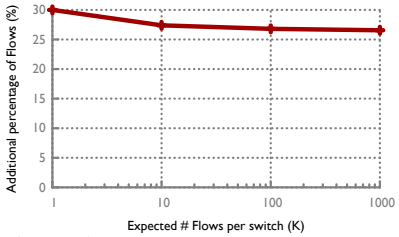

Figure 5: Extra #flows using NetDecode

reporting in a short scale. To fully decouple the benefit of FlowRadar data structure and removing timestamps, we also compare with perfect hashing without timestamps, which can be viewed as the optimal case we can reach.

Figure 3 shows that NetFlow with perfect hashing needs 2.5 MB per switch. FlowRadar needs only 2.88MB per switch with SingleDecode and 2.36MB per switch with NetDecode to store 100K flows with 99% decoding success[4], which is +15.2% and -5.6% compared to 2.5MB used by NetFlow. The best possible memory usage with perfect hashing without timestamps is 1.7MB per switch. With 1M flows, we need 29.7MB per switch for SingleDecode and 24.8MB per switch for NetDecode, which is +18.8% and -0.8% compared to NetFlow with perfect hashing and timestamps.

FlowRadar requires only a small portion of bandwidth to send encoded flowsets every 10ms. Figure 4 shows that we only need 2.3Gbps per switch to send encoded flowsets of 100K flows with 10ms time slot, and 0.23Gbps with 100ms time slot. In Facebook data center and traffic setting [35], a rack switch connects to 44 hosts with 10Gbps links, where each host send at most 100s to 1000s of concurrent flows in 5ms. Suppose there are a total of 2K*44 flows in 10ms in the rack switch, FlowRadar only incurs less than 0.52% of bandwidth overhead (2.3Gbps/(44*10Gbps)) with 10ms time slot.

FlowRadar with NetDecode can support 26.6-30% more flows than SingleDecode, with more decoding time Operators can configure FlowRadar based on the expected number of flows. When the number of flows goes beyond the expected number, we can use NetDecode to decode more flows given the same memory. Figure 5 shows with 1K to 1M expected number of flows, NetDecode can decode 26.6-30% more flows than SingleDecode given the same memory. So our solution can tolerate bursts in the number of flows.

Figure 6 shows the average decoding time of each flowset for the case with 100K expected flows. When the traffic is below 100K flows, the collector can run SingleDecode to quickly detect all the flows within 10 ms. When the traffic goes beyong 100K flows, we need NetDecode, which takes 283ms and 3275ms to decode flowsets with respective 101K flows and 126.8K flows.

We break down the NetDecode time into CounterDecode and FlowDecode. The result is shown in Figure 7. As the number of flows increases, the CounterDecode time increases fast, but the FlowDecode time remains low. If we just need to decode the flows, we need only 135ms, which is very small portion compared to CounterDecode's 3140ms. Note that the burst of flows does not always happen, so it is fine to wait for extra time to get the decoded flows and counters.

We do not rely on the routing information to reduce the NetDecode time, because it only helps reduce the FlowDecode time, which is only a small portion of the NetDecode time. The routing information can help reduce the FlowDecode time by 2 times.

5.2 Scale to many switches

We now investigate how FlowRadar scales with larger networks. For direct comparison, we assume the same number of flows per switch with different network sizes.

The memory and bandwidth usages per switch do not change with more switches: This is because the decoding success rate only relates to the number of flows and number of cells. Obviously this is true for SingleDecode. For NetDecode this is also true, because as long as all flows appear in at least 2 flowsets, NetDecode's decoding rate is similar no matter how many flowsets the flows appear in. The reason is that the bottleneck of the number of flows can be decoded is from CounterDecode, which is independent from other flowsets. For flowsets with 102.5K cells, two such flowsets can already decode more than 110K flows, but the CounterDecode can only support 100K flows (limited by the number of linearly independent equations).

