LOAD BALANCE AND RESOURCE EFFICIENCY IN COMMUNICATION NETWORKS

A Dissertation

by

YUNHONG XU

Submitted to the Office of Graduate and Professional Studies of
Texas A&M University
in partial fulfillment of the requirements for the degree of

DOCTOR OF PHILOSOPHY

Chair of Committee,	Nick Duffield
Committee Members,	Krishna Narayanan
	Alex Sprintson
	Radu Stoleru
Head of Department,	Miroslav M. Begovic

May 2020

Major Subject: Computer Engineering

Copyright 2020 Yunhong Xu

ABSTRACT

Network management is critical for today's network. This study investigates both load balancing and resource efficiency in network management.

For load balancing, one unfavorable situation is that the active traffic uses a portion of the equal-cost paths instead of all. The root causes of load imbalance are not easily identified and located by network operators. Most research work related in this area concerns the design of load balancing mechanisms or network-wide troubleshooting that does not specify the causes of load imbalance. In this study, we describe a computational framework based on network measurements to identify the correlation mechanism causing the load imbalance. We also describe a novel framework based on Coprime to mitigate the load imbalance brought by hash correlations. In evaluation based on real network trace data and topologies, we have proved that we can reduces the error (CV or K-S statistic) by at least one magnitude.

For resource efficiency, today's network demands an increasing switch memory to support the essential functions, such as forwarding, monitoring, and etc. However, the cache memory is restricted when processing data streams in which the input is presented as a sequence of items and can be examined in only a few passes (typically just one). This study introduces a new single-pass reservoir weighted-sampling stream aggregation algorithm, Priority-Based Aggregation (PBA). A naive approach to order sample regardless of key then aggregate the results is hopelessly inefficient. In distinction, our proposed algorithm uses a single persistent random variable across the lifetime of each key in the cache, and maintains unbiased estimates of the key aggregates that can be queried at any point in the stream. Concerning statistical properties, we prove that PBA provides unbiased estimates of the true aggregates. We analyze the computational complexity of PBA and its variants, and provide a detailed evaluation of its accuracy on synthetic and trace data.

In addition to sampling, this study also considers placing classification rules into switches from various network functions. While much work has focused on compressing the rules, most of this work proposes solutions operating in the memory of a single switch. Instead, this study proposed

a collaborative approach encompassing switches and network functions. This architecture enables trade-off between usage of (expensive) switch memory and (cheaper) downstream network bandwidth and network function resources. Our system can reduce memory usage significantly compared to strawman approaches as demonstrated with extensive simulations and prototype evaluation with real traffic traces and rules.

ACKNOWLEDGMENTS

I would like to thank My adviser Dr. Nick Duffield for his patience and encouragement since I joined Texas AM University. His expertise in both theories and practice inspired me to propose the work in this dissertation. Also, I would like to thank Dr. Minlan Yu from Harvard University for her guidance throughout the course of this research.

Thanks also go to my friends and the department faculty and staff for making my time at Texas AM University a great experience. I also want to extend my gratitude to the National Science Foundation for providing the research funding.

Special thanks to Dr. Krishna Narayanan, Dr. Alex Sprintson, and Dr. Radu stoleru for being my committee member. Thanks to Anoosheh Heidarzadeh for attending my defense.

CONTRIBUTORS AND FUNDING SOURCES

Contributors

This work was supported by a dissertation committee consisting of Professors Nick Duffield, Krishna Narayanan and Alex Sprintson of the Department of Electrical and Computer Engineering and Professor Radu Stoleru of the Department of Computer Science.

The analyses depicted in Chapter 4 were conducted in part by Nick Duffield of the Department of Electrical and Computer Engineering.

All other work conducted for the dissertation was completed by the student independently.

Funding Sources

The dissertation research was supported by the National Science Foundation under Grants CNS-1618030 and CNS-1829349.

NOMENCLATURE

ECMP	Equal Cost Multiple Path
WCMP	Weighted Cost Multiple Path
CEF	Cisco Express Forwarding
CRC	Cyclic Redundancy Code
VLAN	Virtual LAN
TTL	Time-To-Live
SDN	Software Defined Networking
XOR	Exclusive OR
HH	Heavy Hitter
SH	Sample and Hold
ASH	Adaptive Sample and Hold
PBA	Priority-based Aggregation
PBASH	Priority-Based Adaptive Sample and Hold
WAN	Wide Area Network
CDF	Cumulative Distribution Function
ISP	Internet service provider
CV	Coefficient of Variance
CO2	Collaborative Overselection
APC	Approximate Packet Classifier
IPv4	Internet Protocol version 4
CAIDA	Center for Applied Internet Data Analysis

TABLE OF CONTENTS

Page

LIST OF FIGURES

FIGURE Page

LIST OF TABLES

TABLE Page

1. INTRODUCTION AND LITERATURE REVIEW*

Network management is important for communication networks. Two major topics in network management - load balancing and resource efficiency - attract researchers focus. In this chapter, we describe the motivation and the literature review of both of them.

1.1 Load Balance

1.1.1 Motivation and Contributions

Load balancing is critical in network operations and management [2] [3] [4]. One widely deployed method is Equal-Cost Multi-Path (ECMP) routing [2], in which packet forwarding to a single destination can occur over multiple least-cost paths based on a hash of the packet header. This allows the utilization of otherwise unused bandwidth on paths to the same destination. Hashing on header fields allows packets within each flow to follow the same path and for this reason, has been proposed in several load balancing schemes; see [3] [5] [6].

A challenging problem for hash-based load balancing is that reuse of identical or related hash-functions for load balancing in different routers can cause load imbalance [7]: a portion of links bear more traffic than others. Many network operators have reported such load imbalance [8][9]. We use the term *hash correlation* to describe associations between hash functions that can lead to load imbalance. One example is Cisco Express Forwarding (CEF) Polarization [8], where different switches repeatedly use the same hash algorithm, resulting in a switch selecting a single link for all traffic destined for one prefix, while other links were underutilized. Another example is the hashing imperfections mentioned in [10], which may incur the loss of high priority traffic.

As the best knowledge of us, there is no work on detecting hash correlations. One type of the related work is focusing on detecting the faults of network components [11] [12] [13] [14] [15] [16], which indirectly detect the load imbalance caused by those failures. Roy et al. [11] propose localizing partial network faults based on measurement at end-hosts. Li et al. [13] capture

*Part of this chapter is reprinted with permission from: Stream Aggregation Through Order Sampling, CIKM,{Pages: 909–918, (November)} © ACM, 2017. https://dl.acm.org/doi/10.1145/3132847.3133042.

the packet loss caused by hardware and software errors. These works assume errors to occur in a single link or switch failure, but in distinction with the work of the present paper, they cannot locate failure caused by the hash correlations between multiple switches. As far as we know, there has been no work on the detection of such correlations.

Besides the detection, the remediation, or even avoidance of hash correlations by design is also challenging. A conceptually simple approach is to use a different hash function for each switch. However, this is difficult to achieve because commodity chips support only a limited number of hash functions. For example, the software development kit for Broadcom switches supports RTAG7 [17], a hashing scheme utilizing seven independent hash functions. The Cisco Nexus 5500 Series offers eight versions of $crc8$ [18]. It's difficult to implement a large set of hash functions because the hash computation could become a bottleneck at high packet rates [19] [20]. For example, the authors in [20] discovered that the generation of Cyclic Redundancy Codes (CRCs) represents the main bottlenecks in iSCSI protocol processing. The authors in [21] mentioned that most theoretical research on CRC are impractical to implement in software or hardware for CRCs with arbitrary data and polynomial widths; In practical design, the network developers need to trade-off the design between high-speed designs and compact designs.

The research work on mitigating the load imbalance focuses on three root causes of load imbalance. The first type focuses on solving the imbalance caused by limitations of the hash functions [8] [22] [7] as described in the introduction. The universal algorithm [8] adds a 32-bit router-specific value to the hash function; however, this does not work for an even number of equal-cost paths. Zhou, etc., [22] propose to mitigate hash-function correlation by randomly setting the VLAN-id for each switch. However, randomizing the VLAN-id has the drawback that network operators are hindered from managing the multiple network domains effectively, e.g., in setting protection between ports based on the VLAN-id [23]. Ariel [7] selects a different hash function for a different value of the TTL field. However, this requires modifying the pipeline of packet processing to support the selection. In distinction, the Coprime-based approach can mitigate hash correlations without requiring changes to the packet processing pipeline.

Another type of works mitigates load imbalance caused by network failures; see [24] [25] [26]. These works respond to failures either by rescheduling the traffic or bypassing the faulty components, but do not address underlying problems cause by hash correlations. The third type of approach addresses load imbalance caused by large (or "elephant") flows whose packets follow the same path. [5] proposed splitting these into smaller "flowlets" that can be load-balanced over different paths. However, those works rely on hash functions to schedule the flowlets, which can also raise correlations among hash functions.

In order to overcome the limitations of the strategies above, we propose a novel framework residing in the Software-Defined-Networking (SDN) [27] controller. The framework aims to mitigate the effect of hash correlations by the selection of the divisors n used to map a flow's hash h to the router or switch egress interface over which it is forwarded, i.e., $h\%n$. Specifically, we establish that when such divisors in operation at different routers are *Coprime*, then the effects of underlying correlations between hashes are reduced significantly. This Coprime-based approach programs in the controller, which avoids the complexity of modifying the processing pipeline for packet forwarding. We modify the mapping from flow key (e.g. destination prefix) to egress interfaces according to parameters determined in the Coprime-based approach. The controller can send the mapping rules using Openflow interfaces provided by the software development kit (SDK) [28] in commodity switches. In summary, our approach is different from the previous methods in three aspects, first, we leverage the current hash functions already implemented in the chips instead of requiring a large of new hash functions; second, we do not modify the packet header, which may bring issues on processing speed and network security; and particularly, our method only programs in the controller and uses the existing interfaces to interact with the dataplane, which does not modify the hardware implemention of the process pipeline in the dataplane.

Instead of solving the correlations when they occur, we eliminates them for hierarchical networks. We propose a color combing approach by comparing the effect of hash functions on traffic to light passing through multiple triangular prisms: the traffic gets recombined at certain switches, and they can reuse the hash functions when the recombining happens. The color combing lever-

ages the trait of a hierarchical topology where traffic passes from one layer to another layer. This enables the reuse of the hash functions without causing the correlations.

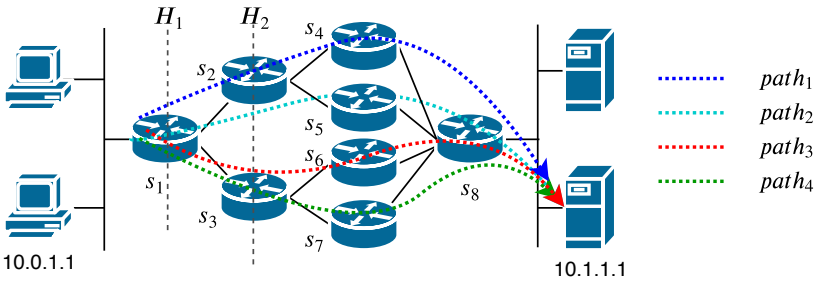
1.1.2 ECMP and Motivating Examples

The motivation of this work is that network operators observe recurring problem of load imbalance [8][9][26] due to hash correlations. In this section, we detail scenarios for correlations that illustrate why they are difficult to mitigate, while at the same time, can lead to performance degradation due to, e.g., link congestion [29].

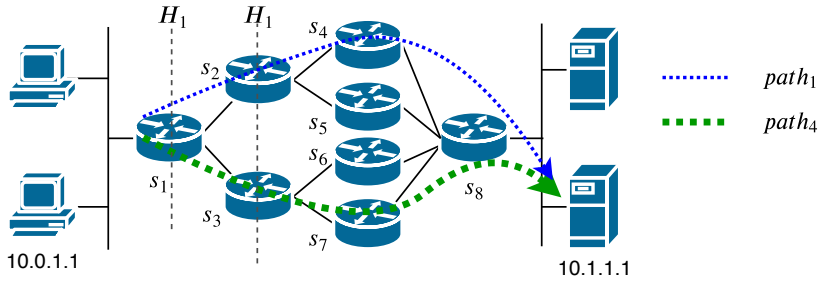
The ECMP - a routing protocol that allows the traffic to be transmitted across multiple paths of equal cost - identifies a set of routes, each of which is equal-cost towards the destination. The routes that are identified are referred to as an *ECMP group*. When forwarding a packet, the routing strategy decides which next-hop path to use based on a hash algorithm. That is, it selects one egress port based on a hash of fields of the packet header, e.g., the 5 tuples (source IP, source port, destination IP, destination port, and protocol). To be specific, the route of the packet is determined by the mapping from the hash to the egress port, e.g., $h \ \% \ n$, where h is the hash value, and n is the number of ports. When the routing hashes flows across paths according to customized weights instead of uniform ones, this variant of ECMP is called WCMP. And the set of routes identified in the WCMP process is referred as a WCMP group.

Each egress port of an ECMP group has the same probability of being selected. For example, Figure 1.1a shows the traffic from 10.0.1.1 to 10.1.1.1 uses four equal cost paths, which are labeled in four colors, where H_1 and H_2 are two independent hash functions. However, this does not work in several practical situations, as described in the following examples.

CEF Polarization In many networks, switches are configured with Equal Cost Multi-Path (ECMP) routing that forwards traffic by selecting one out of multiple ports based on a hash of 5 tuples for the purposes of load balancing. However, the balance of distribution of flows over ports by a given router can depend on the distribution of flows by other routers due to the hash-based mechanisms used to implement load balancing. Many enterprises rely on vendors to ensure there's no hashing problem, but in reality there exist such cases which have not been resolved by



(a) Different hash functions.



(b) The same hash function.

Figure 1.1: Two examples of ECMP.

the vendors; see the following example.

Cisco Express Forwarding (CEF) polarization is the effect when a hash algorithm chooses a particular path and the redundant paths remain completely unused [8]. CEF polarization can cause suboptimal use of redundant paths to a destination network. For example, Figure 1.1b shows that Switches s_1, s_2 and s_3 employ the same hash function (here is H_1), causing that the traffic from $10.0.1.1$ to $10.1.1.1$ only uses two routing paths instead of the four in Figure 1.1a; while Paths 2 and 3 are both idle. The polarization leads to double traffic on Paths 1 and 4, which may result in traffic congestion in burst stage. This is due to the hash correlation between Switches s_2 (s_3) and s_1: the routing choice of Switch s_2 (s_3) depends on Switch s_1. For example, Figure 1.2 illustrates the effect of the hash correlation: the packets on s_2 from s_1 have the same hash value of 0, and thus are forwarded to s_4 which has an index of 0.

Since the shortage of independent hash functions available in switches prevents simply assigning independent hashes to different devices, network operators derive multiple hash functions from one (here is CRC) by using a switch-specific seed (universal-id) [8]: a seed is an initial value to

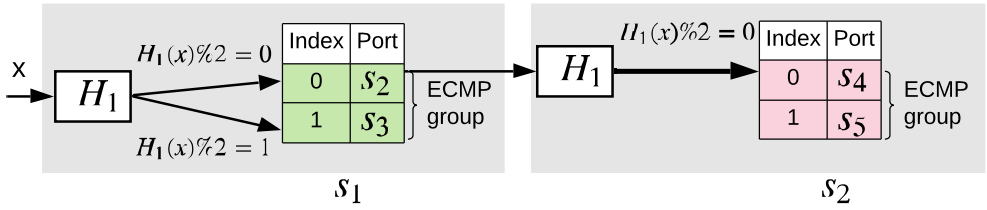


Figure 1.2: The effect of reusing the same hash function.

start the CRC computation via XORing the input data. However, this does not work for an ECMP group of even number of ports. For example, based on Theorem 1, all packets on Switch s_2 (Figure 1.1b) from Switch s_1 have the same routing choice even if they use different seeds for the same CRC; The proof of Theorem 1 can be found in Appendix A.2.

Theorem 1. $crc_i(x \oplus z_1)\%2 = crc_i(y \oplus z_1)\%2$ *leads to* $crc_i(x \oplus z_2)\%2 = crc_i(y \oplus z_2)\%2$*, where* x *and* y *denote the data with the same size in bytes,* z_1 *and* z_2 *are two seeds of* i *bits, and* i *is the integer to denote the width of the* CRC*.*

1.2 Sampling

Packet-based sampling schemes are widely used to characterize network traffic. In traffic measurement, network operators use packet sampling to handle the increasing amount of data transmitted in today's networks. In this subsection, we describe the motivation and the literature review of sampling. We also provide a short summary of our contributions.

We consider a data stream comprising a set of (key, value) pairs (k_i, x_i). Exact aggregation would entail computing the total value $X_k = \sum_{i:k_i=k} x_i$ for each distinct key k in the stream. For many applications, this is unfeasible due to the storage required to accommodate a large number of distinct keys. This constraint has motivated an extensive literature on computing summaries of data streams. Such summaries can be used to serve approximate queries concerning the aggregates through estimates $\widehat{X}_k$ of X_k, typically accomplished by assigning resources to the more frequent keys.

This problem of stream aggregation has drawn the the attention of researchers in Algorithms, Data Mining, and Computer Networking. Nevertheless, applications of this problem continue to emerge in new settings that bring their own challenges and constraints. These include: streams of transactional data generated by user activity in Online Social Networks [30], transactional data from customer purchases in online retailers [31], and streams of status reports from customer interfaces of utility service providers reported via domestic Internet service [32].

A well-established application for real-time streams of operational traffic measurements collected by Internet Service Providers (ISPs) has gathered renewed interest in the context of Software Defined Networks (SDN) [33]. These present the opportunity to move beyond industry standard summaries based on Sampled NetFlow and its variants [34]. Data Center operators increasingly wish to control the traffic in there networks at a finer space and time granularity than has been typical for Wide Area Networks, requiring per flow packet aggregates over time scales of seconds or shorter [35, 36]. A crucial management goal is to balance traffic loads over multiple network paths, and between servers. Two distinct analysis functions can support his goal:

- *Heavy Hitter Identification.* Heavy Hitters (HHs) are flows or groups of flows that contain a

disproportionate fraction of packets and/or bytes. These may be present in the exogenous loads, or may be indicative of underlying problems in the load balancing mechanisms [37].

• *General Purpose Summarization.* (key, aggregate) summaries over flows or groups for flows can be further aggregated over arbitrary subpopulation selectors, e.g., for what-if analyses for load balancing. This aggregation capability is present in Stream Databases developed to run on high speed traffic measurement systems [38].

Sampling is an attractive summarization method for supporting applications including those just described., First, sample sets can serve downstream applications designed to work with the original data, albeit with approximate results. Second, sampling supports retrospective queries using selectors formulated after the summary was formed. This enables sum queries over subpopulations whose constituent keys are not individually heavy hitters. Finally, sampling can often be tuned to meet specific goals constraints on memory, computation and accuracy that match data characteristics to query goals. We distinguish between two types of space constraint. The *working storage* used during the construction of the summary may be limited. An example is stream summarization of Internet traffic by routers and switches, where fast memory used to aggregate packet flows is relatively expensive [39]. But the *final storage* used for the finished summary generally has a smaller per item requirement than the working storage. A final storage constraint can apply, for example, when storage must be planned or pre-allocated for the summary, or when the size of the summary is limited in order to bound the response time of subsequent queries against it.

Reservoir Sampling [40] is commonly used to obtain a fixed size sample. In stream aggregation reservoir sampling, an arriving item (k, x) is used to modify the current aggregate estimate $\widehat{X}_k$ or X_k if k is in the reservoir, e.g., by adding x to $\widehat{X}_k$. If k is not in the reservoir, and the capacity of m is already used, a random decision is made whether to discard the arriving item, or to instantiate a new aggregate for k while discarding one of the items currently in the reservoir. In general, discard probabilities are not uniform, but are weighted as a function of aggregate size to realize estimation goals for subsequent analysis. In addition, estimates of retained items must be adjusted in order to maintain statistical properties of the aggregate estimates, such as unbiasedness. The

time complexity to process an arriving item and adjust the estimates of the retained items is a crucial determinant for the computational feasibility of stream aggregation. Fixed size summaries are essential in cases where the stream load can vary significantly over time and is not otherwise controlled. A prime example comes from Internet traffic measurement, where the offered load can varying significantly both due to time-of-day variation, and due to exogenous events such as routing changes. Reservoir sampling acts to adapt the sampling to variations in the rate of arriving items, e.g. to take a periodic fixed size sample per router interface.

In the earliest work in reservoir sampling k items from a stream of distinct keys [40], the n^{th} item is chosen with probability $1/n$, giving rise to a uniform sample. To approximately count occurrences in a stream with repeated keys, Concise Samples [41] used uniform sampling, maintaining a count of sampled keys. In network measurement Sampled NetFlow [34] takes a similar approach maintaining an aggregate of weights rather than counts. In Counting Samples [41], previously unsampled keys are sampled with a certain probability, and if selected, all matching keys increment the key counter with probability 1. Sample and Hold [42] is a weighted version of the same approach. Both schemes can be extended to adapt to a fixed cache size, by decreasing the sampling probability and resampling all current items until one or more is ejected. The set of keys cached by ASH is a PPSWR sample (sampling probability proportional to size without replacement), also known as bottom-k (order) sampling with exponentially distributed ranks [43, 44, 45]. The comparisons of this work use the form of ASH for Frequency Cap Statistics from [46], applied in the case of unbounded cap; see also an equivalent form in [47]. The number of deletion steps from a reservoir of size m in a stream of length n is $O(n \log m)$ and each such deletion step must process $O(m)$ items, based on generation of new randomizers variables for each item. By contrast, PBA requires only a single randomizer per key, and is able to maintain items in a priority queue from which discard cost in only $O(\log m)$. Concerning memory usage, PBA requires maintenance of larger working storage per item, while the implementation of ASH in [46] temporarily requires a similar amount during the discard step. Final storage requirements are the same. Step Sampling [48] is a related approach in which intermediate aggregates are exported.

Order sampling has been proposed as a mechanism to implement uniform and weighted reservoir sampling in the special case that items have unique keys [49]. In order sampling, all sampling decisions depend on a family of random order variables generated independently for each arriving item. For arrival at a full reservoir of capacity m, from the $m + 1$ candidate items (those currently in the reservoir and the arriving item) the item of lowest order is discarded. Several order sampling schemes have been proposed to fulfill different weighted sampling objectives; including Probability Proportional to Size (PPS) sampling [50] also known as Priority Sampling [51], and Weighted Sampling without Replacement [45, 52, 43]. Stream order sampling can be implemented as a priority queue in increasing order [51]. While order sampling can be applied directly to an unaggregated stream and samples aggregated post-sampling, this is clearly wasteful of resources.

Beyond sampling, many linear sketching approaches have been proposed; see e.g. [53, 54, 55, 56], [57]. More recently, L_p methods have been proposed in which each key is sampled with probability proportional to a power of its weight [58, 59, 60]. A general approach to sketch frequency statistics in a single pass is proposed in [61], with applications to network measurement in [62]. A drawback of sketch methods is that for a given accuracy, their space is logarithmic in the size of the key domain, which can be problematic for large domains such as IP addresses. Retrieval of the full set of aggregates (as opposed to query on specific keys) is costly, requiring enumerating the entire domain for each sketch; tuning of the sketch for specific queries, e.g., using dyadic ranges, is preferable. In our case, the full summary can be read directly in $O(m)$ time. Space factors in the sketch-based methods also grow polynomially with the inverse of the bias, whereas our method enables unbiased estimation. Beyond these comments, we do not perform an explicit comparison with sketch-based methods, instead referring the reader to a comparative evaluation of sketches with ASH for subpopulation queries in [47].

This dissertation proposes Priority-Based Aggreation (PBA), a new sampling-based algorithm for stream aggregation built upon order sampling that can provide unbiased estimates of the per key aggregates. PBA and its variants provide greater accuracy across a variety of heavy hitter and subpopulation queries than competitive methods in data driven evaluations. Our specific contributions

are as follows:

Estimation Accuracy. PBA is a weighted sampling algorithm developed from Priority Sampling that yields a stream summary in the form of unbiased estimates of all aggregates in the stream. A modification of PBA uses biased estimation to reduce error for smaller aggregates, while having negligible impact on accuracy for larger aggregates. In experimental comparisons with a comparable sampling based method, Adaptive Sample and Hold [42, 63], our methods reduced weighted relative estimation error over all keys by between 38% and 65% at sampling rates between 5% and 17% when applied to synthetic and network traffic traces. The accuracy for rank queries was also improved.

Computational Complexity. To the best of our knowledge, PBA is the first algorithm to employ order sampling based on a single random variable per key in the context of stream aggregation. This enables PBA to achieve low computational complexity for updates. It is average $O(1)$ to process each arrival that is either added to a current aggregate, or that presents a new key that is not selected for sampling. The exception comes when an arriving key not currently in storage replaces an existing key; the complexity of this step is worst case $O(\log m)$ in a reservoir of capacity m. Retrieval of the estimates is $O(1)$ per key.

Priority-Based Adaptive Sample and Hold (PBASH). We incorporate the well known weighted Sample and Hold [42] algorithm as a pre-sampling stage, for which the sampling probabilities are controlled from the adaptation of the PBA second stage. This enables us to exploit the computational simplicity of the original (unadaptive) Sample and Hold algorithm while taking advantage of the relatively low computational adaptation costs of PBA, as compared with existing versions of Adaptive Sample and Hold [64, 63].

1.3 Collaborative Selection

As the increasing scale of the communication network, traffic monitoring requires large amount of resources that may cause a burden on the network devices. One example is that the switch memory cannot meet the demand of multiple monitoring tasks. In this subsection, we review the work on handling the conflict of limited switch memory and the demand of the monitoring tasks.

Software Defined Networking (SDN) is a powerful enabler for Network Function Virtualization (NFV). By moving network appliance functionality from proprietary hardware to software, NFV promises to bring greater openness and agility to network data planes [65]. An SDN controller can install classification rules at switches and set up tunnels to direct the selected traffic to receivers that run various network functions [66]. In the cloud context, traffic is classified for different L4-L7 network functions such as software load balancers [67], WAN optimizers, and proxies. Operators may mirror packets to analysis functions (e.g., intrusion detection, or deep packet inspection) to debug network problems [68].

In enterprise networks, about half of network devices are middleboxes that provide network functions [69]. These functions can require millions of fine-grained traffic rules. Tens of thousands of 5-tuple flows may be active at each top-of-rack switch rack [70, 66], each potentially requiring a rule to mirror selected packets towards network analysis functions [68]. A cloud can contain 100K tenants [67], with a customized packet processing pipeline for each pair of communicating tenants requiring 200M rules, assuming 2% of tenant pairs communicate with each other, as in [71].

Current switch memory is insufficient to store these numbers of rules. Even storing 500K 5-tuple prefix rules would exhaust typical on-chip SRAM (e.g., 20-60 MB in Trident2 [72])), most of which is already used for regular packet forwarding and other match-action rules [73]. The reduction in memory consumption attainable by rule compression [74, 75, 76] does not change the conclusion, with compression ratios of e.g. 29% for 2 decisions and 48.3% for more distinct decisions classifier [76]. While approximate data structures such as Bloom Filters and Cuckoo Filters [77], can reduce the memory usage by a further 50%, no systematic treatment of the consequences of the resulting false positives has been provided.

There have been a variety of hash-based data structures for storing key-value pairs (which is classification rules and actions in our setting) such as Bloomier Filter [78] [79], SILT[80], Shared-node Fast Hash Table (SFHT) [81], and Cuckoo Filter [77]. We choose Cuckoo Filter because it provides high memory efficiency and low hash collisions. It is also easy to implement with commodity switches because it takes constant time to match packets to rules. The general idea of overselection and prefix aggregation can be adapted to other hash-based data structures beyond Cuckoo Filters.

Bloomier filter[78] extends the original Bloom filter to support value query rather than key detection. The most novel merit of Bloomier filter is it need not explicitly store the actual value of an item. When inserting a value, it first computes the buckets via different hash functions, just like what is done in Bloom filter. Then it chooses a bucket for the item. What it stores in the bucket is the XOR results of the key and all the values in other candidate buckets. When looking up a key, it only needs to XOR all the values in the computed buckets. The most advantage of Bloomier filter is the fixed look up time for every query and high storage efficiency. However, since the XOR value depends on a batch of buckets, it is very hard to update in Bloomier filer but to rebuild a new one.

SILT[80] builds the cascaded Key-Value storages in three different levels with different emphasis between the memory efficiency and access overheads. The first level storage is easiest to access while the third level storage loads data in a very tight format. The number of operations to access the third level data is several times larger than that of the first level data. Thus it puts the frequently accessed data in the first level. But it can't support fixed query time. Also, data stored in the deeper level storage is hard to be updated.

Song *et al.* [81] improves the counting Bloom filter to support storing values. The proposed SFHT uses an on-chip counting Bloom filter as an index table that can rapidly detects whether an item is in the storage. For the queries hitting the index table, SFHT retrieves the value stored in the off-chip memory. This approach significantly reduce the number of off-chip memory accesses so that accelerates the lookup performance. It also supports fast update in the price of redun-

dantly storing some values. The main drawback of SFHT is similar to SILT: it has diverse query time for different keys. Besides, some items need more copies in SFHT to guarantee high update performance, though the number is smaller.

After comparing these candidate storage structures, we believe Cuckoo Filter is the best one to meet our requirement. It inherits the high load factor (more than 95%) and one-access look up per query from original Cuckoo hashing. It also achieves more memory efficiency through storing fingerprints, which perfectly meets our philosophy of overselection.

The authors of [82] propose using a Bloom pre-filter to select packets in heavy-hitter flows, then use deterministic classification on non-matching packets. This approach resides entirely in the switch and does not employ approximate classification outside the pre-filter.

To reach the high throughput for packet classification, Ternary Content Addressable Memory (TCAM) is widely used and the de facto industrial standard of packet classification. The main drawbacks of TCAM is the limited capacity.

There have been many algorithms on aggregating prefixes reduce the number of rules at switches [83, 84]. Rather than aggregating existing rules, we leverage over-aggregation that allows an aggregated prefix to cover unwanted traffic and introduces overselection. [83] aggregates prefixes belonging to different actions and punches "holes" for the over-aggregated prefixes. Though the holes are somewhat similar to our Blacklist, they store both the valid prefixes and the over-aggregated prefixes in the same trie so it has to search twice for a single IP address. Instead, we use the separate blacklist to filter unwanted packets in advance.

A recent work [75] introduces loss compression of classification rules to reduce the memory usage but may take the wrong actions to the traffic. Instead, never matches packets to the wrong receivers, but only overselects traffic to some receivers. Such overselection has a much smaller impact compared to wrong delivery. In addition to overselection caused by prefix aggregation, we also introduced overselection caused by hash collisions and understand the tradeoffs of the two types of overselection.

Lossy Compression that allows compress rules with errors. It defines a special wildcard class

'?' to represent the rules that fails to be added in the new set which is derived by dynamic programming in a prefix trie. If a packet falls into this class, it is forwarded to a second level classifier to get further classified. The '?' class is actually a mistake of the classifier and need to be fixed by the extra patch. In our traffic selector, we will never have any mistakes (i.e., a wanted packet is ensured to be forwarded to the right place). What we have involved are that false positive packets that form the redundant traffic on the link between the selector and middlebox, and it can be rapidly eliminated by the middlebox. To our best knowledge, none of the similar work has been done yet.

Thus, many works focus on the rule compression in TCAM based packet classification . Bit Weaving [85] first migrates the DON'T CARE bits (*) of a non-prefix TCAM entry to the tail, which makes it become a prefix. Then it merges the prefixes to reduce the number of entries. TCAM Razor [76] translates the the classifiers to the decision zones in a diagram and compresses the entries through dynamic programming. [74] construct the new wild card rules in cache to reach the best trade off point between hardware and software.

For one-dimension rule aggregation, the main solution is to utilize the trie based aggregation to compress the prefixes [83, 84]. The most famous one is the Optimal Routing Table Constructor (ORTC) in routing table compression that can reach the optimal bound in theory. These methods work on the classifiers which target on the Longest Prefix Matching (LPM). The basic idea is to adjust the prefixes aggregation combinations (through expand some high level prefixes and aggregate the derived prefixes with others) to find out the best solution with fewest prefixes.

Since our traffic selector does not involve LPM and priorities among rules, these solutions cannot be directly transferred.

Note that all these prefix aggregation solutions work on the semantically equivalent level. However, in our scenario, we can even aggregate more in the price of involving some acceptable over-selections. In this way, we can break the theoretical bound of the semantically equivalent level prefix aggregation to save more memory.

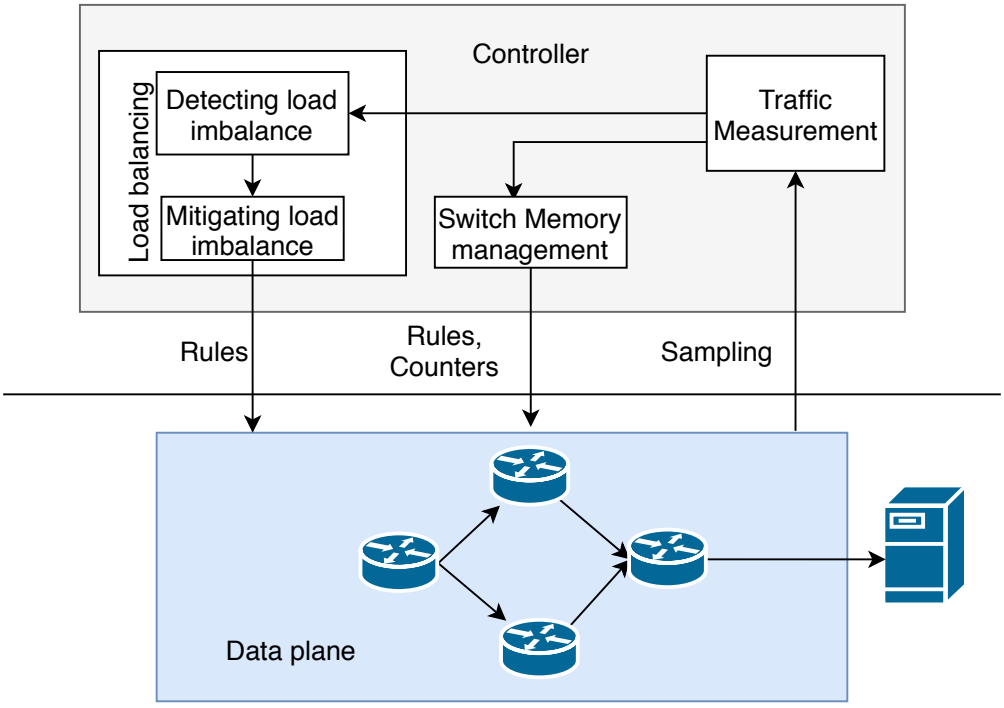


Figure 1.3: The framework

1.4 Outline

The framework of the dissertation is shown in Figure 1.3: detecting the load imbalance, mitigating the load imbalance, switch memory management, and sampling for traffic measurement which can serve for both detecting the load imbalance and switch memory management. In this subsection, we describe the outline of the following work.

Chapter 2 describe the detection of load imbalance caused by hash correlations. We first identify the load imbalance caused by the hash correlation via a hypothesis testing framework; see § 2.1. Section 2.2 depicts the approach of locating hash correlations; Section 2.3 evaluates the performance under a set of realistic experimental settings.

Chapter 3 focuses on mitigating hash correlations. Section 3.1 presents the Coprime theorem and the Coprime-based approach to mitigate the hash correlations; Section 3.2 describe a Coprime selector for network-wide correlations, which reduces the memory usage and promotes the traffic uniformity via diversifying the error bounds; the section also describes how to apply the Coprime-

based approach to handle WCMP; Section 3.3 presents the color-combining trick for hierarchical networks. Section 3.4 emphasizes on the evaluation under a set of realistic experimental settings.

Chapter 4 presents the sampling techniques. Section 4.1 describes the PBA algorithm and establishes unbiasedness of the corresponding estimators. Section 4.2 describes four optimizations of these basic algorithms. Section 4.2.1 describes *Deferred Update* for which we show that the unbiasing of estimates that must be performed on all aggregates after another is discarded can be deferred, for each such aggregate until an item with matching key arrives. Section 4.2.2 describes pre-aggregation of successive items with the same key in the input stream. Section 4.2.3 describes the use of Sample and Hold as an initial sampling stage, and how its adaptation is controlled from PBA. Section 4.2.4 describes a scheme to reduce estimation errors for small aggregates through the introduction of bias. Section 4.3 specifies the algorithm incorporating these optimizations, describes our implementation, and reports on computational and space complexity. Section 4.5 describes data driven evaluations. Proofs are deferred to Section 4.4.

Chapter 5 describe the algorithms on collaborative selection. In detail, Section 5.2 details switch design combining approximate prefix matching and lossy compression through aggregation. Section 5.3 extends the design to rules with multiple matching fields and multiple actions. Section 5.4 distributes rules to multiple switches to avoid overstripping a single switch. In Section 5.5, we evaluate the performance of under a wide range of configurations using network traces. In Section 5.6, we examine the downstream impact of overselection and argue that in many cases the impact is minimal apart from a small increase in load.

Chapter 6 conclude our work.