Decoding requires proportionally more cores with more switches: The SingleDecode time per switch only relates to the number of flows in a flowset. For example, to decode 100K flows within 10ms, we need the same number of cores at the remote collector as the number of switches. This means for a network with 27K servers (K=48 FatTree) and 16 cores per server, we need about

[4]Note that even in the 1% of cases we cannot successfully decode all flows, we can still decode 61.7% of the flows on average.

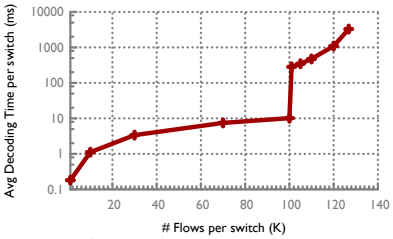

Figure 6: Decoding time

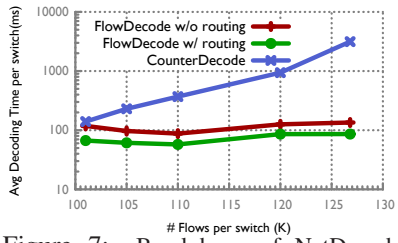

Figure 7: Breakdown of NetDecode Time

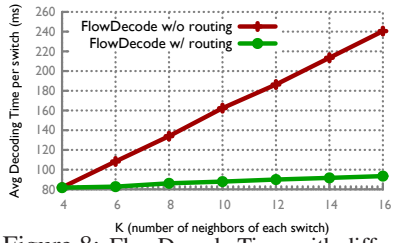

Figure 8: FlowDecode Time with different network size

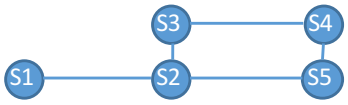

Figure 9: A flow path that has cycle

0.65% of the servers for the decoding.

NetDecode only happens during bursts of flows. The decoding time per switch increases slowly with more switches, because most time is spent on CounterDecode, which only relates to the number of flows in a flowset.

The FlowDecode time increases with larger networks, because it takes more time to check a decoded flow with the neighboring switches, when there are more neighbors in a larger network. In a FatTree network, suppose each switch has k neighbors. The total number of switches in the network is $n = \frac{5}{4}k^2$, so each flowset only checks with $O(\sqrt{n})$ other flowsets. We tested the FlowDecode time with different FatTree network sizes by increasing k from 4 to 16. The memory on each switch is set expecting 100K flows for SingleDecode. We generate traffic such that the number of flows on each switch reaches the maximum number (126.8K) that could be NetDecoded. Figure 8 shows the result. The FlowDecode time increases linearly with k. However, it is still a small portion compared to CounterDecode time. For 126.8K flows per switch and $k = 16$ FatTree, FlowDecode only takes 0.24 seconds, which is 7.1% of the total decoding time. Routing information can speed up FlowDecode to 0.093 seconds, which is 2.9% of the total decoding time.

6 FlowRadar Analyzer

We show two use cases of FlowRadar: transient loop and blackhole detection with network-wide flow analysis and providing per-flow loss map with temporal analysis.

6.1 Transient loop/blackhole detection

With FlowRadar, we can infer the path for each flow by concatenating the switches that have records for that flow. As a result, we can easily provide a network-wide map of all the loops and blackholes, the time they happen, and the flows they affected.

Loops: We first identify all the switches that see the same flow during each time slot. If the switches form a cycle, then we suspect there is a loop. We cannot conclude that there is a loop because this may be caused by a routing change. For example, in Figure 9, we may observe counters at all the switches in one time slot with FlowRadar, which forms a cycle (S2,S3,S4,S5). However, this may be caused by a routing change from S1 $\rightarrow$ S2 $\rightarrow$ S5 to S1 $\rightarrow$ S2 $\rightarrow$ S3 $\rightarrow$ S4 $\rightarrow$ s5 within the time slot. To confirm, we need to compare the counter on the hop that is not in the cycle (counter1), and the counter on one hop in the cycle (counter2). If counter1 $<$ counter2 then we can conclude that there is a loop. For example, if counter on S1 $<$ counter on S3, we know this is a loop.