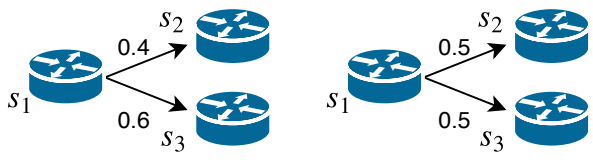
In this chapter, we describe an approach called *IcorLIm* to detect the hash correlations. We first identify the load imbalance caused by the hash correlation via a hypothesis testing framework; see § 2.1. We then present our approach of identifying the hash correlation in § 2.2.

To formulate our detection framework, we introduce the following concepts: *the victim*: A victim is one ECMP group that displays load imbalance; otherwise, the ECMP group is *uniform*. When there is a hash correlation between one ECMP group and a victim, we call this ECMP group the correlated ECMP group of the victim; and the two are correlated with each other.

2.1 Identifying Victim

Determining whether one ECMP group is uniform (not victim) requires to measure the difference between the observed distribution and the uniform distribution. One metric to evaluate the difference is the squared error. For example, Figure 2.1 shows that the squared error between the two distributions is 0.01, where the float number in the figure shows the ratio of number of flows on one port. However, the squared error itself can only quantify the difference between the two distributions. To determine whether they are different, we need a threshold, the squared error below which, the difference is significantly small. But it is non-trivial to set a threshold because it depends on the distribution.

To overcome the drawback of the squared error, we frame the problem by a hypothesis testing:



$$squared\ error: (0.4-0.5)^2+(0.6-0.5)^2=0.02$$

Figure 2.1: One example of the squared error of two distributions.

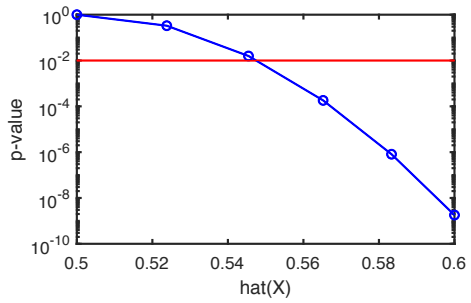


Figure 2.2: One example of the p-value for an ECMP group of two ports, where $\widehat{x}$ indicates the ratio of the flows of one port.

the null hypothesis is that the ECMP group is uniform. We employ the K-S statistic [86], which indicates the supremum distance between two Cumulative Distribution Functions (CDFs): the CDF of the observed distribution and the CDF of the uniform distribution.

We need to check whether the K-S statistic is significantly small, and thus, the ECMP group is uniform. The null hypothesis can be accepted or rejected by comparing the p-value - the probability of two CDFs being the same - to a probability threshold, denoted by T, beyond which the hypothesis is accepted. For example, Figure 2.2 shows the p-value for one ECMP group; When the ratio of flows on one port out of two is beyond 0.54, the p-value is below the threshold (here is 0.01), and we reject the hypothesis of the ECMP group being uniform; and by definition, it is a victim.

2.2 Locating Hash Correlations

The next step is to find the correlated ECMP groups with the victim.

Let $p(v', v)$ denote the pairwise p-value of two ECMP groups v' and v, we compute $p(v', v)$ by the distribution of *pairwise flows* over the ports of v, where *pairwise flows* denotes the traffic passes through both ECMP groups. For example, Figure 2.3 shows that the number of *pairwise flows* of ECMP groups v_1 and v_4 is 50 (labeled by red in Figure 2.3); the distribution over the two ports of v_4 is $0 : 1(0 : 50)$.

To determine whether ECMP group v' and the victim v are correlated or not, one method is to

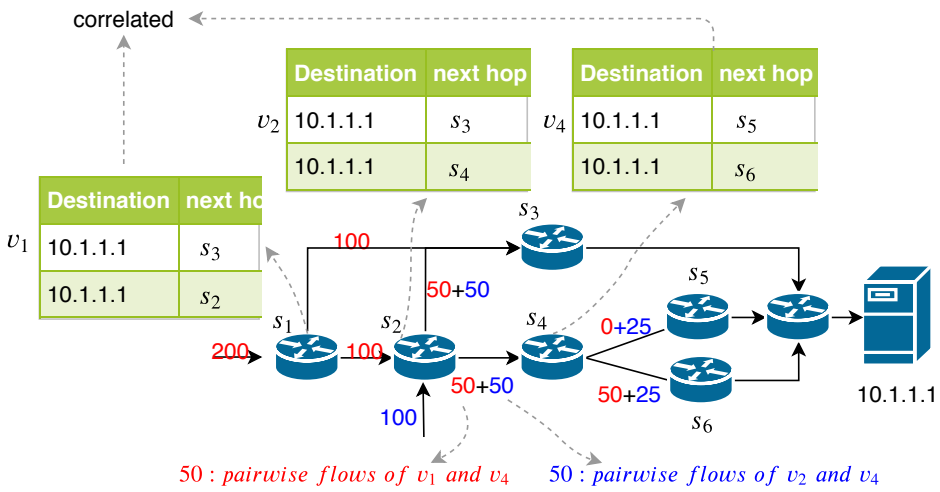


Figure 2.3: An example of the correlations between non-neighbor ECMP groups.

compare $p(v', v)$ with a probabilistic threshold, below which, the two ECMP groups are correlated. However, the method does not work when the two ECMP groups are non-neighbors. For example, Figure 2.3 shows that the two ECMP groups v_1 and v_4 are correlated; and both $p(v_1, v_4)$ and $p(v_2, v_4)$ are below the threshold, e.g., 0.01, because the distribution of pairwise flows of v_1 and v_4 over the two next-hops s_5 and s_6 is 0:1 (0:50 in Figure 2.3); and the distribution of (v_2, v_4) over s_5 and s_6 is 1:3 (25:75 in Figure 2.3), leading to small p-values according to Figure 2.2.

To identify the non-neighbor correlation, we propose to sort the ECMP groups along the same routing path in ascending order by $p(v', v)$ and identify the first rank as the correlated ECMP group, where v is the victim and v' is another ECMP group along the same routing path with v. Note that we also need to sort the ECMP groups in ascending order by the distance to v if they have the same p-value. We use the term *Min-based* to denote this approach.

Min-based cannot handle multiple hash correlations since it identifies only one correlated ECMP group by obtaining the first rank. Our approach of solving multiple correlations is to group the ECMP groups to many clusters. Starting from a victim v, we trace back along the routing path, and put the ECMP groups into the cluster until we reach another victim v' or no ECMP group. Note that v' is also inside the cluster of v. Figure 2.4 shows one example of the grouping; both ECMP

groups v_2 and v_4 are victims. From v_2, we trace back the routing path and put v_1 into $Cluster_1$, leading to two ECMP groups: v_1 and v_2 in $Cluster_1$. It is similar for v_4, and when we detect v_2 in the backward process, it terminates because v_2 is a victim. $Cluster_2$ also comprises two ECMP groups: v_2 and v_4.

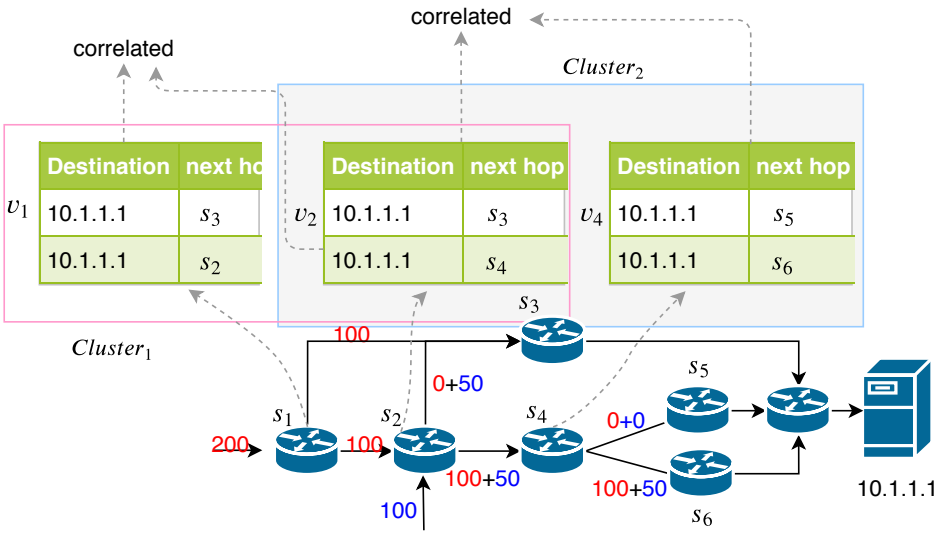


Figure 2.4: An example of the subgroups.

In addition to the hash correlations within a cluster, they also exist among clusters. For example, Figure 2.4 shows that v_1 and v_4 are also correlated. We can identify them based on the following properties:

Property 1. Transitive: if two ECMP groups are correlated, they are also correlated with the third ECMP group, which correlates with either of them; this is due to the rolling property of XORs and CRCs; see Theorem 1.

Property 2. Directional: when ECMP group v' is correlated with Victim v, the reverse case where v is correlated with v' is not supportive. The correlation is directional because the forwarding path is directional.

The above properties of hash correlations enable us to find the correlations among clusters in two steps: first, build a directional graph: the node is the ECMP group, and the edge is the correlated pair (v', v); and second, identify each pair (v'', v), where v is reachable from v'' and hence, v'' is correlated with v.

2.3 Evaluation

In this section, we evaluate the performance of IcorLIm. First, we compare IcorLIm with two strawman approaches: Threshold-based and Min-based. The results show that IcorLIm can achieve an F1 score of 1.0, but the other approaches can only achieve around 0.6.

2.3.1 Experiment Setup

Topology: We conducted the evaluations under two topologies: B4 WAN [10], and CLOS [87] [88]. B4 WAN is a mesh topology, which connects multiple data centers; and CLOS is a tree-based topology, which is used in Jupiter [88]: Google's datacenter. We study three CLOSs and they distinguish by the number of nodes in an aggregation block: CLOS-1 with four nodes, CLOS-2 with eight nodes, and CLOS-3 with 16 nodes. The information of the topologies is shown in Table 2.1; the connection within one aggregation block of CLOS-1 and CLOS-2 is shown in Figure 2.5.

Topology	#Nodes	#Nodes in an aggregation block
B4 WAN	12	-
CLOS-1	-	4
CLOS-2	-	8
CLOS-3	-	16

Table 2.1: B4 WAN and three CLOSs.

Traces: we use the traffic traces from the Facebook Web servers [35], which samples the packets at the rate of 1:30000. We interpolate the traces to simulate the actual traces according to the

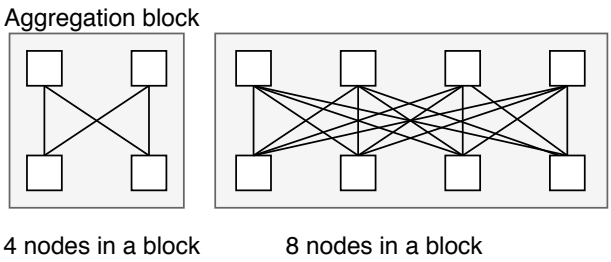


Figure 2.5: The connection inside one agreggration block of CLOS-1 and CLOS-2.

distribution from the work [35]. We select a 10-min section of the trace, which contains 1.5 million 5-tuples flows.

Hash functions: We use the RTAG7 [17] for ECMP which includes seven hash functions - two $CRC16$s, four XORs and one $CRC32$; and the $CRC32$ can be used as two $CRC16$s: the lower 16 bits and the higher 16 bits.

Correlation settings: we study four correlation types: no hash correlation between two ECMP groups, one correlation between two neighbor ECMP groups, one correlation between two non-neighbor ECMP groups, and the mixed type with multiple correlations.

Metric for simulations: we use F1 score [89] to evaluate the performance of the approaches because it considers both precision and recall: a higher value indicates a better performance. The F1 score is computed by 2*((precision*recall)/(precision+recall)), where precision [90] is defined as (true positives (TPs))/(true positives (TPs) + false positives (FPs)), and the recall [90] is defined as (TPs)/(TPs + false negatives (FNs)). For example, if four ECMP group pairs out of ten have hash correlations, IcorLIm identifies two out of four and one false positive; we have TPs = 2, FPs = 1, FNs = 2, precision = 2/3, recall = 0.5, and F1 score = 0.57.

The data points shown on the figures in the subsequent subsections are the average value of ten iterations. To diverse the traces, we map the source IP address and the destination IP address to random IP addresses in each iteration.

2.3.2 The Results

In this section, we compare IcorLIm and the two strawman approaches: Threshold-based and Min-based under B4 WAN, CLOS-1, CLOS-2 and CLOS-3. The results are shown in Figures 2.6, 2.7, 2.8 and 2.9.

Figures 2.6 shows that, in B4 WAN, IcorLIm improves the F1 score by 40% compared with Threshold-based. Both of IcorLIm and Min-based can reach an F1 score of 1.0, but Threshold-based can only obtain an F1 score of 0.7. The reason is that only single hash correlation exists along a routing path in B4 WAN; both IcorLIm and Min-based can identify it by sorting the p-values; but Threshold-based raises false negatives since it cannot handle non-neighbor correlations.

We refer from Figures 2.7, 2.8 and 2.9 that, for all CLOSs, the F1 score of IcorLIm can reach 1.0, however, the other two stawmans can only achieve around 0.6. IcorLIm increases the F1 score by around 50% because both Threshold-based and Min-based cannot handle multiple correlations along the same routing path.

The results also indicate that, for all topologies, the number of flows passing through one ECMP group affects the F1 score. For example, the F1 score of CLOS-1 increases with more flows and gets stable at around 100. The reason is that a small number of flows cannot accurately represent the distribution, leading to an inaccurate p-value. In addition, CLOS-3 requires more flows than CLOS-1 because an ECMP group includes more ports for CLOS-3.

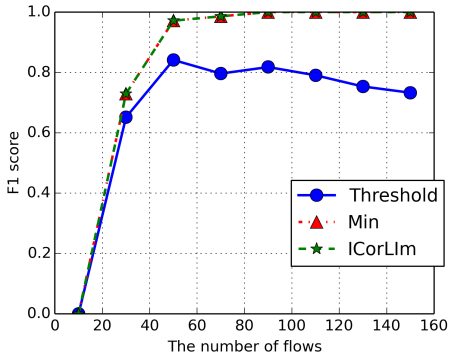


Figure 2.6: Comparing IcorLIm with the Threshold-based and the Min-based for B4.

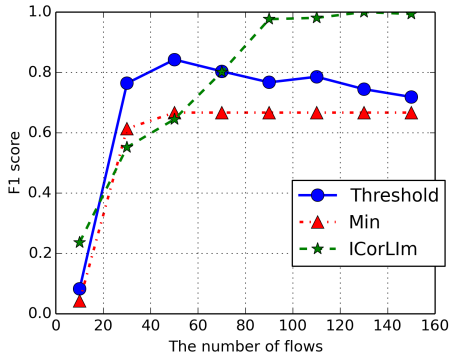


Figure 2.7: Comparing IcorLIm with the Threshold-based and the Min-based for CLOS-1.

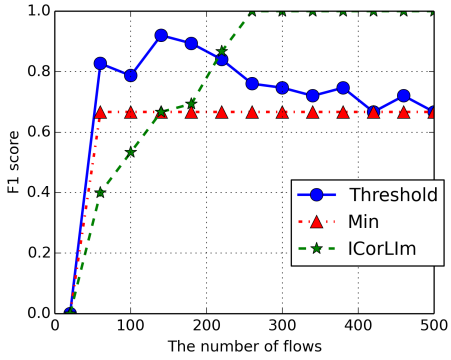


Figure 2.8: Comparing IcorLIm with the Threshold-based and the Min-based for CLOS-2.

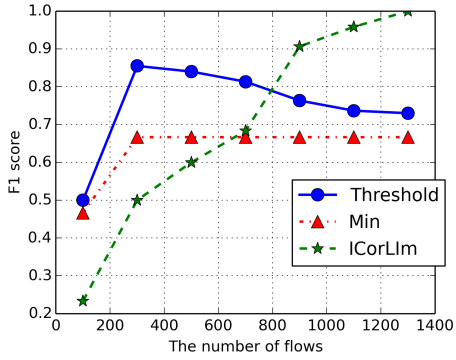


Figure 2.9: Comparing IcorLIm with the Threshold-based and the Min-based for CLOS-3.

3. MITIGATING HASH CORRELATIONS

In the last chapter, we have proposed algorithms to detect the load imbalance caused by hash correlations. In this chapter, we describe the approaches to mitigate hash correlations.

3.1 Mitigating Correlation

In this section, we describe an approach to mitigate the hash correlation between two ECMP groups. Two even-size ECMP groups for the traffic destined to the same destination may cause hash correlations when they use the same CRC hash function (with different seeds); while either ECMP group being odd-size does not. This inspires us to change the size of one ECMP group from even to odd, making them coprime to each other, without changing the physical connection. We use the term *correlated ECMP groups* to denote the groups which manifest hash correlation. Following the theorem for mitigating hash correlations based on Coprime (§ 3.1.1), we present the Coprime-based approach in § 3.1.2.

3.1.1 A Theorem Concerning Coprimes

Theorem 2 shows mitigating hash correlations through Coprime; the proof is described in Appendix A.3.

We use Coprimes to transform two hash functions - $H_1 = H\%m_1$ and $H_2 = H\%m_2$ - to $H_1' = H\%q_1\%m_1$ and $H_2' = H\%q_2\%m_2$; where H denotes a hash function, m_1 and m_2 are two non-coprime integers, and q_1 and q_2 are two Coprimes.

Let X_1 and X_2 denote the output space of H_1' and H_2', respectively, if $\forall i, \forall j, P(X_2 = j|X_1 = i) = P(X_2 = j)$, no hash correlation exists between the two hash functions. We have,

Theorem 2. $\forall i, \forall j, P(X_2 = j|X_1 = i) = P(X_2 = j)$ *if the following two conditions are satisfied:*

Condition 1*:* $q_1 \gg m_1$ *or* $q_1\%m_1 = 0$*, and* $q_2 \gg m_2$ *or* $q_2\%m_2 = 0$*;*

Condition 2*:* $\widehat{H} \gg q_1q_2$*, where* $\widehat{H}$ *is the highest hash value of Hash function* H*.*

The theorem shows that the correlation between the two hash functions H_1 and H_2 can be

eliminated via applying Coprimes to them.

3.1.2 Coprime-based Approach

We propose a Coprime-based approach to mitigate the hash correlation between two ECMP groups given Theorem 2. The approach comprises two parts: choosing a Coprime q and the routing mapper (§3.1.2.1), which replicates each entry in the ECMP group to fit q slots in a commodity switch; we use the term *Mapped ECMP (MCMP) group* to denote the replications.

One way to take the Coprime into effect is to alter the data pipeline: recording the Coprime for each ECMP group and adding a modulus operation. Figure 3.1 shows the procedure: The first two entries in the multi-path table store an ECMP group for the packets to Destination $10.1.1.0/24$. The next two entries in the table carry the ECMP group for the packets destined to prefix $10.1.2.0/24$. When getting a match in the Longest-Prefix-Match table (LPM) for a packet, the switch determines the egress port by looking up the multi-path table. For example, a packet with destination $10.1.2.1$ points to the ECMP group with an index of 2 in the multi-path table. The switch calculates the offset into the multi-path table for a packet by, first, hashing over the header fields, e.g., the 5-tuple, and second, taking modulo of the coprime (here is 5) and the number of entries for the ECMP group (here is 2) sequentially. The modulo (here is $9 \ \% \ 5 \ \%2 = 0$) added to the group's base index (here is 2) determines the egress port for the incoming packet ($0 + 2 = 2$).

However, the above approach needs the effort from the chip vendor, even resulting in replacing the current chips to meet the demand. Instead, we propose an approach to program in the controller which interacts with the data plane via the existing interfaces. Figure 3.2 shows an example of the approach: we use five entries in the multi-path table to carry the MCMP group for the packets destined to prefix $10.1.2.0/24$. A packet with destination $10.1.2.1$ points to the MCMP group with an index of 6 in the multi-path table. The switch calculates the offset into the multi-path table for a packet by hashing over the header fields, and then taking modulo of the number of entries for MCMP group (here is 5). The modulo (here is $9 \ \% \ 5 = 4$) added to the group's base index (here is 2) generates the egress port for the incoming packet ($(9 \ \% \ 5) + 2 = 6$). In this example, the controller interacts with the data plane to put the MCMP group to the multi-path table.

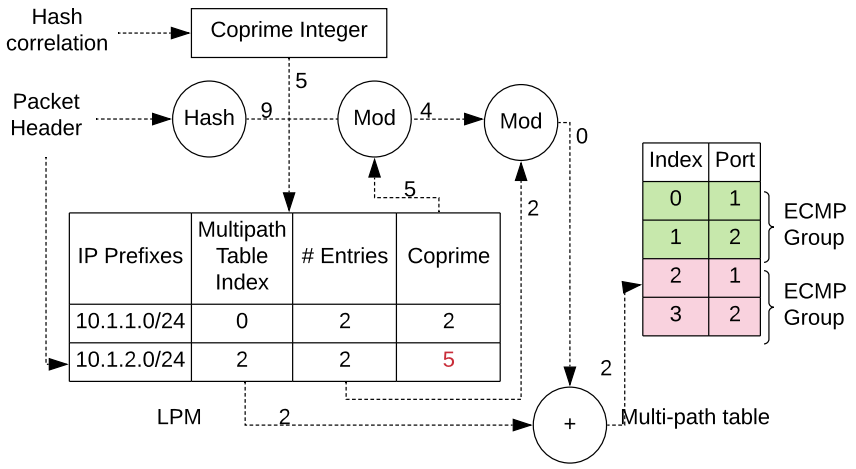


Figure 3.1: A procedure of applying the Coprime via changing the pipeline of packet processing.

3.1.2.1 Routing mapper

We associate the Coprime to the ECMP group v through a procedure called routing mapper: it maps L egress ports to q buckets, where L is the size of the ECMP group, and q is the Coprime (Coprime to the size of another ECMP group, which correlates with v). The routing mapper duplicates q/L times for each of the first $q\%L$ entries, and $q/L+1$ times for the remaining entries in the ECMP group. For example, Figure 3.2 shows that the routing mapper transforms the second ECMP group to an MCMP of five entries: three for the first egress port, and two for the other. We also denote the routing mapper for ECMP by *ECMP mapper* to distinguish with the routing mapper for WCMP (describe in § 3.2.2.1).

3.1.2.2 Coprime-based Approach

We describe an algorithm of choosing a Coprime to meet a given error $\widehat{e}$ for an ECMP group. Algorithm 4 shows the procedure: line 4 chooses the smallest integer q which, first, achieves an error $error(v,q)$ bellow $\widehat{e}$, and second, is co-prime to V, where $error(v,q)$ computes the error of applying q to ECMP group v, $is\_coprime(q,q_i)$ returns true if q and Integer q_i, are Coprime; otherwise false, and $q_i \in V$ denotes the Coprime of the ith correlated ECMP group of v; initially, q_i is the size of the ith ECMP group.

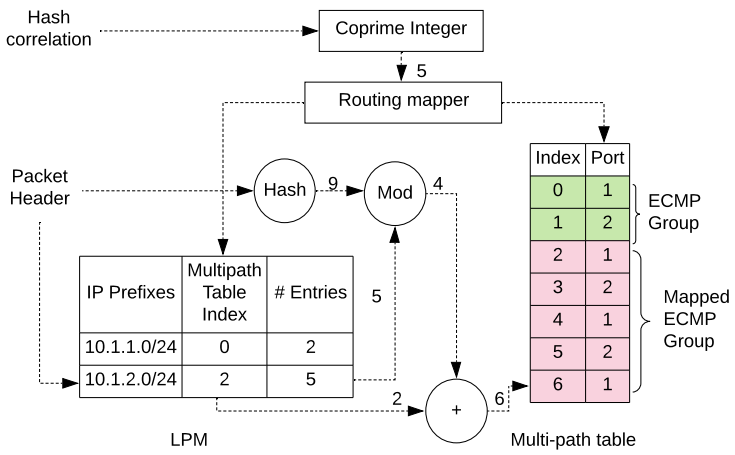


Figure 3.2: The procedure of mitigating the load imbalance caused by hash correlations.

Algorithm 1 Coprime-based Approach

1: **procedure** $selectCoprime(v, V, \widehat{e})$
2: $q = L$
3: **while** $True$ **do**
4: **if** $error(v, q) \leq \widehat{e}$ and $\forall q_i \in V, is\_coprime(q, q_i)$ **then**
5: return q
6: $q + 1$

We employ the coefficient of variance (CV) [86] to quantify the effect ($error(v, q)$ in Algorithm 4 of applying a Coprime: a larger CV indicates higher non-uniformity. The CV is computed as the ratio of the standard deviation to the mean of U, $U = \{u_i | i \in [1, L]\}$, where u_i indicates the number of duplicates for the ith egress port in an ECMP group; u_i is either q/L or $q/L+1$ depending on the routing mapper. For example, a Coprime of 5 leads to $U = [3, 2]$ for the second ECMP group in Figure 3.2. Its standard deviation is 0.5, and the average is 2.5, giving the coefficient of variation as 0.2.

3.2 Network-Wide Correlations

In this section, we describe mitigating hash correlations in a network-wide scale. We present a procedure called Coprime (§ 3.2.1) selector: it trades off between the effect of Coprime on solving

the hash correlation and the switch memory. Later, we extend the Coprime-based approach to handle WCMP (§ 3.2.2).

3.2.1 Coprime Selector

When handling many hash correlations, we need to consider the trade-off between the error of Coprime and the memory capacity. Small Coprime values cannot effectively mitigate hash correlation because the correlation leading to nonuniform distribution of flows over egress interfaces (described in Theorem 2); Larger Coprime integers can reduce the hash correlation, but comes at a cost that the Coprime value, which determines the number of the routing entries, can quickly run out of the table entries in a commodity switch, typically numbering in the small thousands [3]. This issue motivates us to devise a Coprime selector - a mechanism to choose a Coprime for each correlated ECMP group - considering both constraints.

In this subsection, we present two algorithms regarding the Coprime selector: one is to reduce the memory given an error (§ 3.2.1.1); the other is to minimize the error given a fixed memory (§ 3.2.1.2).

3.2.1.1 Memory Reduction

One method to handle many correlations is to choose a Coprime for each ECMP group in an arbitrary order. However, this causes many large Coprimes co-exist on a single switch, which exhausts the multi-path table. For example, Figure 3.3 shows Switch s_1 has two large Coprimes: both are 5, and thus, they occupy ten entries in the table; while each of other switches only has two entries.

To avoid overstripping a single switch, we propose to handle the ECMP groups in an descending order by the memory usages of the switches where the ECMP groups reside. We show the procedure in Algorithm 2 which selects a Coprime for each ECMP group to meet the demand of the error denoted by E. The ECMP groups are aggregated by its destination prefix; for Prefix $dest$, the ECMP groups are sorted according to $M[S[v]]$ (line 3), where v denotes the ECMP group, $S[v]$ is the switch which comprises v, and $M[S[v]]$ is the memory usage of Switch $S[v]$. We assign L

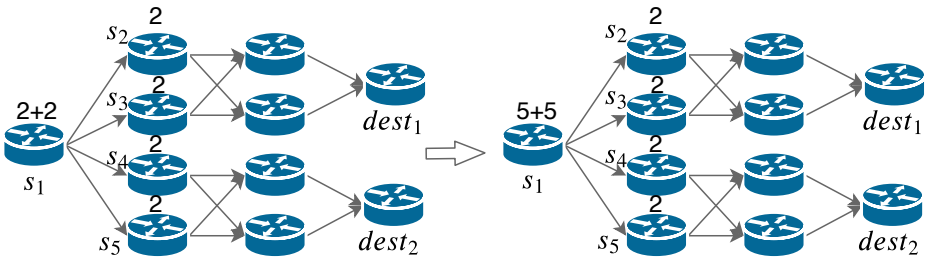


Figure 3.3: An example of coprime selection in an arbitrary order.

to the first ECMP group (line 7) and leverage Algorithm 4 to process other groups (line 9), where L is the number of entries of ECMP group v. For example, Figure 3.4 shows the Coprime selector via Algorithm 2: each of Switches $s_2 - s_5$ has one Coprime (here is 5), and Switch s_1 has two small Coprimes (2), leading to a maximum of five entries instead of ten in Figure 3.3.

Algorithm 2 Coprime Selector for Memory Reduction

```
1: procedure coprimeSelectorMemory(E)
2:     for each dest do
3:         sort ecmp_groups[dest] by M descendingly
4:         for v ∈ ecmp_groups[dest] do
5:             if v is the first item in ecmp_groups[dest] then
6:                 q = L
7:             else
8:                 q = selectCoprime(v, V, E[S[v]])
9:             M[S[v]] += c − L
```

3.2.1.2 Minimize Error

Given a fixed memory, one method to reduce the error is to iteratively increase the upper bound of error - starting from a small value - until the memory usage fits the capacity. However, when one switch reaches its capacity, other switches are still capable to allow more flow rules to reduce the error. Instead of one bound for all, we enable diverse errors for different switches. Algorithm 3 shows our approach: we employ Procedure $coprimeSelectorMemory$ in Algorithm 2 to choose

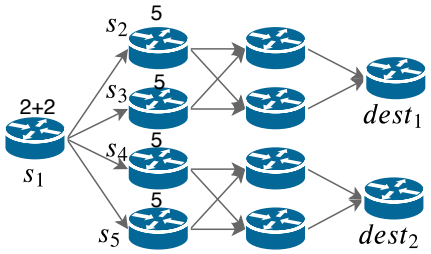


Figure 3.4: An example of Coprime selection to reduce the memory usage.

Coprimes (Line 4) in each iteration; after identifying the bottleneck switch s_m, which uses the maximum memory (line 5), we update $E[s_m]$ by δ_e (line 6), where $E[s_m]$ is the bound of error for the ECMP groups on Switch s_m, and δ_e denotes the updating rate: a pre-defined constant. When the memory usage is below the capacity C, the procedure ends (line 3).

Algorithm 3 Coprime Selector for Error Reduction

```
1: procedure coprimeSelectorError
2:     max_m = C + 1
3:     while max_m > C do
4:         coprimeSelectorMemory(E)
5:         max_m, s_m = max(M), argmax(M)
6:         E[s_m] += δ_e
```

3.2.2 WCMP

In this section, we extend the Coprime-based approach to handle WCMP. We apply the routing mapper to handle WCMP in § 3.2.2.1.

3.2.2.1 WCMP Mapper

One straightforward method of applying the routing mapper to WCMP is to regard a WCMP group as an ECMP group of W logical ports, where W is the sum of the weights. In this method, the routing mapper maps the index of the multi-path table to the logical ports in a round-robin

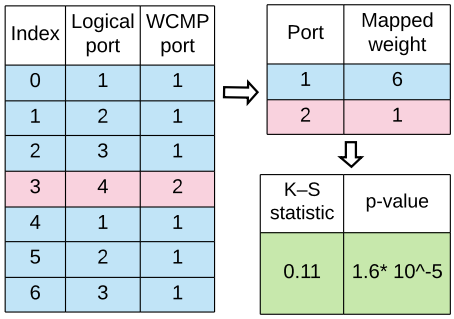
manner. However, the weights after mapping deviate largely from the WCMP weights. For example, Figure 3.5a shows that the mapper duplicates two ports, whose weights are one and three, to fit seven slots: the routing mapper treats the WCMP as an ECMP with four logical ports (three port 1s, and one port 2). The observed weights are $6:1$, which leads to a $K-S$ statistic of 0.11 and a p-value of $1.6 * 10^{-5}$.

To reduce the difference between the observed weights and the desired weights, we describe a method called *WCMP mapper*, which handles the quotient and the remainder of q/W differently. For the quotient, we replicate each WCMP port by $w_i * q/W$ times; for the remainder r ($r = q \ \% \ W$), we duplicate the first r' WCMP ports one more time than others when $r \neq 0$, where $r' = r \ \% \ L$. Therefore, each of the first r' WCMP ports is repeated for $w_i * q/W + r/L + I(r \neq 0)$ times, and each of the remaining ports gets $w_i * q/W + r/L$ duplicates, where $I(r \neq 0)$ denotes the indicator function and returns 1 if $r \neq 0$, otherwise, 0. Note that the WCMP ports should be sorted in an descending order by their weights to make sure that the ports with higher weights are replicated one more time when handling Remainder r. Figure 3.5b shows an example of the WCMP mapper, where the first four entries have the same mappings as Figure 3.5a, and the remaining three entries are mapped to port 1 for twice and port 2 for once. The K-S statistic (here is 0.035) is smaller than Figure 3.5a; and the p-value (0.56) is significantly large to accept that the CDFs with/without Coprime are the same.

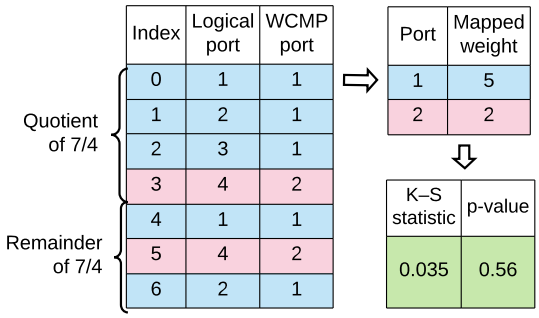
Based on the above observation, we propose Theorem 3 and prove it in Appendix A.4.

Theorem 3. *When handling WCMP, WCMP mapper achieves a smaller $K-S$ statistic than ECMP mapper.*

After applying the routing mapper to handle WCMP, we need to extend the Coprime selector for WCMP. For non-uniform weights, we employ the K-S statistic and the p-value described in Chapter 2. Because the non-uniform weights of WCMP prevents using the coefficient of variance, which is only for uniform distributions. Figure 3.6 shows the examples of both the K-S statistic and the p-value for one WCMP group with weight $3:1$; When the $K-S$ statistic is below 0.07, the p-value is beyond the threshold (here is 0.01), and we accept the hypothesis of two CDFs being

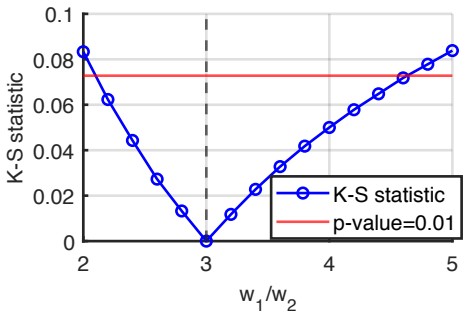


(a) ECMP Mapper for a WCMP group of two ports

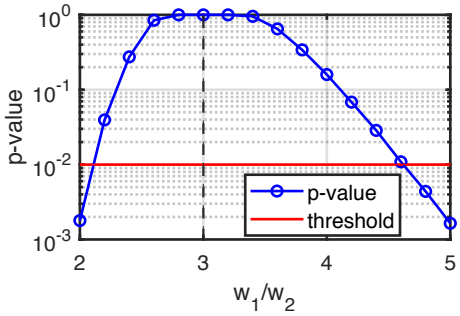


(b) WCMP Mapper for a WCMP group of two ports

Figure 3.5: Examples of the routing mapper for WCMP.



(a) K-S statistic



(b) p-value

Figure 3.6: One example of the K-S statistic and p-value for a WCMP group with weight 3:1.

the same.

Instead of

3.2.2.2 Weight Change

The weights of a WCMP group may change due to the network dynamics, such as routing change, congestions, link failures, etc. Figure 3.7 shows the procedure of handling weight change: after recomputing a Coprime integer via Algorithm 4, the controller sends the forwarding rules from the WCMP mapper to the data plane. Upon receiving new forwarding rules, the data plane

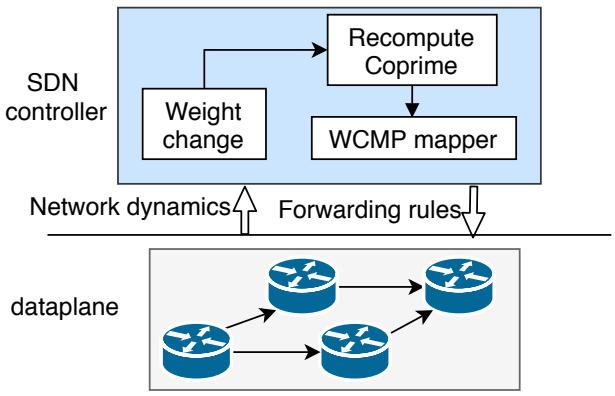


Figure 3.7: The procedure of handling weight change.

substitutes the old rules. Note that generating new weights is beyond the scope of this paper; one can refer to the work [3] for new weights.

We ignore weight change in a small scale when we still accept the null hypothesis that the two CDFs - the one after applying the current Coprime and the one of the new weights - being the same; in other words, the p-value is still beyond the probability threshold after weight change.

3.3 Hash Allocation for Hierarchical Networks

In this section, we describe the hash allocation for hierarchical networks. One intuitive method is round robin, where hash functions are repeatedly assigned to switches. However, this algorithm runs out of the limited hash functions quickly when the switches are beyond the hash functions, leading to hash correlations. Instead, we assign hash functions in a hierarchical way to eliminate the hash correlations.