Blackholes: If a transient blackhole is longer than a slot's time, we can detect it by seeing the path of some flows stopped at some hop. If a transient blackhole is shorter than a slot's time, we still see a large difference between the counters before and after the blackhole at one slot. Note that we do not need the counters, but only the flow information to detect blackhole. Thus, during flow bursts, we can run FlowDecode without CounterDecode to detect blackholes faster.

Evaluation: We create a FatTree k=4 topology with 16 hosts and 20 switches in DeterLab [2]. We modify Open vSwitch [6] to support our traffic collection. We direct all the packets to the user space and maintain the encoded flowsets. We install forwarding rules for individual flows with different source and destination IP pair. We send persistent flows from each host to all the other hosts, which send one packet every 5 ms. This is to make sure that each flow has at least one packet in each time slot even if some packets is close to the slot's boundary.

We simulated a case that a VM migration causes a transient loop when the routing table on the edge switch S1 of the old VM location is updated so it sends packets up to the aggregation switch S2. But S2 has not been updated so it sends packets back to S1. We manually updated a rule at the edge switch S1 at around 10ms, which forms a loop S1 $\rightarrow$ S2 $\rightarrow$ S1, where S2 is an aggregation switch. We can detect the loop within 10ms.

To generate a blackhole, we manually remove a routing rule at an edge switch. We can detect the blackhole

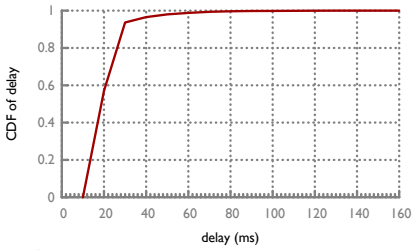

Figure 10: CDF of loss detection delay

within 20 ms. This is because there are still traffic in the first 10ms when the blackhole happens. So we can only confirm in the next 10ms.

6.2 Per-flow loss map

FlowRadar can generate a network-wide loss map by comparing the per-flow counters between the upstream and downstream switches (or hosts) in a sequence of time slots. A simple approach is that for each flow, the difference between the upstream and downstream counters is the number of losses in each time slot. However, this approach does not work in practice because it is impossible for the two switches capture exactly the same set of packets, even though today's data centers often have well synchronized clocks across switches at milliseconds level. This is because there are always packets on the fly between upstream and downstream switches (e.g., in the switch output queue).

To address this problem, we can wait until the flow finishes to compare its total number of packets at different hops. But this takes too long. Instead, we can detect losses faster by comparing counters for flowlets instead of flows. Suppose a time slot in FlowRadar is 10ms. We define *flowlets* as bursts of packets from a flow that are separated by gaps larger than a time slot [27]. With FlowRadar, we can identify flowlets between two time slots with counters equal to zero. Given a flowlet f, the upstream and downstream switches collect sequences of counters: $U_1...U_t$ and $D_1...D_t$ (D_0 and D_{t+1} are zero). We compute the total number of losses for the flowlet f as $\sum_{i=1}^{t}(U_i) - \sum_{i=1}^{t}(D_i)$. This is because if a packet does not arrive at the downstream switch for at least 10ms, it is very likely this packet is lost.

With this approach, we can get the accurate loss numbers and rates for all the flowlets that have finished. The key factor for our detection delay is the duration of flowlets. Fortunately, in data centers, many flows have short flowlets. For example, in a production web search workload [13], 87.5% of the partition/aggregate query flows are separated by a gap larger than 15 ms. 95% of query flows can finish within 10ms. Moreover, 95% of background large flows have 10-200 ms flow completion times with potential flowlets in them.

Evaluation: We evaluate our solution in a k=8 FatTree topology in a ns-3 simulator [5]. The FatTree has 128 hosts connected with 80 switches using 10G links. We take the same workload distribution from a production web search data center [13], but add the 1000 partition-aggregate queries per second with 20 incast degree (i.e., the number of responding nodes) and packet sizes of 1.5KB. The queue size of each port in our experiment is 150KB which means 100 packets of size 1.5KB. The flowlet durations are mostly shorter than 30ms with the maximum as 160ms. 50% of background traffic has 0ms interarrival time indicates application sends a spike of flows. The rest at least 40% of background traffic has interarrival time larger than 10ms for periodical update and short messages.

We run FlowRadar to collect encoded flowsets every 10ms at all the switches. We define detection delay as the time difference between when the loss happens and when we report it. Figure 10 shows the CDF of loss detection delay. We can detect more than 57% of the losses within 20ms, and more than 99% of the losses within 50ms.

7 Implementation

We now discuss the implementation issues in FlowRadar.

Encode and export flow counters at switches: FlowRadar only requires simple operations (e.g., hashing, XOR, and counting) that can be built on existing merchant silicon components. For example, hashing is already used in Layer 2 forwarding and ECMP functions. With the trend of programmable switches (e.g., P4 [8]), FlowRadar can be easier to implement.

We have implemented our prototype in P4 simulator [9], which will be released at [3]. We use an array of counters to store our counting table and flow filter. On each packet's arrival, we use the modify_field_with_hash_based_offset API to generate the k_c hash values for counting table and k_f hash values for flow filter, and use bit_xor API to xor the header into the flowXOR field. In the control plane, we use the stateful_read_counter API to read the content in our data.

Since the encoded flowset is small, we can export the entire encoded flowset to the collector rather than exporting them on a per flow basis. To avoid the interruptions on the data plane during the exporting phase, we can use two encoded flowset tables: the incoming packets update one table while we export data in another table. Note that there is a tradeoff between the memory usage and exporting overhead. If we export more often (with a smaller export interval), there are fewer flows in the interval and thus require fewer memory usage. Operators can configure the right export interval based on the number of flows in different time scales and the switch performance. For this paper, we set the time interval as 10 ms.

Deployment scenarios: Similar to NetFlow, we can deploy FlowRadar's encoded flowset either per port or per switch. The per-switch case would use less memory than per-port case because of multiplexing of flows. That is, it is unlikely that all the ports experience a burst in terms of the number of flows at the same time.

In the per-switch case, we still need to distinguish the incoming and outgoing flows (e.g., the two unidirectional flows in the same connection). One way to do this is to store the input port and output port as extra fields in the encoded flowset such as InputPortXOR and OutputPortXOR as what we did for the 5-tuple flow fields.[5] Another way is to maintain two encoded flowsets, one for incoming flows and another for outgoing flows.

FlowRadar can be deployed in any set of switches. FlowRadar can already report the per-flow counters in short time scales independently at each deployed switch. If FlowRadar is deployed at more switches, we can leverage network-wide decoding to handle more number of flows in a burst. Note that our network-wide decoding does not require full deployment. As long as there are flows that traverse two or more encoded flowsets, we start to gain benefits from network-wide decoding. Operators can choose where to deploy, and they know the flows where they deployed FlowRadar. In the ideal case, if all switches are deployed, then we know the per-flow counters at all locations, and the paths of the flows. Operators could also choose a subset of switches. For example, if we deploy only on ToR switches, the counters still cover all the events (e.g. loss) in the network, but we no longer know the exact locations where the flows appear in the network. As we mentioned in Section 5.2, the decoding success rate does not change as long as we have at least 2 flowsets, so partial deployment does not affect decoding success rate.

8 Related Work

8.1 Monitoring tools for data centers

Due to the problems of NetFlow, data center operators start to invent and use other monitoring tools. In addition to sFlow [40] and EverFlow [44], there are other in-network monitoring tools. OpenFlow [32] provide packet counters for each installed rules, which is only useful when the operators know which flows to track. Planck [34] leverages sampled mirroring at switches, which may not be sufficient for some monitoring tasks we discussed in Section 2. There are also many end-host based monitoring solutions such as SNAP which captures TCP-level statistics [41] and pingmesh [25] which leverages active probes. FlowRadar is complementary to the end-host based solutions by providing in-network view for individual flows.