We can leverage the hierarchical property of the topologies to assign the hash functions: assign one hash function to each layer. For example, Figure 3.8 shows allocating one of three hash functions - H_1, H_2 and H_3 - to each of the three layers for a simple tree-based topology. This ensures no re-use of hash functions along the routing path from the top layer to the bottom.

However, when the traffic does not go from the top layer to the bottom, the per-layer allocation cannot assign one hash function to each layer. For example, Figure 3.9 shows the topology needs

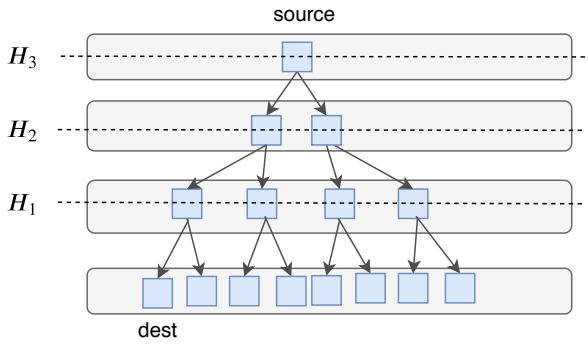


Figure 3.8: An example of per-layer hash allocation for a tree-based topology.

$4 + n$ independent hash functions, because each separation point - where the traffic get separated - needs a hash function. For example, from ToR 1 to ToR 2, the six separation points demand six hash functions: $\widehat{H_1}$, $\widehat{H_2}$, H_1, $\widehat{H_3}$, $\widehat{H_4}, H_2$; from ToR 2 to ToR 3, the traffic also passes through six separation points, leading to six hash functions: $\widehat{H_1}$, $\widehat{H_2}$, H_2, $\widehat{H_3}$, $\widehat{H_4}, H_3$. The number of aggregation blocks in a large topology easily exceeds the limited hash functions.

We observe that upon the recombining of the traffic, the switch can reuse the hash function. We compare the traffic going through a hash function to the light passing through a triangular prism: the light passes through prism and get separated to its component colors; the traffic goes through the hash function and gets forwarded to multiple ports. For example, Figure 3.10 shows that Switch s_2 can still divide the traffic into multiple streams even if it uses the same hash function H_1 as Switch s_1. We use the term *color combining* to denote the recombining of the traffic.

The color combining in hierarchical networks enables switches to reuse the hash functions. For example, Figure 3.11 shows that Switch s_5 (s_6) can reuse $\widehat{H_3}$, where $\widehat{H_3}$ is a hash function. The traffic gets separated at Switch s_1, recombined when it arrives at s_2, s_3 and s_4, and separated again when leaving s_5, leading to the reuse of $\widehat{H_3}$ on s_5 (s_6) without causing hash correlations.

3.4 Evaluation

We conducted simulations under a set of realistic settings - real-network traces, different topologies and hash functions - to evaluate the performance of . The results show we can reduce the errors (CV or K-S statistic) by one magnitude compared with round robin.

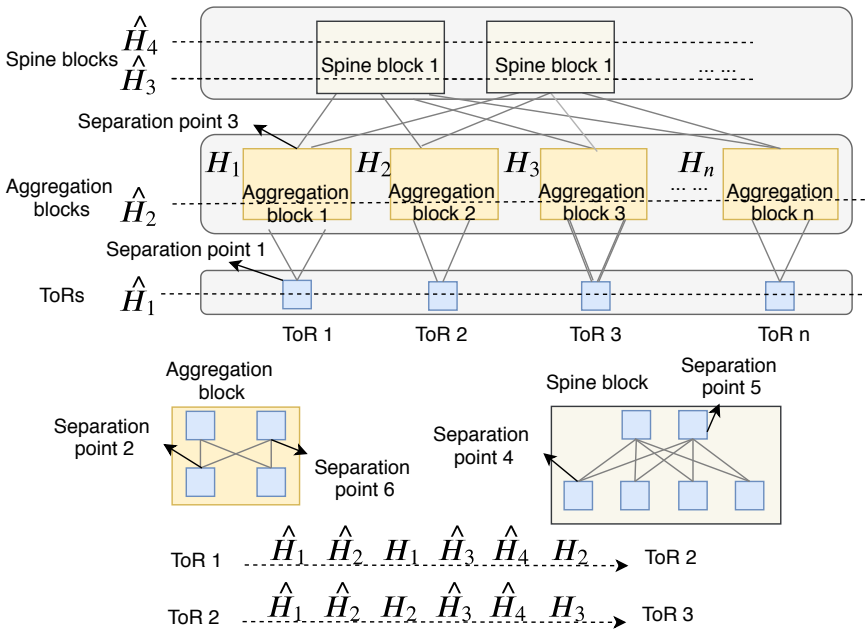


Figure 3.9: The example of no hash reuse along each routing path for a hierarchical network.

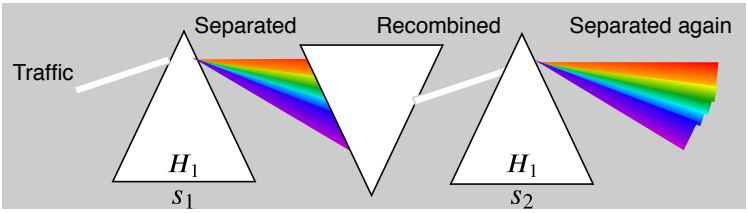


Figure 3.10: Comparing the effect of hash function on traffic to the light passing through triangular prisms.

3.4.1 Experiment Setup

We use CAIDA's passive traces dataset [91] from a high-speed Chicago monitor on a commercial backbone link: each 1-second segment has $475.37k$ packets and $12.91k$ 5-tuple flows on average.

We employ two topologies: the hierarchical topology in Figure 3.11 and six ISP topologies [92] shown in Table 3.1, where $File\_ID$ denotes the file identification number. To ensure a large number of correlations, we use k-shortest ($k = 4$) path routing [93] for the routing.

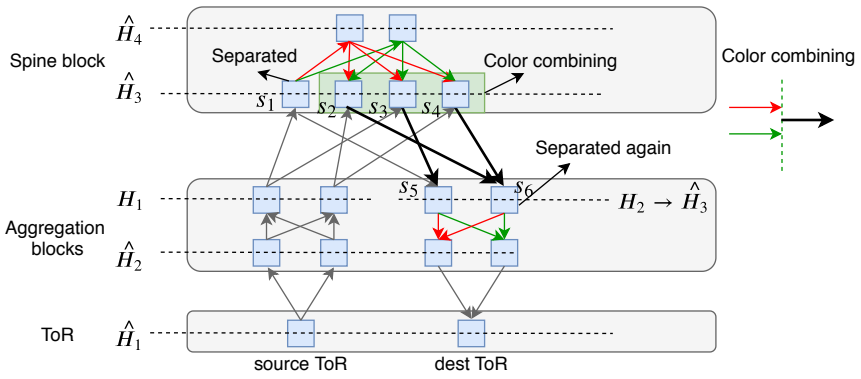


Figure 3.11: An example of color combing.

#File_ID	1	2	3	4	5	6
#nodes	69	96	76	131	122	271
#links	146	153	159	322	371	961

Table 3.1: The ISP topologies.

We use the RTAG7 [17] for ECMP which includes seven hash functions - two $CRC16$s, four XORs and one $CRC32$; and the $CRC32$ can be used as two $CRC16$s: the lower 16 bits and the higher 16 bits. When study the effect of hash functions, we also use other versions of $CRC16$s because we need beyond eight hash functions.

For each simulation, we count the number of flows for each port in each ECMP group and use them to compute the metrics (CV, K-S statistic or p-value).

3.4.2 Performance of Coprime-based approach

3.4.2.1 The Results for One ISP Topology

The Coprime-based approach reduces CVs by about one order of magnitude compared with round robin for an ISP topology ($File\_ID = 1$), as referred in Figure 3.12. We conducted the simulations with Algorithm 3 under the capacity of 225 entries; round robin uses up to 150 entries. Figure 3.12 shows that the CVs of the Coprime-based approach are less than 0.1; while round robin shows higher CVs: almost 20% are beyond 0.4. The Coprime-based chooses Coprime for

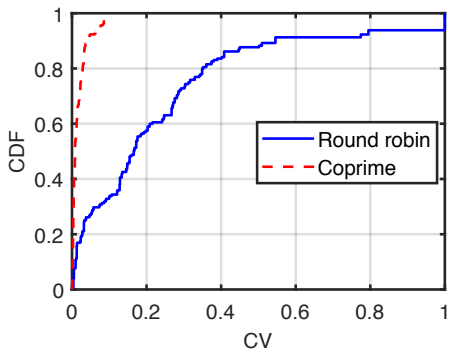


Figure 3.12: CDF plots of coefficient of variation (CV) of ECMP groups for an ISP topology, where CV = (standard deviation) / mean

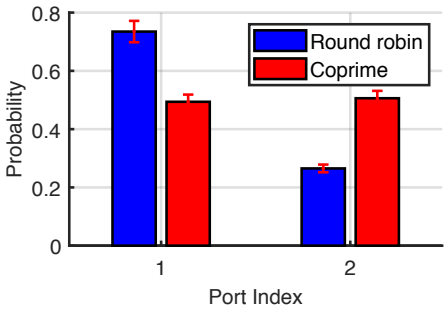


Figure 3.13: The histogram of the number of flows over two ports in one ECMP group.

each correlated pair to mitigate the correlations, and thus, the flow distribution of each ECMP group are more uniform than round robin. For example, the histogram in Figure 3.13 shows that Port 2 bears only 2% more flows than Port 1 for Coprime; while the round robin distributes flows by almost $7:3$.

3.4.2.2 Memory Reduction

From Figure 3.14, we found that Algorithm 2 (labeled by "Min" in Figure 3.14) reduces memory by $15\% - 50\%$ compared with an arbitrary order; we control the upper bound of the CV to be 0.1. For example, Figure 3.15 ($File\_ID = 1$) shows that Algorithm 2 uses less than 180 entries (indicated by the vertical dashed line); while a few switches are with beyond 300 entries for an arbitrary order.

3.4.2.3 Memory Overhead

We studied the trade-off between memory and CV, and found that higher memory causes smaller CVs (Figure 3.16). We also referred that when the memory is beyond 225, the CVs stabilize. We can choose a larger Coprime to reduce the CV when employing a sufficient capacity; the CV can be ignored when the Coprime large enough, which satisfies the conditions in Theorem 2.

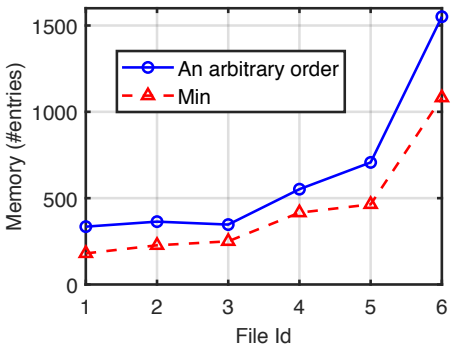


Figure 3.14: The maximum memory usage of the switches.

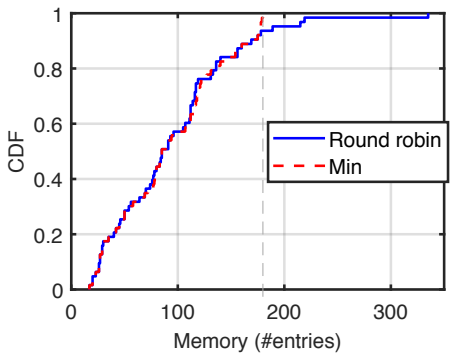


Figure 3.15: The CDF plots of the memory usages of the switches for one ISP topology.

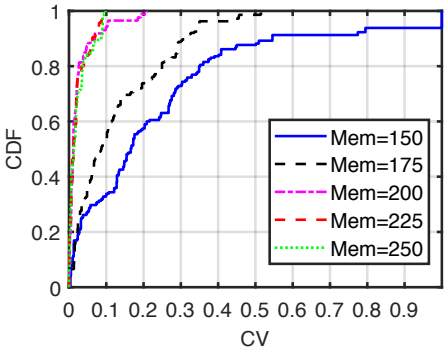


Figure 3.16: The CDF plots of CVs for the ECMP groups in an ISP.

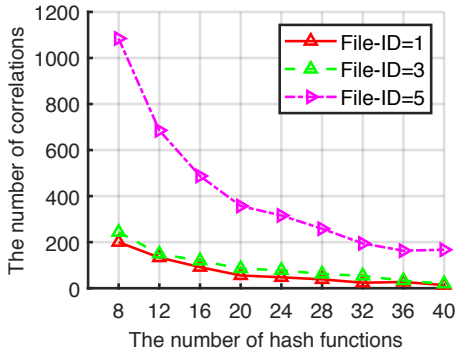


Figure 3.17: The number of correlated pairs of ECMP groups.

3.4.2.4 The Effect of Hash functions

We found that more hash functions causes less correlations (Figure 3.17); the hash functions are allocated via round robin. For example, the correlations can reach 200 with eight hash functions, but the number is only about ten with 40 hash functions (the curve of $File{-}ID = 1$ in Figure 3.17). We also found that a larger topology generates more correlations. For example, Figure 3.17 shows that $File - ID = 5$ has 1000 more correlations than $File - ID = 1$ with eight hash functions.

Additionally, we found that the Coprime-based approach uses less memory to achieve the same CV with more hash functions (Figure 3.18). For example, Figure 3.18 shows that the memory decreases by 31.11% for 40 hash functions compared with eight hash functions.

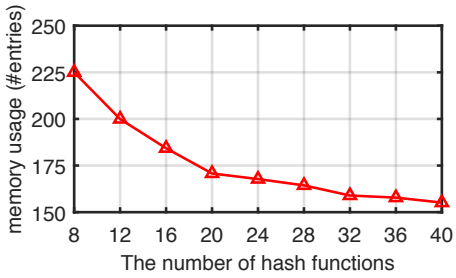


Figure 3.18: The memory usages under multiple number of hash functions.

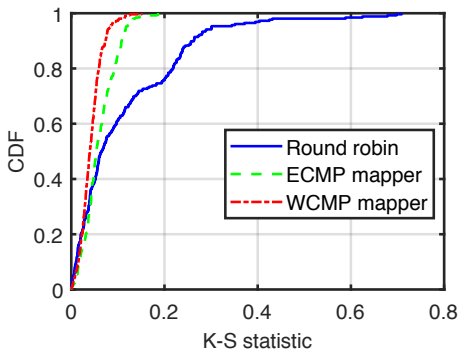


Figure 3.19: The CDF plots of K-S statistic of the WCMP groups in an ISP topology.

3.4.3 The Results for WCMP

Figure3.19 shows that the WCMP mapper reduces the K-S statistic by at least 88% and 54% compared with the round robin and the ECMP mapper, respectively. We conducted the simulation by changing the weights of ECMP groups (of two ports) to $3:1$ in the topology with $File-ID=1$. We also refer from Figure 3.19 that the K-S statistic is below 0.15 for WCMP mapper; while a few values can reach 0.2 for ECMP mapper. WCMP mapper outperforms ECMP mapper is consistent with Threorem 3.

Suppose the probabilistic threshold $T=0.1$ (described in § 3.2.2.1), Figure 3.20 refers that more than 98% WCMP groups accept the null hypothesis of the observed weights being the same with the WCMP weights when using WCMP mapper; while the ratios are about 60%, and 80% for round robin and ECMP mapper. For example, Figure 3.21 compares the WCMP mapper with ECMP mapper for one WCMP group (of weights $3:1$): the WCMP mapper achieves a larger p-value when the Coprime is 7, 11 and 15.

We found that the decrease of the sum of weights reduces the K-S statistic under the same capacity of memory (here is 300). We chose two sets of weights: 1:1 (known as ECMP) and 3:1 for the simulation and the results in Figure 3.22 show about 90%, and 60% WCMP groups have the K-S statistic below 0.05 for the two weight sets, respectively. The reason is that the WCMP group is regarded as an ECMP group with W ports, where W is the sum of weight, and a larger

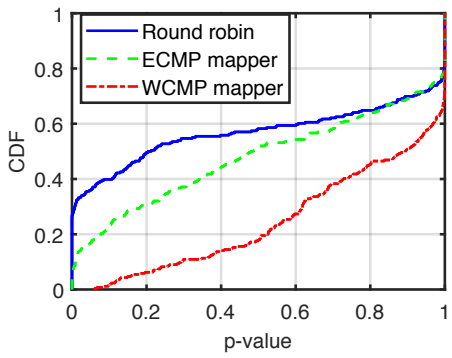


Figure 3.20: The CDF plots of p-values of the WCMP groups.

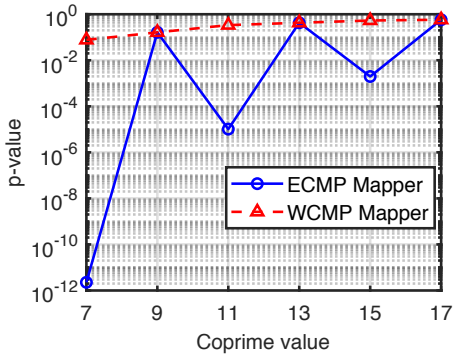


Figure 3.21: The p-values of one WCMP group.

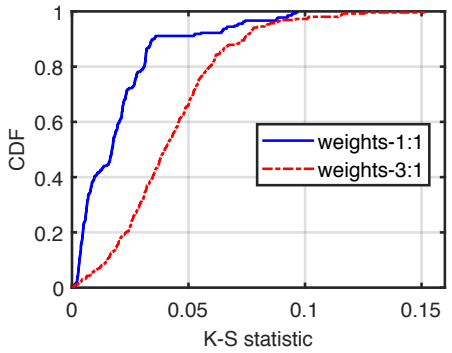


Figure 3.22: The effect of weights on the K-S statistic.

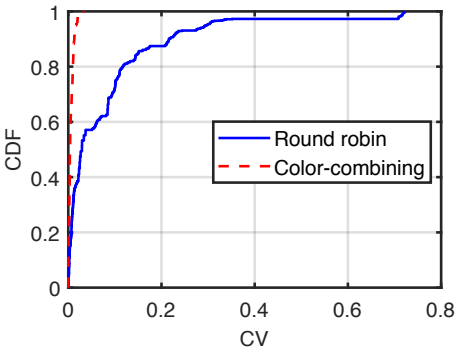


Figure 3.23: CDF plot of coefficient of variation (CV) of ECMP groups for a hierarchical topology.

ECMP size leads to a larger Coprime to achieve the same error (K-S statistic).

3.4.4 The Results of Hierarchical Topology

Color combining reduces the CVs by almost two orders of magnitude than round robin for the hierarchical topology. Figure 3.23 refers that the CVs of color combining are smaller than 0.01, while they can reach 0.72 for round robin. We reuse the hash functions when color combining happens, which eliminates the correlations; and thus, the flows distribute uniformly in each ECMP group.