8.2 Measurement data structures

There have been many hash-based data structures for measurement. Compared to them, FlowRadar has three unique features: (1) Store flow-counter pairs for many flows; (2) Easy to implement in merchant silicon; (3) Support network-wide decoding across switches.

Data structures for performance measurement and volume counting: Varghese et. al. proposed a group of data structures for loss, latency, and burst measurement [28, 37]. However, none of these solutions can maintain per flow metrics and scale to a large number of flows. There are many hash-based data structures that can keep per-flow state with small memory [15, 42, 36, 43]. However, most of them do not suit for NetFlow because they can only keep the values (i.e., per flow state). Instead, FlowRadar provides the key-value pairs (i.e., the flow tuples and the packet counters) and can scale to a large number of flows.

Hash-based data structures for storing key-value pairs: Cuckoo hashing [33] and d-left hashing [14, 38] are two hash table designs that can store key-value pairs with low memory usage. However, both are hard to implement in merchant silicon for NetFlow. This is because NetFlow requires inserting a flow immediately for an incoming packet so that follow up packets can update the same entry (i.e., *atomic read-update* operations). Otherwise, if one packet reads a cell that is being updated by a preceding packet, the counters become incorrect. Today, merchant silicon already has transactional memory that supports *read-update* operations in an atomic way for counters. However, typical merchant silicon can handle read-update operations against only a few (up to four) 4B- or 8B-long counters for each packet[6]. This is because to support high link rate of merchant silicon (typically a few Tbps today), merchant silicon must resort to a highly-parallelized packet-processing design, and the atomic-execution logic is at odds with such parallelism. In fact, to support such atomic read-update semantics for a small number of counters, merchant silicon has to employ various complicated hardware logic similar to operand forward [7].

A d-way Cuckoo hash table [33] hashes each key to d positions and stores the key in one of the empty positions. When all the d positions are full, we need to rebuild the table by moving items around to make room for the new key. However, this rebuilding process can only be implemented with switch software (i.e., the control plane),

[5] Similarly, one can easily add other flow properties (e.g., VLAN) as XOR sum fields.

[6] Note the total number of counters can still be larger; only the number of concurrently read-and-updatable counters is small.

because it requires multiple, and often-unbounded number of memory accesses [33]. Running the rebuilding process in switch software is not suitable for NetFlow, because NetFlow requires atomic read-update semantics.

d-left hashing splits a hash table with n buckets into d equal subtables each with n/d buckets, where each bucket contains L cells to hold up to L keys. d-left hashes a new key to d buckets, one in each subtable, and put the key in the bucket with the least load, breaking ties to the left. d-left requires first reading all Ld cells and testing if there is any match for an incoming flow. If there is a match, we increment the counter; otherwise, we put a new entry in an empty cell in the least-loaded bucket. There are two key challenges in supporting d-left: First, rather than read-update operations, d-left requires *atomic read-test-update* operations. The testing logic requires not only more ALUs and MUXes but also significantly increase the complexity of the atomic operation logic, making the critical section much longer in time. Second, d-left can only make insertion decisions after the testing on all Ld cells (each cell with 13 bytes 5-tuple fields and 4 bytes counter) are finished, which also increases the size of the atomic operation logic. Longer atomic operation duration can be a disaster for highly parallelized packet processing in merchant silicon.

In contrast, FlowRadar is easier to implement in merchant silicon, because of three reasons: First, FlowRadar only requires *atomic read-update* operations (i.e., increment/xor) rather than *atomic read-test-update*, which is much simpler in silicon design and has shorter atomic operation duration. Second, FlowRadar only requires atomic operations on a single cell and packets can update different cells in parallel. Thus FlowRadar requires significantly shorter atomic operations and is better fit for merchant silicon with high line rate.

It is impossible to support d-left with today's merchant silicon because the smallest d-left configuration (i.e., $d = 4$ and $L = 1$) needs to atomically read-test-update 4*17=68B, but today's silicon only supports 4*8B=32B. Thus, we compare FlowRadar with the basic d-left setting (i.e., $d = 4$ and $L = 1$) that may be supported in future silicon, and the setting recommended by [16] (i.e., $d = 3$ and $L = 5$) which is even harder to implement. To hold 100K flows on a memory of 2.74MB, the basic d-left has an overflow rate of 1.04%; both FlowRadar and the recommended d-left have no overflow. During flow bursts, FlowRadar can still report flows even when the counters cannot be decoded. Such flow information can be used for a variety of tasks like transient blackhole detection, route verification, and flow duration measurement. For example, to hold 152K flows in 2.74MB memory, the basic d-left has an overflow rate of 10%; the recommeded d-left has an overflow rate of 1.2%; FlowRadar can still decode all 152K flows (but not their counters).

Invertible Bloom filter Lookup Table (IBLT): FlowRadar is inspired by Invertible Bloom filter (IBF) [21] and Invertible Bloom filter Lookup Table (IBLT) [24]. IBF is used to keep a set of items. By comparing two IBFs, one can easily extract the differences between two sets. Rather then keeping a set of elements, FlowRadar needs to collect a key-value store of flows and their packet counters.

IBLT is an extension of IBF that can store key-value stores. Our counting table is built upon IBLT, but has two key extensions: *(1) How to handle value updates.* Since IBLT does not have a flow filter before it to identify if a key is new or old, it treats an existing key with a new value as a new key-value pair which has duplicated keys with existing key-value pairs. It then uses an arithmetic sum instead of a XOR sum in FlowXOR field, and a sum of hash values of the flows instead of a simple flow counter. This design takes more bits in both FlowXOR and FlowCount fields, which takes as much memory as FlowRadar uses for the flow filter. It also requires computations over large numbers (beyond 64bit integer), and more complex hash functions. Our experiments show that IBLT saves only 2.6% of memory for 100K keys but at the expense of 4.6 times more decoding time. *(2) How to decode the keys.* Our single node encoding scheme is similar to IBLT's, but takes much less time because of the simple FlowXOR and FlowCount fields. Moreover, with an extra flow filter, we support network-wide flow and counter decoding across multiple encoded flowsets.

9 Conclusion

We present FlowRadar, a new way to provide per-flow counters for all the flows in short time scales, which provides better visibility in data center networks. FlowRadar encodes flows and their counters with a small memory and constant insertion time at switches. It then introduces network-wide decoding of flowsets across switches to handle bursts of flows with limited memory. Our design can be improved in many aspects to further reduce the cost of computation, memory, and bandwidth, such as reducing the NetDecode time and better ways to leveraging redundancies across switch hops.

10 Acknowledgment

We thank our shepherd Sujata Banerjee, George Varghese, and the anonymous reviewers for their helpful feedbacks. This paper is partially supported by CNS-1453662, CNS-1423505, CNS-1413972, NSFC-61432009, and Google.

APPENDIX

A Algorithms

Algorithm 3: Decoding at a single node

```
1  Function SingleDecode(A)
2      flowset = ∅;
3      foreach c where CountTable[c].FlowCount==1 do
4          flow = A.CountTable[c].FlowXOR;
5          flowset.add(flow);
6          count = A.CountTable[c].PacketCount;
7          for j=1..k_c do
8              l=H_j^C(flow);
9              A.CountTable[l].FlowXOR =
               CountTable[l].FlowXOR ⊕ flow;
10             A.CountTable[l].FlowCount -= 1;
11             A.CountTable[l].PacketCount -= count;
12         end
13     end
14     return flowset;
```