4. SAMPLING*

Sampling is a critical step for network monitoring, and one example is detecting load imbalance described in Chapter§ 2. In this chapter, we describe the algorithms for sampling, which combines the priority sampling and flow aggregation.

4.1 Priority-Based Aggregation

4.1.1 Preliminaries on Priority Sampling

Priority Sampling m items from a set of $n > m$ weights $\{x_i : i \in [n]\}$ is accomplished as follows. For each item i generate u_i uniformly in $(0, 1]$, and compute its priority $r_i = x_i/u_i$. Retain the (random) top m priority items, and for each such item define the estimate $\widehat{x}_i = \max\{x_i, z\}$, where z is the $(m + 1)^{\text{st}}$ largest priority. For the remaining $n - m$ items define $\widehat{x}_i = 0$. Then for each i, $E[\widehat{x}_i] = x_i$ where the expectation is taken of the distribution of the $\{u_i : i \in [n]\}$. Priority sampling can be implemented as reservoir streams sampling, taking the first m items, then processing the remaining $n-m$ items in turn, provisionally adding each to the reservoir then using the above algorithm to discard one item.

4.1.2 Algorithm Description

We consider a stream of items $\{(k_t, x_t)\}_{t\in T}$ where $T = [|T|] = \{1, 2, \ldots, |T|\} \subset N$. $x_t > 0$ is a **size** and k a **key** that is a member of some keyset K. Let

$$X_{k,t} = \sum_{s\le t, k_s=k} x_s \tag{4.1}$$

denote the total size of items with key k arriving up to time t whose key is k. Let K_t denote the set of unique keys arriving up to and including time t. We aim to construct a fixed size random summary $\{\widehat{X}_{k,t} : k \in \widehat{K}_t\}$ where $\widehat{K}_t \subset K_t$ with $|\widehat{K}_t| \le m$ which provides unbiased estimates over

*Part of this chapter is reprinted with permission from: Stream Aggregation Through Order Sampling, CIKM,{Pages: 909–918, (November)} © ACM, 2017. https://dl.acm.org/doi/10.1145/3132847.3133042.

all of K_t $E[\widehat{X}_{k,t}] = X_{k,t}$ for all $k \in K_t$. Implicitly $\widehat{X}_{k,t} = 0$ for $k \notin \widehat{K}_t$.

To accomplish our goal we extend Priority Sampling to include aggregation over repeated keys. Sampling will be controlled by a family of weights $\{W_{k,t} : k \in \widehat{K}_t\}$. These generalize the usual fixed weights of priority sampling in that they can be both random and time dependent, although within certain constraints that we will specify. The arrival $(k, x) = (k_t, x_t)$ is processed as follows:

1. If the arriving key is in the reservoir, $k \in \widehat{K}_{t-1}$ then we increase $X_{k,t} = X_{k,t-1} + x$, leave the sample keyset unchanged, $\widehat{K}_t = \widehat{K}_{t-1}$, and await the next arrival.

2. If the arriving key is not in the reservoir, $k \notin \widehat{K}_{t-1}$, then we provisionally admit k to the sample set forming $\widehat{K}'_t = \widehat{K}_{t-1} \cup \{k\}$. We initialize $\widehat{X}_{k,t}$ to x, q_k to 1, and generate the random u_k uniformly on $(0, 1]$. Then:

   (a) If $|\widehat{K}| \leq m$ we set $\widehat{K}_t = \widehat{K}'_t$ and await the next arrival.

   (b) Otherwise $|\widehat{K}| > m$, we discard the key

$$k^* = \arg\min_{k' \in \widehat{K}_t} W_{k',t}/u_{k'}$$

   from $\widehat{K}'_t$ and set $z^* = W_{k^*,t}/u_{k^*}$. For each remaining $k' \in \widehat{K}$ we update $q_{k',t} = \min\{q_{k',t-1}, W_{k,t}/z^*\}$ and $\widehat{X}_{k',t} = \widehat{X}_{k',t-1} q_{k,t-1}/q_{k',t}$.

While the description above is convenient for mathematical analysis, we defer a formal specification to Section 4.3, where Algorithms 5 and 6 incorporate optimizations described in Section 4.2 that improve performance relative to a literal implementation of steps (1), (2), (2a), (2b) above.

4.1.3 Unbiased Estimation

We now establish unbiasedness of $\widehat{X}_{k,t}$ when $W_{k,t}$ is the cumulative increase in the size in k since k was last admitted to the sample. For each key k let T_k denote the set of times t at which k was admitted to a full reservoir, i.e.,

$$T_k = \{t : k \neq \widehat{K}_{t-1},\ k \in \widehat{K}_t, |\widehat{K}_{t-1}| = m\} \tag{4.2}$$

When $k \in \widehat{K}_{t-1}$, let $\tau_{k,t} = \max[0, t-1] \cap T_k$ denote the most recent time prior to t at which k was admitted to the reservoir, and for the arriving key k_t we set $\tau_{k_t,t} = t$ prior to admission.

Let $T^0 = \{t : k_t \notin \widehat{K}_{t-1}\} \subseteq T$ denote the times at which the arriving key was not in the current sample. Let $\tau_t = \max[0, t-1] \cap T^0$ denote the most recent time prior to t that an arriving key was not the sample. For an integer interval Y we will use the notation $Y^0 = T^0 \cap Y$. For any $t \in T$ and $k \in \widehat{K}'_t$, u_k was generated at time $\tau_{k,t}$. If k is discarded from $\widehat{K}'_t$, a subsequent arrival of k in an item will have a new independent u_k generated.

Our first version of PBA is governed by the exact weights $W_{k,t}$ that the total size in key k of arrivals since k was most recently admitted to sample, i.e.,

$$W_{k,t} = X_{k,t} - X_{k,\tau_{k,t}-1} = \sum_{s \in [\tau_{k,t}, t] : k_s = k} x_s \tag{4.3}$$

Note that $W_{k,t}$ can be maintained in the sample set by accumulation. For each $t \in T^0$ and $i \in \widehat{K}'_t$ let

$$z_{i,t} = \min_{j \in \widehat{K}'_t \setminus \{i\}} \frac{W_{j,t}}{u_j}. \tag{4.4}$$

and z_s denote the unrestricted minimum $z_s = \min_{j \in \widehat{K}'_t} \frac{W_{j,t}}{u_j}$. The conditions under which $i \in \widehat{K}'_t$ survives sampling are

$$\{i \in \widehat{K}_t\} = \{i \in \widehat{K}'_t\} \cap \{W_{i,t}/u_i > z_{i,t}\} \tag{4.5}$$

As a consequence $z_{i,s} = z_s$ if $i \in \widehat{K}_s$. For $t \in T^0$ define

$$q_{k,t} = \min\{1, \min_{s \in [\tau_{k,t}, t]^0} W_{k,s}/z_s\} \tag{4.6}$$

and

$$Q_{k,t} = \begin{cases} q_{k,t} & \text{if } k = k_t \\ q_{k,t}/q_{k,\tau_t}, & \text{otherwise} \end{cases} \tag{4.7}$$

For $k \in K_t$, define $\widehat{X}_{k,t}$ iteratively by

$$\widehat{X}_{k,t} = \begin{cases} \widehat{X}_{k,t-1} + \delta_{k,k_t} x_t & t \notin T^0 \\ (\widehat{X}_{k,t-1} + \delta_{k,k_t} x_t)/Q_{k,t} & t \in T^0,\, k \in \widehat{K}_t \\ 0, & \text{otherwise} \end{cases} \tag{4.8}$$

where $\delta_{i,j} = 1$ if $i = j$ and 0 otherwise. The proof of the unbiasedness of $\widehat{X}_{k,t}$ is deferred to Section 4.4.

Theorem 4. *$\widehat{X}_{k,t}$ is unbiased: $E[\widehat{X}_{k,t}] = X_{k,t}$.*

We have also proved that replacing $W_{k,t}$ with an affine function of the current estimator $\widehat{X}_{k,t}$ also yields an unbiased estimator at the next time slot. This has the utility of reducing memory usage since a separate $W_{k,t}$ per aggregate is not needed. However, we also found in experiments that this estimator was not so accurate. For both variants of the estimator, we can derive unbiased estimators of $\mathrm{Var}(\widehat{X}_{k,t})$, These can be used to establish confidence intervals for the estimates. Due to space limitations we omit further details on all the results summarized in this paragraph.

4.2 Optimizations

4.2.1 Deferred Update

For each i, $q_{i,t}$ is computed as the minimum over s of $W_{i,s}/z_s$. As it stands, this is more complex that the corresponding computation in Priority Sampling for fixed weights W_i, where W_i/z_t^* is computed once for each arrival. By comparison, it appears that in principle, we must update $q_{i,t}$ for all $i \in K_t$ at each $t \in T^0$. We now establish that for each key k, $q_{k,t}$ needs only be updated when an item with key k arrival, i.e.., at t for which $k_t = k$. Updates for times t in T^0 for which $k_t \neq k_t$ can be deferred until the first time $t' > t$ for which $k_{t'} = k$, or whenever an estimate of $\widehat{X}_{k,t}$ needs to be computed. This property is due to the constancy of the fixed weights between updates and the monotonicity of the sequence z_t^*. For $t \in T^0$ let $z_t^* = \max_{s \in [0,t]^0}\{z_s\}$. Let d_t denote the key that is discarded from $\widehat{K}'_{t-1}$ at time $t \in T^0$, i.e., $\{d_t\} = \widehat{K}'_{t-1} \setminus \widehat{K}_t$. When

$t \in T^0$ and $i \in \widehat{K}_t$ define $q^*_{k,t}$ recursively by

$$q^*_{i,t} = \min\{q^*_{i,\tau_t}, W_{i,t}/z^*_t\} \tag{4.9}$$

unless $k_t = i$ in which case $q^*_{i,t} = \min\{1, W_{i,t}/z^*_t\}$. The proof of the following result is detailed in Section 4.4.

Theorem 5. *(i) $t \in T$ implies $z^*_t = z_t$.*

(ii) $q_{i,t} = q^*_{i,t}$ for all t where these are defined..

Theorem 5 enables computational speedup as compared with updating each key probability at each $t \in T^0$. Since z^*_t is monotonic in t, we only need to update the probabilities $q_{i,t}$ for links i whose weight increases after admitting a key at time t. Likewise, we perform a final update at the end of the stream, or at any intermediate time when an estimate is required.

4.2.2 Pre-aggregation

Pre-aggregation entails summing weights over consecutive instances of the same key before passing to PBA. Pre-aggregation saves on computational complexity of updating priorities, instead of updating a single counter. This also results in an unbiased estimator whose variance at least as large as PBA.

4.2.3 Priority-Based Adaptive Sample and Hold

Sample and Hold [42] with a fixed parameter is a simple method to preferentially accumulate large aggregates. However, in this form, Sample and Hold cannot adapt to variable load or a fixed buffer. Adaptive Sample and Hold (ASH) [42, 64] using resamples to selectively discard from the reservoir. We propose to retain the advantages of Sample and Hold within an adaptive framework by using it as a front end to PBA, with its sampling parameters adapted directly from the time-varying threshold of PBA.

We call this coupled system Priority-Based Adaptive Sample and Hold (PBASH). When an arriving item (k, x) finds its key k is not in the current sample $\widehat{K}_t$, the item is sampled with proba-

bility $p_t(x) = \min\{1, w/z_t^*\}$ where the current threshold z_t^* provides scale that takes into account the current retention probabilities for items in the reservoir. In order to preserve unbiasedness, the weight of any such item is normalized to $x/p_t(x) = \max\{x, z_t^*\}$. Subsequent items in the aggregate that find their key already stored are selected with probability 1 and their sizes passed to PBA without any such initial normalization. Unbiasedness of the final estimate then follows from the chain rule for condition expectations (see e.g [94]) since PBA provides an unbiased estimate of the unbiased estimate produced by the ASH stage. We note that ASH pre-sampler uses the PBA data structure to determine whether a key is in storage. All key insertion and deletions are handled by PBA component. We specify PBASH formally in Algorithm 6 of Section 4.3

4.2.4 Trading Bias for MSE: Error Filtering

Unbiased estimation of aggregates is effective for larger aggregates since averaging over estimated contributions to the aggregate reduces error. Smaller aggregates do not enjoy this property, motivating supplementary approaches to reduce error. A strawman approach is to count the number of estimates terms in the aggregate, and use this value as a criterion to adjust or exclude small aggregates. Another strawman approach filters based on estimated variance, excluding aggregates with a high estimated relative variance. The disadvantage of these approaches is that they require another counter. Instead, we are drawn to find mechanisms to accomplish this goal that do not require extra storage.

Our approach is quite simple: we ignore the contribution of the first item of every newly instantiated aggregate to its estimate, although in all other respects, sampling proceeds as before. Thus, while the renormalized item weight does not contribute to the aggregate estimator $\widehat{X}_k$, the unnormalized item weight does contribute to W_k used in Theorem 4. The resulting estimator is clearly biased since it underestimates the true aggregate on average, but reduces as the experiments reported in Section 4.5 will show.

Abbrev.	Description	Reference
PBA	Priority-Based Aggregation	Alg. 5
PBA-EF	PBA w/ Error Filtering	Alg. 5
PBASH	Priority-Based Adaptive Sample & Hold	Alg. 6
PBASH-EF	PBASH w/ Error Filtering	Alg. 6
ASH	Adaptive Sample & Hold	[64, 63]
SH	Sample & Hold (Non-Adaptive)	[42]

Table 4.1: Nomenclature for Algorithms. Reprinted with permission from [1].

4.3 Algorithms and Implementation

4.3.1 Algorithm Details

The family of PBA algorithms using true weights is described in Algorithm 5. (Our nomenclature for the Algorithms in given in Table 4.1). Pre-aggregation over consecutive items bearing the same key (see Section 4.2.2) takes place in lines 2–8. The pre-aggregates are passed to the main loop in line 9. In the main loop, deferred update (Section 4.2.1 takes place before aggregation to an existing key in lines 14–15. Otherwise, a new key entry is instantiated in lines 17–19. With error filtering (Section 4.2.4), the first update of the estimate is omitted at line 18. When a new key arrives at the full reservoir, selection of a key for discard takes place in lines 22-24. In our implementation, we break this step down further. The aggregates are maintained in a priority queue implemented as a heap. An incoming new key is rejected if its priority is less than the current minimum priority; see Section 4.3.3. After the stream has been processed, remaining deferred updates to the estimates occur in lines 10–11. This step could also be performed for any or all aggregates in response to a query. Algorithm 6 describes the modifications to the main loop for PBASH. A new pre-aggregate key is instantiated only if it passes the Sample and Hold admission test at line (7).

4.3.2 Data Management & Implementation Details

In common with other stream aggregation schemes for (key, value) pairs, PBA requires efficient access to the aggregate corresponding to the incoming key k. Hash-tables provide an efficient

Algorithm 4 Coprime-based Approach

procedure $selectCoprime(v, V, \widehat{e})$
    $q = L$
    while $True$ **do**
        if $error(v, q) \leq \widehat{e}$ and $\forall q_i \in V, is\_coprime(q, q_i)$ **then**
            return q
        $q + 1$

Algorithm 5 PBA: Priority-Based Aggregation w/ Optional Error Filtering

procedure PSHTRUE(m)
    $K = \emptyset$; $z^* = 0$; $k_{\text{old}} =$ first key k
    while (new keyed weight $(k_{\text{new}}, x_{\text{new}})$) **do**
        if ($k_{\text{new}} = k_{\text{old}}$) **then**
            x_{tot} += x_{new}
        else
            mainloop($k_{\text{old}}, x_{\text{tot}}$)
            $k_{\text{old}} = k_{\text{new}}$; $x_{\text{tot}} = x_{\text{new}}$
    mainloop($k_{\text{new}}, x_{\text{tot}}$)
    for $(k' \in K)$ **do**
        update(k', z^*)

procedure MAINLOOP(k, x)
    if ($k \in K$) **then**
        updatek, z^*
        $a(k)$ += x; $w(k)$ += x
        break
    $K = K \cup \{k\}$; $w(k) = x$; $q(k) = 1$
    $a(k) = x$; *Omit if Error Filter
    generate $u(k)$ uniformly in $(0, 1]$
    if ($|K| \leq m$) **then**
        break
    $k^* = \arg\min_{k' \in K}\{w(k')/u(k')\}$
    $z^* = \max\{z^*, w(k^*)/u(k^*)\}$
    $K = K \setminus \{k^*\}$;
    Delete $a(k^*)$, $u(k^*)$, $q(k^*)$, $w(k^*)$

procedure UPDATE($\tilde{k}, \tilde{z}$)
    $a(\tilde{k}) = a(\tilde{k}) * q(\tilde{k})$
    $q(\tilde{k}) = \min\{q(\tilde{k}), w(\tilde{k})/\tilde{z}\}$
    $a(\tilde{k}) = a(\tilde{k})/q(\tilde{k})$

Algorithm 6 Priority-Based Adaptive Sample and Hold PBASH w/ Optional Error Filtering; mainloop only

```
1:  procedure MAINLOOP(k, x)
2:      if (k ∈ K) then
3:          updatek, z*
4:          a(k) += x; w(k) += x
5:          break
6:      Generate r uniformly in (0, 1]
7:      if r < min(1, x/z*) then
8:          K = K ∪ {k}
9:          a(k) = max(x, z*)*Omit if Error Filter
10:         w(k) = x
11:         q(k) = 1
12:         generate u(k) uniformly in (0, 1]
13:     if (|K| ≤ m) then
14:         break
15:     k* = arg min_{k'∈K}{w(k')/u(k')}
16:     z* = max{z*, w(k*)/u(k*)}
17:     K = K \ {k*};
18:     Delete a(k*), u(k*), q(k*), w(k*)
```

means to achieve this, with the hash $h(k)$ of the key k referencing a location where the aggregate, or in general its unbiased estimator is maintained. PBA also maintains priorities as a priority queue. We implement this as a heap. The question then arises as to how to efficiently combine the heap and hash aspects of the aggregate store.

We manage this with a combined structure that we call a HashHeap. This comprises two components. The first is a hash table that maps a key k to a pointer $\pi(k)$ into the second component. The second component is a min-heap that maintains an entry (k, w, u, a, q) for each aggregate in storage, where k is the key, u is the uniform random variable associated with k, w the current incremented weight since last admission, a the current unbiased estimate, and q the current sampling probability. The heap is ordered by the priority $r = w/u$ which is computed as required, rather than be maintained. The heap is implemented in an array so that parent and child offsets can be computed from the current offset of a key in the standard way.

Collision Resolution. In our design, keys are maintained in the heap, not in the hash. Collision

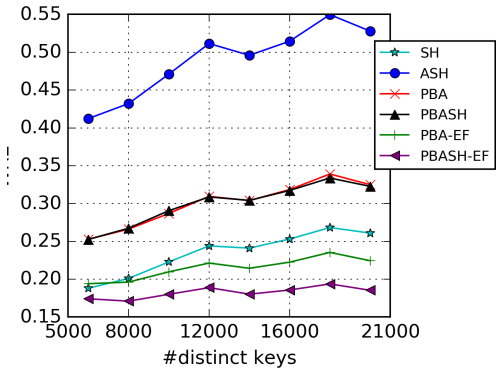


Figure 4.1: Weighted relative error over all keys as a function of distinct key count in reservoir size $m = 1,000$. Reprinted with permission from [1].

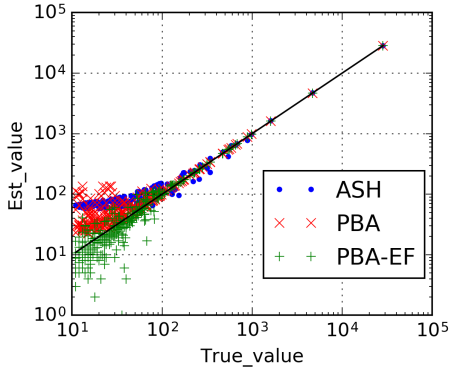


Figure 4.2: Scatter plot of estimate vs. true aggregates for 10^4 distinct keys sampled into reservoir size $m = 500$. Reprinted with permission from [1].

identification and resolution is performed by following a key k to its position $\pi(k)$ in the heap. We illustrate for key insertion using linear probing, which has been found to be extremely efficient for suitable hash functions [95]. Let h denote the hash function. Suppose key k is to be accessed. To find the offset of key k in the heap we probe the pointer hash table from $h(k)$ until we find the pointer π whose image in the hash table is k. Probing to a vacant slot in the hash table indicates the key is not in the heap. For insertion, the offset of the required location in the heap is stored in the vacant slot in the hash table. Our approach is similar to one in [96], the difference being that in that work they key is maintained in the hash table, while each heap entry maintains a pointer back to is a corresponding hash entry. Our approach avoids storage for this second pointer, instead of computing it as needed from the key maintained in the heap.

4.3.3 Computational and Storage Costs

Aggregation to an existing key is $O(1)$ average. All aggregation operations for a key k are increasing its weight w and hence also for its priority. Aggregation requires realignment of the heap, which is performed by bubbling down. i.e. swapping an element with its smallest priority child until it no longer has a larger priority than the child. The pointer offsets of the children are computed from the key k as outlined above. The average cost for aggregation operation is $O(1)$. For simplic-

ity, we assume a perfectly balanced tree of depth h and that the key to be aggregated is uniformly distributed in the heap. Then the average bubble down cost is no worse than $\sum_{\ell=0}^{h} 2^{\ell-h}(h-\ell) \leq 2$.

Rejection of New Keys is $O(1)$ worst case. When an arriving item (k, x) is not present in reservoir, its priority is computed and compared with the lowest priority item in the heap. Access to this item is $O(1)$. If arriving item has lower priority it is discarded. The estimates of the remaining items must be updated, but as established in Section 4.2.1, each update for a given key can be deferred until the next arrival bearing that key.

Insertion/eviction for New Key is $O(\log m)$ worst case. If the arriving item has higher priority than the root item, the later is discarded, the new item inserted at the root, then bubble down to its correct position in the heap. This has worst case cost $O(\log m)$ for a reservoir of size m.

Retrieval is $O(1)$ per aggregate. Any aggregate must undergo a final deferred update prior to retrieval, incurring an $O(1)$ cost.

Storage Costs. *Final Storage.* PBA, PBASH and ASH all have the same final storage cost, requiring a (key, estimate) pair for all stored aggregated. *Working Storage:* PBA and PBASH are most costly for working storage, requiring additional space per item for q, w and the HashHeap pointer. The quasirandom number u can be computed on demand by hashing. These are maintained during stream aggregation, but discarded at the end.

4.4 Proofs of the Theorems

Proof of Theorem 4. For each k we proceed by induction on $t \geq s_k = \min\{s : k_s = k\}$ and establish that

$$E[\widehat{X}_{k,t}|\widehat{X}_{k,t-1}, C] - X_{k,t} = \widehat{X}_{k,t-1} - X_{k,t-1} \tag{4.10}$$

for all members C of a covering partition (i.e., a set of disjoint events whose union is identically true). Since $\widehat{X}_{k,s_k-1} = X_{k,s_k-1} = 0$ we conclude that $E[\widehat{X}_{k,t}] = X_{k,t}$.

For $s_k \leq s \leq s'$ let $A_k(s) = \{k \notin \widehat{K}_{s-1}\}$ (note $A_k(s_k)$ is identically true), let $B_k(s, s')$ denote the event $\{k \in \widehat{K}_s \ldots, \widehat{K}_{s'}\}$, i.e., that k is in sample at all times in $[s, s']$. Then for each $t \geq s_k$ the collection of events formed by $\{A_k(s)B_k(s, t-1) : \ s \in [s_k, t-1]\}$, and $A_k(t)$ is a covering

partition.

(i) *Conditioning on* $A_k(t)$ On $A_k(t)$, $k_t \neq k$ implies $\widehat{X}_{k,t} = \widehat{X}_{k,t-1} = 0 = X_{k,t} - X_{k,t-1}$. On the other hand $k_t = k$ implies $t \in T^0$. Further conditioning on $z_{k,t} = \min_{j \in \widehat{K}_{j,t-1}} W_{j,t-1}/u_j$ then (4.8) tells us that

$$P[k \in \widehat{K}_t | A_k(t), z_{k,t}] = P[W_{k,t}/u_k \leq z_{k,t}] = q_{k,t} \tag{4.11}$$

and hence

$$E[\widehat{X}_{k,t} | X_{k,t-1}, A_k(t), z_{k,t}] = \widehat{X}_{k,t-1} + X_{k,t} - X_{k,t-1} \tag{4.12}$$

regardless of $z_{k,t}$.

(ii) *Conditioning on* $A_k(s)B_k(s, t-1)$ *any* $s \in [s_k, t-1]$. Under this condition $k \in \widehat{K}_{t-1}$ and if furthermore $k_t \in \widehat{K}_{t-1}$ then $t \notin T^0$ and the first line in (4.8) holds. Suppose instead $k_t \notin K_{t-1}$ so that $t \in T^0$. Let $\mathcal{Z}_k(t, s) = \{z_{k,r} : r \in [s, t]^0\}$. Observing that

$$P[B_k(t, s) | A_k(s), \mathcal{Z}_k(t, s)] = P[\cap_{r \in [s,t]^0} \{z_{k,r} \leq \frac{W_{k,r}}{u_k}\}] = q_{k,t}$$

then

$$\begin{aligned} & P[k \in \widehat{K}_t | B_k(t-1, s) A_k(s), \mathcal{Z}_k(t, s)] \\ = \; & \frac{P[B_k(t, s) | A_k(s), \mathcal{Z}_k(t, s)]}{P[B_k(t-1, s) | A_k(s), \mathcal{Z}_k(t-1, s)]} = \frac{q_{k,t}}{q_{k,\tau_t}} = Q_{k,t} \end{aligned} \tag{4.13}$$

and hence

$$E[\widehat{X}_{k,t} | \widehat{X}_{k,t-1}, A_k(s), \mathcal{Z}_k(t, s)] = \widehat{X}_{k,t-1} \tag{4.14}$$

independently of the conditions on the LHS of (4.14). AS noted above, $k \in \widehat{K}_{t-1}$ on $B(t-1, s)$ hence $X_{k,t} = X_{k,t-1}$ and we recover (4.10). Since we now established (4.10) over all members C of a covering partition, the proof is complete. □

Proof of Theorem 5. $t \in T$ means the arriving $k_t \neq d_t$ is admitted to the reservoir and hence

$$z_t = \frac{W_{d_t,t}}{u_{d_t}} \geq \frac{W_{d_t,s}}{u_{d_t}} \geq z_s \tag{4.15}$$

for all $s \in [\tau_{d_t,t}, t]^0$. The first inequality follows because $W_{d_t,s}$ is nondecreasing on the interval $[\tau_{d_t,t}, t]^0$. The second inequality follows because the key d_t survives selection throughout $[\tau_{d_t,t}, t]^0$ and hence its priority cannot be lower than the threshold z_s for any s in that interval. Since d_t was admitted at $\tau_{d_t,t}$, then $d_{\tau_{d_t,t}} \neq d_t$ and hence we apply the argument back recursively to the first sampling time $m+1$. This establishes $z_t \geq z_t^*$ and hence $z_t = z_t^*$.

(ii) i is admitted to $\widehat{K}_t$ if $t \in T$ with $i = k_t \neq d_t$ and hence by (i), $q_{i,t} = \min\{1, W_{i,t}/z_t\} = \min\{1, w_{i,i}/z_t^*\} = q_{i,t}^*$. We establish the general case by induction. Assume $t \in T^0$ and $q_{i,s} = q_{i,s}^*$ for all $s \in [\tau_{i,t}, \tau_t]^0$, and consider first the case that $z_t > z_{\tau_t}^* t$. Then $z_t^* = z_t$ hence $q_{i,t}^* = q_{i,t}$. If instead $z_t \leq z_{\tau_t}^*$ then $z_{\tau_t}^* = z_t^*$ and

$$\frac{W_{i,t}}{z_t} \geq \frac{W_{i,t}}{z_t^*} \geq \frac{W_{i,\tau_t}}{z_t^*} = \frac{W_{i,\tau_t}}{z_t^*} \tag{4.16}$$

Thus we can replace z_t by z_t^* but use of either leaves the iterated value unchanged, since by the induction hypothesis, both are greater than $q_{i,\tau_t} \leq W_{i,\tau_t}/z_{i,\tau_t}^*$ □

4.5 Evaluation

This section comprises a performance evaluation for PBA and PBASH for accuracy and space and time complexity. We used both synthetic trace with features mimicking observed statistical behavior of network traffic, and real-word network traces from measured network denial of service attacks. These traces are chosen to represent dynamic network traffic, and serve to stress-test the summarization algorithms in their ability to adapt to dynamic conditions. The evaluation represents measurement of network traffic over short time scales (at the time scale of seconds or shorter) that are if increasing interest for use in fine-scale traffic management in data center networks [36].

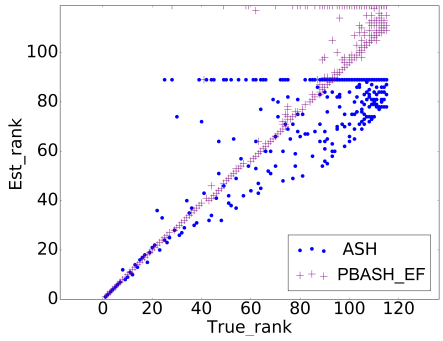


Figure 4.3: Scatter of Estimated, Actual distinct ranks, PBASH and ASH. 5% sampling; data as Figure 4.2. Reprinted with permission from [1].

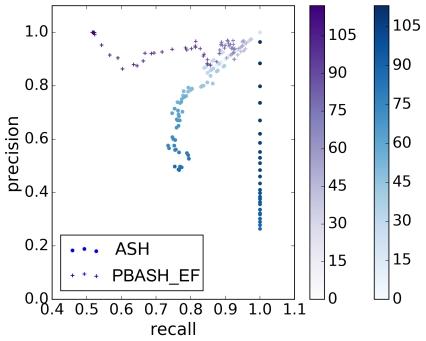


Figure 4.4: Scatter of Prec(R), Recall(R) for distinct ranks, rank R on colormap. 5% sampling; data as Figure 4.2. Reprinted with permission from [1].

4.5.1 Traces and Evaluation Metrics

Trace Data and Platform. The simulations ran on a 64-bit desktop of 8 cores with i7-4790 CPU running at 3.60GHz, each trial taking several seconds to tens of seconds.

Trace 1: Synthetic Trace. This trace was generated first by specifying a key set ranging in size from 6×10^3 to 2×10^4, and then for each key generating a set of unit weighted items whose number is drawn independently from a Pareto distribution with parameter 1.2. The items are presented in random order. This trace is motivated by the observed heavy-tailed distribution of packets per flow aggregate in network traffic [97].

Trace 2: Network Trace with Distributed Denial of Service Attack (DDoS). This trace is used to emulate the effect of network flooding with small packets The traces is a 1-second CAIDA trace with 4.7×10^5 packets and 62299 distinct tuples (srcIP, dstIP, srcPort, dstPor, protocol) randomly mixed by 1-second DDoS traces [91] with packet sending rate from 1.6×10^4 to 6.0×10^6 packets per second and distinct tuples from 6.4×10^3 to 4.5×10^5. The average size of one packet in the CAIDA trace is 495.5 Bytes and that of the DDoS trace is 65.5 Bytes.

Trace 3: Dynamic Network Trace. The trace adds noise to a 15-second CAIDA trace. For each second, let the total byte volume be V, we generate a random probability $p \in (0, 1)$, and pV noise

from another CAIDA trace is added to the original 1-second trace.

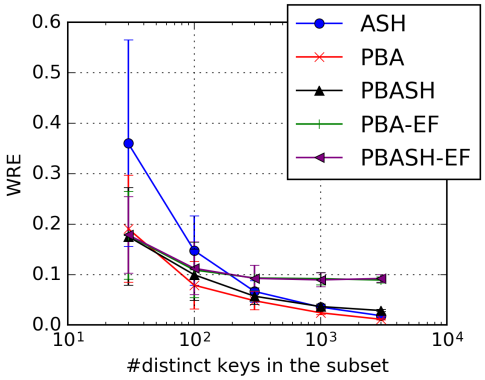


Figure 4.5: WRE as a function of subpopulation size over 100 trials for 10^4 distinct keys sampled into reservoir size $m = 500$. Reprinted with permission from [1].

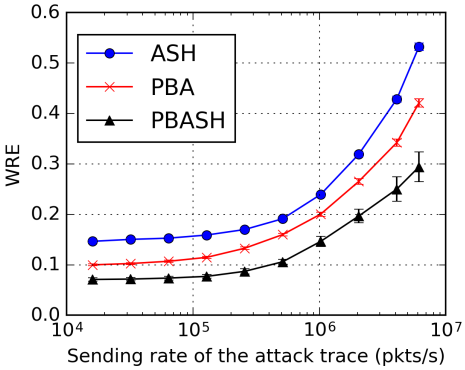


Figure 4.6: WRE for mixed DDos traces at varying packet sending rates, and reservoir size $5,000$. Reprinted with permission from [1]

Evaluation Metrics. The following metrics are measured against reservoir size, set as an independent variable, averaged over 100 trials. For each trial, we randomize the order of the items in the traces. In addition, we randomly regenerate *Trace 1* for each trial.

Execution time: This is the average time per packet over a trace

Subpopulation Accuracy: Our accuracy metric is the Weighted Relative Error (WRE), which we apply in two forms. The first is the average $\sum_k |\widehat{X}_k - X_k| / \sum_k X_k$ where the sum runs over all distinct keys k. To evaluate accuracy for subpopulation queries we use a similar metric $\sum_S |\widehat{X}(S) - X(S)| / \sum_S X_S$ where $X(S) = \sum_{k \in S} x_k$ is the subset sum over a keyset S, and the sum runs over randomly chosen keysets $S \subset K$ of a given size t.

Ranking Accuracy: To compute accuracy for top-R rank queries, we first rank distinct values of true and estimated aggregates, the latter rounded to reduce noise. Let $\widehat{N}(R)$ (respectively) and $N(R)$ denote the set of keys with true (respectively estimated) distinct rank $\leq R$. Then for a top-R

rank query, the precision and recall are

$$\text{Prec}(R) = \frac{|N(R) \cap \widehat{N}(R)|}{\widehat{N}(R)} \text{ and } \text{Rec}(R) = \frac{|N(R) \cap \widehat{N}(R)|}{N(R)} \tag{4.17}$$

4.5.2 Accuracy Comparisons

Figure 4.1 illustrates error metrics for PBA, PBASH, and ASH in a reservoir of size $m = 1,000$ processing items from the synthetic Trace 1. The number of distinct keys varies from 6,000 to 20,000, representing a key sampling rate ranging from 17% down to 5%, WRE was reduced, relative to ASH, by about 40% for PBA and PBASH, by 53–57% for PBA-EF, and by 58–65% for PBASH-EF. As shown, PBASH and PBASH-EF are able to achieve lower WRE than a best-case (non-adaptive) Sample and Hold (SH) in which the sampling rate chosen so as minimize WRE.

To better understand the difference in error between PBA, PBA-EF and ASH, we drill down within an individual experiment. Figure 4.2 is a scatter plot of estimated vs. true aggregate for the two methods for a synthetic trace containing 10^4 distinct keys sampled into a reservoir of size 500, i.e., a key sampling rate of 5%. The figure shows how PBA improves estimation accuracy for smaller weight keys, ASH having a larger additive error (note the logarithmic vertical axis). As expected, PBA-EF further reduces the estimation error for small aggregates, typically underestimating the true value.

Rank Estimation. We evaluate rank estimation performance, focusing on algorithms involving Error Filtering since rankings should be less sensitive to bias than variability. Figure 4.3 shows a scatter plot of estimates vs. actual distinct ranks at 5% sampling for PBASH-EF and ASH. Although both perform well for low ranks (larger aggregates), we observe increasing rank noise for ASH in mid to low ranks. The horizontal clusters in each case correspond to aggregates not sampled; there are noticeably more of these of lower true rank for ASH than PBA-EF. Figure 4.4 shows precision and recall for top-R rank queries. Precision is noticeably better for PBASH-EF, particularly for middle ranks.

Subpopulation Weight Estimation. Figure 4.5 shows WRE for subpopulations over 100 random

selected subpopulations as a function of subpopulation size. For small subpopulations up to size 100, PBA and derived methods provide up to about a 60% reduction in WRE relative to ASH. For larger subpopulations, the difference between the unbiased methods (PBA, PBASH, ASH) approach the same behavior as variance difference becomes less important due to averaging, while the bias of the error filtering methods persist.

Network dynamics : We study the effect in accuracy on an emulated DDoS attack with Trace 2. Figure 4.6 shows the effect on WRE as the DDoS traffic rate increases, in a reservoir of size 5,000. The number of distinct keys increases in proportion to the attack traffic rate, with legitimate traffic representing a smaller proportion of the total. PBA and PBASH achieve lower error than ASH, even as errors for all methods increase, and PBASH-EF (not shown) achieves 60% reduction in error compared with ASH. Figure 4.7 shows a time series of WRE for the dynamic traffic of Trace 3, samples taken over successive 250ms windows in a reservoir size 5,000. PBA and PBASH have smaller fluctuations in WRE in response to the dynamics than ASH achieving similar reduction as before.