Algorithm 4: Linear equations for CounterDecode

```
1  Function ConstructLinearEquations(A,S)
2      M=ZeroMatrix; b=ColumnVector;
3      foreach flow_t in S do
4          for j=1..k_c do
5              l = H_j^C(flow_t); M[l,t] = 1;
6          end
7      end
8      foreach CountTable[j] in A do
9          b[j] = CountTable[j].PacketCount;
10     end
```

References

[1] http://www.cisco.com/c/en/us/products/collateral/ios-nx-os-software/ios-netflow/prod_white_paper0900aecd80406232.html.

[2] deterlab.net.

[3] Flowradar implementation in p4. https://github.com/USC-NSL/FlowRadar-P4.

[4] NetFlow. https://www.ietf.org/rfc/rfc3954.txt.

[5] ns-3 simulator. https://www.nsnam.org/.

[6] Open vSwitch. http://openvswitch.org/.

[7] Operand forwarding. https://en.wikipedia.org/wiki/Operand_forwarding.

[8] P4 language consortium. p4.org.

[9] P4 simulator. https://github.com/p4lang.

[10] Packet loss impact on tcp throughput in esnet. http://fasterdata.es.net/network-tuning/tcp-issues-explained/packet-loss/.

[11] Solving the mystery of link imbalance a metastable failure state at scale. https://code.facebook.com/posts/1499322996995183/.

[12] Router overhead when enabling netflow. http://blog.tmcnet.com/advanced-netflow-traffic-analysis/2013/05/router-overhead-when-enabling-netflow.html, 2013.

[13] M. Alizadeh, A. Greenberg, D. A. Maltz, J. Padhye, P. Patel, B. Prabhakar, S. Sengupta, and M. Sridharan. Data center TCP (DCTCP). In *SIGCOMM*, 2010.

[14] Y. Azar, A. Z. Broder, A. R. Karlin, and E. Upfal. Balanced allocations. *SIAM J. Comput.*, 29(1), 1999.

[15] F. Bonomi, M. Mitzenmacher, R. Panigrahy, S. Singh, and G. Varghese. Beyond bloom filters: From approximate membership checks to approximate state machines. In *SIGCOMM*, 2006.

[16] F. Bonomi, M. Mitzenmacher, R. Panigraphy, S. Singh, and G. Varghese. Bloom filters via d-left hashing and dynamic bit reassignment extended abstract. In *Forty-Fourth Annual Allerton Conf., Illinois, USA*, pages 877–883, 2006.

[17] P. Cheng, F. Ren, R. Shu, and C. Lin. Catch the whole lot in an action: Rapid precise packet loss notification in data centers. In *NSDI*, 2014.

[18] Cisco. Netflow performance analysis. *White paper*, 2005.

[19] G. DeCandia, D. Hastorun, M. Jampani, G. Kakulapati, A. Lakshman, A. Pilchin, S. Sivasubramanian, P. Vosshall, and W. Vogels. Dynamo: Amazon's highly available key-value store. In *ACM SIGOPS*, 2007.

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

[21] D. Eppstein, M. Goodrich, F. Uyeda, and G. Varghese. What's the difference? efficient set difference without prior context. In *SIGCOMM*, 2011.

[22] C. Estan, K. Keys, D. Moore, and G. Varghese. Building a better netflow. *ACM SIGCOMM*, 2004.

[23] C. Estan and G. Varghese. Data streaming in computer networking. In *Workshop on Management and Processing of Data Streams*, 2003.

[24] M. T. Goodrich and M. Mitzenmacher. Invertible bloom lookup tables. In *arXiv:1101.2245v2*, 2011.

[25] C. Guo, L. Yuan, D. Xiang, Y. Dang, R. Huang, D. Maltz, Z. Liu, V. Wang, B. Pang, H. Chen, Z.-W. Lin, and V. Kurien. Pingmesh: A large-scale system for data center network latency measurement and analysis. In *SIGCOMM*, 2015.

[26] N. Handigol, B. Heller, V. Jeyakumar, D. Mazières, and N. McKeown. I know what your packet did last hop: Using packet histories to troubleshoot networks. In *NSDI*, 2014.

[27] S. Kandula, D. Katabi, S. Sinha, and A. Berger. Dynamic load balancing without packet reordering. *SIGCOMM Comput. Commun. Rev.*, 37(2), 2007.

[28] R. Kompella, K. Levchenko, A. Snoeren, and G. Varghese. Every microsecond counts: Tracking fine-grain latencies with a loss difference aggregator. In *SIGCOMM*, 2009.

[29] A. Kuzmanovic and E. W. Knightly. Low-rate tcp-targeted denial of service attacks (the shrew vs. the mice and elephants). In *SIGCOMM*, 2003.

[30] J. Mai, C.-N. Chuah, A. Sridharan, T. Ye, and H. Zang. Is sampled data sufficient for anomaly detection? In *Proceedings of the 6th ACM SIGCOMM conference on Internet measurement*, IMC '06, pages 165–176, New York, NY, USA, 2006. ACM.

[31] M. Mathis, J. Semke, J. Mahdavi, and T. Ott. The macroscopic behavior of the tcp congestion avoidance algorithm. In *SIGCOMM Comput. Commun. Rev.*, 1997.

[32] N. McKeown, T. Anderson, H. Balakrishnan, G. Parulkar, L. Peterson, J. Rexford, S. Shenker, and J. Turner. OpenFlow: Enabling Innovation in Campus Networks. *SIGCOMM Computer Communication Review*, 38(2), 2008.

[33] R. Pagh and F. F. Rodler. Cuckoo hashing. In *Algorithms — ESA 2001. Lecture Notes in Computer Science 2161*.

[34] J. Rasley, B. Stephens, C. Dixon, E. Rozner, W. Felter, K. Agarwal, J. Carter, and R. Fonseca. Planck: Millisecond-scale monitoring and control for commodity networks. In *SIGCOMM*, 2014.

[35] A. Roy, H. Zeng, J. Bagga, G. Porter, and A. C. Snoeren. Inside the social network's (datacenter) network. In *SIGCOMM*, 2015.

[36] H. Song, S. Dharmapurikar, J. Turner, and J. Lockwood. Fast hash table lookup using extended Bloom filter: An aid to network processing. In *SIGCOMM*, 2005.

[37] F. Uyeda, L. Foschini, F. Baker, S. Suri, and G. Varghese. Efficiently Measuring Bandwidth at All Time Scales. In *NSDI*, 2011.

[38] B. Vöcking. How asymmetry helps load balancing. *J. ACM*, 50(4), 2003.

[39] W. Vogels. Performance and scalability. `http://www.allthingsdistributed.com/2006/04/performance_and_scalability.html`, 2009.

[40] M. Wang, B. Li, and Z. Li. sflow: Towards resource-efficient and agile service federation in service overlay networks. *Distributed Computing Systems, International Conference on*, 0:628–635, 2004.

[41] M. Yu, A. Greenberg, D. Maltz, J. Rexford, L. Yuan, S. Kandula, and C. Kim. Profiling Network Performance for Multi-tier Data Center Applications. In *NSDI*, 2011.

[42] M. Yu, L. Jose, and R. Miao. Software Defined Traffic Measurement with OpenSketch. In *NSDI*, 2013.

[43] D. Zhou, B. Fan, H. Lim, D. G. Andersen, and M. Kaminsky. Scaling up clustered network appliances with ScaleBricks. In *SIGCOMM*, 2015.

[44] Y. Zhu, N. Kang, J. Cao, A. Greenberg, G. Lu, R. Mahajan, D. Maltz, L. Yuan, M. Zhang, B. Y. Zhao, and H. Zheng. Packet-level telemetry in large datacenter networks. In *SIGCOMM*, 2015.