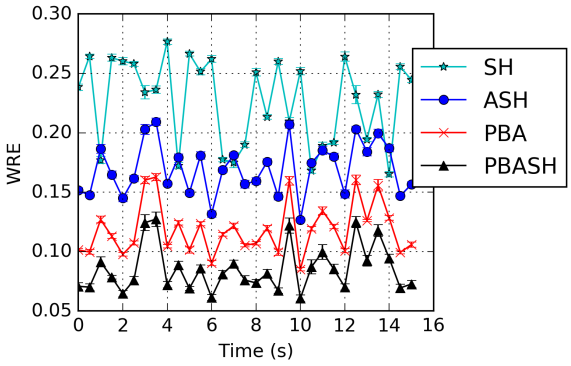


Figure 4.7: The impact of traffic dynamics by adding random noise when the reservoir size is $5,000$. Reprinted with permission from [1].

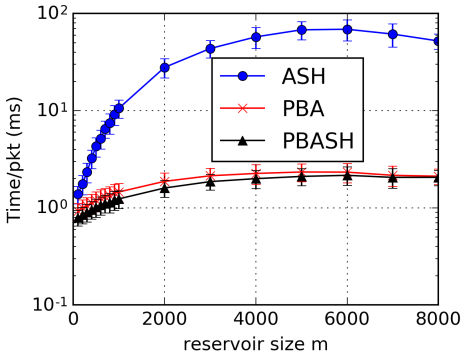


Figure 4.8: The time complexity compared to ASH with varying reservoir sizes and 10^4 distinct keys. Reprinted with permission from [1].

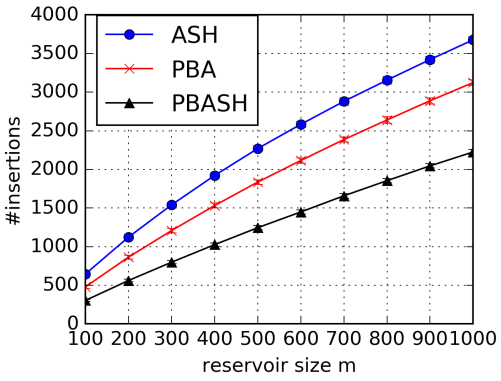


Figure 4.9: The number of insertions when the reservoir size is from 100 to $1,000$. Reprinted with permission from [1].

4.5.3 Computational Complexity

Figure 4.8 shows the processing time per packet of PBA, PBASH and ASH. No optimizations of ASH were used beyond the specification in [63]. With this proviso, the $O(m)$ cost for key eviction from reservoir size m for ASH appears evident through the initial linear growth of the time per packet. The noticeably lower growth for PBA and PBASH are expected due to its $O(\log m)$ time for inserting a new key after eviction of a current key. Since insertion/eviction is the most costly part for all algorithms we display the experimental number of these for each algorithm in Figure 4.9. PBASH has about half the insertions of ASH, another factor in its smaller per packet time. PBASH also has a smaller number of insertions than PBA. This is to be expected, since the PBASH pre-sampling stages causes fewer keys to be admitted to the reservoir.

5. COLLABORATIVE PACKET SELECTION

This chapter addresses the problem of how to store and apply large numbers of classification rules to network traffic. Our work is based on the observation that network bandwidth and downstream processing resources are relatively cheap compared with switch memory. We propose *COllaborative Overselection (CO2)* , a framework for packet classification that distributes classification functionality between switches and the network functions at end hosts in order to achieve the best tradeoff between switch memory, communication bandwidth, and computational resources.

5.1 Architecture Overview

5.1.1 Packet Classification by Collaborative Overselection.

The high-level idea of CO2 is to realize savings in switch memory by coordinating approximate classification in the switches with a downstream collection of selected traffic. Without such coordination, approximate classification would alone have to maintain the false positive rate within tolerable limits. Our key observation is that instead of merely *tolerating* false positives, we can *embrace* such false positives by allowing them to proceed to the receivers where they can be ameliorated as required (or not) by applications. Furthermore, we retain the flexibility to operate the most cost-effective trade-off between switch memory usage and downstream bandwidth consumed by this extra traffic, which we call the *overselected* traffic. Data center networks in particular are accommodating of overselection, due to advances in link speed (e.g., 40 Gbps or higher) and topology design that make bandwidth relatively available [98]. Furthermore, Cloud VMs provide cheap and flexible capacity to process overselected packets at receivers.

5.1.2 Architecture and Challenges.

Figure 5.1 shows the CO2 architecture. At a high level, the SDN controller installs the traffic classification rules at the switch. The switch performs approximate classification based on these rules, which has the side-effect of forwarding the overselected packets to the receivers. The receivers detect overselection and provide feedback to the controller which then dynamically re-

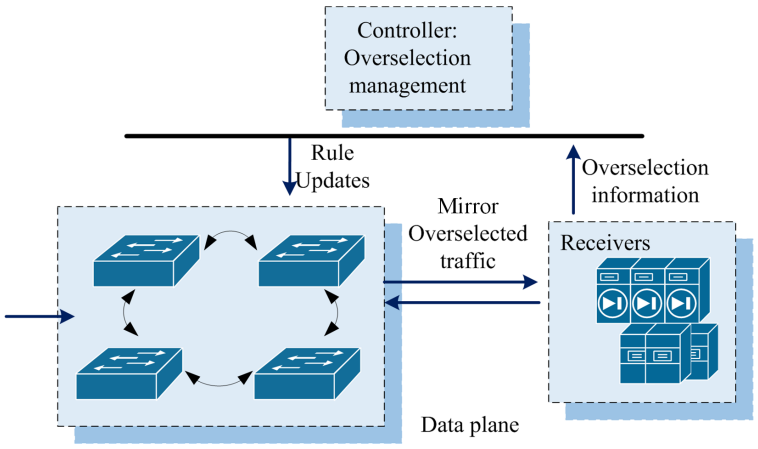


Figure 5.1: CO2 System Architecture.

configures the classifiers at the switches in order to control the amount of overselection. We now explore the rationale for this design.

The key challenge within this generic architecture is how to take advantage of the switch memory savings afforded by approximate classification while at the same time controlling the effects of overselection that potentially involves disproportionately large numbers of packets on specific prefixes. Thus we have four design decisions: (i) which approximate classification mechanisms to employ; (ii) what mechanisms to use to reduce overselection on specific prefixes, (iii) how to detect overselection; and (iv) how to configure the subsystems of CO2 to control overselection based on measurements.

Approximate Classification Mechanisms. CO2 employs an Approximate Packet Classifier (APC) that reduces switch memory usage by the combination of compressing rules with lossy aggregation, then storing the resulting rules in an approximate data structures. Hash-based approximate data structures such as Cuckoo Filters [77] need $O(\log_2(1/\epsilon))$ bits per entry, where ϵ is the false positive rate for retrieval. Thus, in CO2 where false positives are explicitly controlled and tolerated, Cuckoo Filters can provide considerable memory savings over exact approaches, e.g. based on hash tables or tries, for example reducing the storage for 500K entries by 56% while incurring a false positive rate of 1%; see [77].

Lossy rule aggregation achieves further memory savings. In the CO2 design, we tolerate some limited misclassification at the APC that occurs when smaller prefixes aggregated with larger prefixes that have a different action. These two forms of approximation are mutually beneficial for resource usage in that rule aggregation reduces the number of entries and hence reduces hash overselection collisions in a given memory size. In the previous example, the 500K rules can be aggregated down to 356K, achieving a further 16% memory reduction, while maintaining a 1% overselection rate.

Our work is distinct from the lossless approach of [85] that used dynamic programming applied to per rule traffic estimates, adding extra complexity to the system. We emphasize that in CO2, every packet receives a classification: there are no false negatives, in distinction with [75].

Reducing Overselection. Both classification mechanisms in the APC give rise to overselection. *Hash-based overselection* occurs due to hash collisions. *Aggregation overselection* results from rule prefixes being aggregated into a larger prefix associated to a different action. In both cases, overselection can take one of two forms. In the first, *wanted packets*—those that match an original rule—may be mapped to one of more incorrect actions. In the second, *unwanted packets*—those that match no original rule—may be mapped to at least one action other than blocking. In CO2, a *Prefilter*, placed in front of the APC, provides a transparent mechanism to intercept both types of packet before such actions occur. First, it can *whitelist* a subset of the original rules and forward matching packet directly to their receiver. Second, it can *blacklist* and block unwanted traffic which would otherwise be overselected.

Overselection Detection. In CO2, overselection is detected at the receiver, where processing cycles are cheaper. Furthermore, since the sensitivity to overselection is application specific, receiver policy can specify the importance of overselection, and may in some cases ignore it entirely.

Dynamic Control of Overselection. Since the degree and locality of overselection will change in response to traffic changes or network configuration, we require the capability reconfigure the APC and Prefilter in the switches responsively. The SDN controller plays a coordinating role, receiving feedback from receivers concerning overselection, and reconfiguring switches accordingly.

CO2 provides *network wide adaptation* across multiple switches and receivers, partitioning their memory between the APC and Prefilter functions.

5.1.3 Example Use Cases.

We give an illustrative example of operations supported by our architecture. Consider the task of detecting the heavy hitters (HHs) in a subnet with predefined traffic rule set. The original rules can be aggregated and installed to the switches, and the Prefilter is empty initially. A heavy hitter (HH) detector in the receiver sets an HH threshold and sends feedback to the controller, which reconfigures the switch to whitelist those flows whose byte volumes are around the threshold and blacklist the top overselected flows. As a result, the potential HHs are selected without further overselection, and without disruption form the largest overselected unwanted flows.

5.2 CO2 Switch Design

Within the architectural framework described above, the high-level problem we address is: *how to minimize the overselection rate incurred by approximate classification in the switch?*. In this Section we describe the detailed operation of the components of the CO2 collaboration and how their capabilities are used to solve this problem. The detail of the Approximate Packet Classifier are presented in Section 5.2.1. Specifically, Section 5.2.1.1 describes our algorithm for rule aggregation. The use of Cuckoo Filters as the approximate data structure within the APC in Section 5.2.1.2, where we focus on prefix matching and propose an efficient method to store and match prefixes of varying length. Trade-offs between hash and aggregation overselection from these methods are described in Section 5.2.1.3. Section 5.2.2 describe the Prefilter. While, we frame our initial description in terms of a single dimensional key prefix , we show how to extend our solution to multiple dimensions in a pipeline of single-dimensional tables in Section 5.3. With these components, we can restate our problem as follows: *how to minimize the overselection rate for a given set of rules through configuration of the size and parameters of the APC (storing lossy aggregated rules in an approximate data structure) and the Prefilter.*

One strawman approach is to use a Cuckoo Filter whose entry is the fingerprint generated by

hashing a rule, which we call it Cuckoo Only approach. However, if we have a small memory available for the rules, the only thing we can do with the base approach is to reduce the fingerprint size, but this will cause a large number of wrong selections. Therefore, the question is how to reduce the number of wrong selections with a small available memory?

5.2.1 Approximate Packet Classification Mechanisms

5.2.1.1 Lossy Rule Aggregation

We now describe a threshold based rule aggregation algorithm for which pseudocode is given as Algorithm 7. Each rule is key/action pair where the key specifies the properties of matching packets to which the action is applied. In this section we focus on keys taking the form of a single address prefix; the extension to multidimensional keys is presented in Section 5.3. We will use the term *rule-key* to describe a prefix in the original rule set, and define a *non-key* to be a prefix of minimal length that does not contain a rule-key. The set of rule keys and non-keys under an arbitrary prefix p contains all the addresses under p. Let $N(p)$ count the total number of rule-keys and non-keys under p. We introduce the *Aggregation Ratio*, $AR(p)$, as the ration of the number of non-keys to the number of rule-keys under an arbitrary prefix p. Hence $AR(p) = N(p \setminus \cup_i p_i)/N(\cup_i p_i)$ where the union is over all rule-keys p_i under p. For the example shown in Figure 5.2, prefix p (the blue node) contains 4 rule keys (shown as red nodes) with the remaining addresses falling under 2 non-keys (shown as black nodes). Hence $AR(p) = 1/2$.

In Algorithm 7, a threshold value T_{AR} for $AR(p)$ determines when the rule-keys under p can be aggregated to a single rule with key p, with aggregation occurring provided $AR(p) \leq T_{AR}$. It takes the original rule set and the target number N_t of keys as inputs. The outputs are the aggregated rule set and a set containing all non-keys. The main idea is to aggregate the keys to their high level prefix by $AR \leq T_{AR}$, and increase T_{AR} step by step until the number of keys is reduced to the target number. The threshold-based aggregation method is lossy. For the example shown in Figure 5.2. If all the rules under p are aggregated any traffic matching the two black nodes is subject to aggregation overselection.

Algorithm 7 Rule Aggregation

function RULE-AGGREGATION(())
    Build a trie from the ruleset, and let $\{v_i\}$ denote the nodes on the tree.
    for v_i **do**
        if v_i is nonleaf and v_i has one child **then**
            Add another child to v_i and label the newly added child as non-key.
        Calculate the number of the non-keys b_i, the number of the keys w_i covered by v_i and the aggregation ratio AR_i , and denote the tuple as $k_{v_i} = (b_i, w_i, AR_i = b_i/w_i)$.
    $T_{AR} = min(AR_i)\ for\ all\ v_i, n = N$
    while $n > N_t$ **do**
        for v_i **do**
            if $d_i \leq AR$ **then**
                $n = n - w_i + 1$
                PER-NODE-PROCESS(())
        $T_{AR} = min(AR_i)\ for\ all\ v_i$
    Traverse the tree to get all the keys.
    Return the keys and the non-keys.

function PER-NODE-PROCESS(())
    for u_j is the descendent of v_i **do**
        if u_j is labeled as non-key **then**
            Add u_j to non-key pool.
        Remove u_j
    for w_j is the ancestor of v_i **do**
        $k_{w_j} = (b_j - b_i, w_j - w_i + 1, d_i = b_i/w_i)$
    $k_{v_i} = (0, 1, d_i = 0)$

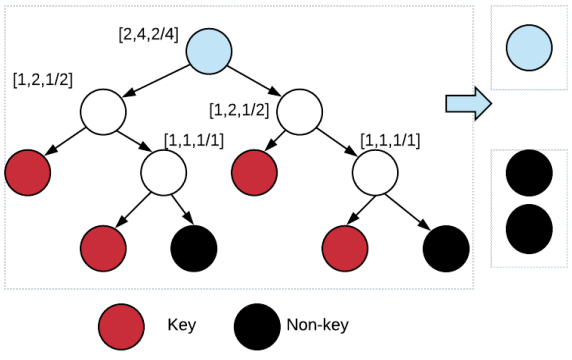


Figure 5.2: An example of rule aggregation, where the red nodes indicate wanted keys, and the black nodes indicate non-keys.

5.2.1.2 *Hash-based Approximate Prefix Matching*

We now describe approximate prefix matching of key prefixes using Cuckoo Filters [77] in the APC; the extension to multidimensional keys is presented in Section5.3. Cuckoo Filter storage locations are managed, in part, by a hash of the input known an the *fingerprint*. The Cuckoo filter as well adapted to the CO2 architectural principles because the field length of the fingerprint determines the hash collision rate and hence the overselection rate.

We now discuss storage and retrieval of rule prefixes of the form x/ℓ where x is the subnet address and ℓ the prefix length. A strawman approach is to maintain a Cuckoo Filter CF_ℓ for rules of each prefix length ℓ [99]. However, this approach is not memory-efficient because the prefix length distribution of rules is both non-uniform and variable. Thus is would be efficiently pre-allocate Cuckoo filters for different prefix lengths while meeting a target overselection rate. For this reason, we maintain a single Cuckoo Filter in which the fingerprint for each rule prefix x/ℓ is computed from the concatenated key $\ell.x$. All rules are stored using these keys. When a packet bearing address y arrives, the presence in the Cuckoo Filter of $y_\ell = \ell.(y \text{ AND}/\ell)$ is tested for each ℓ for which length ℓ-prefixes are stored, and the resulting actions reported. The prefix length is included in the key to avoid confusion between prefixes of different length that have the same subnet address. Fingerprint collisions between distinct keys $\ell.x$ gives rise to overselection, manifest either by multiple matches reported for a stored prefix or one or more matches reported for a non-stored prefix.

5.2.1.3 *Overselection Tradeoffs.*

Lengthening the fingerprint for a given number of stored keys reduces the overselection rate due to hash collisions. However, in fixed memory size, lengthening the finger print reduces the number of keys that can be stored, requiring further rule aggregation which would increase aggregation overselection. Thus we seek to determine the "ideal" finerprint length, i.e., that minimizes the combined overselection rate.

The false positive rate for Cuckoo filters can be analyzed in terms of its number of slots b per

buckets. According to [77] , the overselection rate $v \geq \frac{2b}{2^f}$ where f is the fingerprint length. Thus any fingerprint length f than can meet a combined overselection rate v_T must obey

$$f \geq f_T = \log_2 \frac{2b}{v_T} \tag{5.1}$$

We inspect fingerprint lengths $f \geq f_T$ to find the one that minimizes the combined overselection rate based on calibration from traffic samples. The target number of rules N_T can be derived after f_T is fixed by subtracting the resulting memory size of the Cuckoo filter from the total memory size.The analysis is based on the number of distinct flows, however, we use the bytes of the flows to calculate the overselection rate. We find from the simulation in 5.5 that the overselection rate roughly satisfies the trend where the rate decreases by 2 as f increases by 1. We can refer that the false positive rate with distinct flows and that with flow bytes are the same if the flow bytes are uniformly distributed. Practically, they are heavy-tail distributed, but the number of 'large' flows being overselected is much smaller than 'small' flows, if we assume the numbers are also heavy-tail distributed, by binning the flows into different ranges by their byte sizes, the total flow sizes of the bins can be roughly uniformly distributed.

5.2.2 Overselection Reduction

Prefiltering. In this section, we describe how Prefiltering operates to reduce both hash and aggregation overselection by intercepting the traffic responsible (both wanted and unwanted) before it reaches the APC. The Prefilter is dynamically configured by the controller on the basis of information supplied by the receivers. In order to optimize use of memory in the switch, the Prefilter is configured to treat those flows responsible for the largest overselection, as determined by the receivers and controller.

We now describe the two action of the Prefilter under this configuration. First, the Prefilter whitelists a subset of the original rules by applying their stated actions to matching traffic; we call the corresponding keys *white keys*. Whitelisting may be used to map to the correct action for a large flow which is subject to hash-based overselection, or which is associated with the incorrect

action due to lossy rule aggregation. Second, the Prefilter may blacklist designated unwanted traffic by blocking packets of selected non-keys; we call these *black keys*. Figure 5.2 illustrates the operation black keys in the Prefilter. The red nodes are aggregated to the root, but the root covers two unwanted prefixes (black nodes). The two IP addresses can be blacklisted in the Prefilter to avoid aggregation overselection.

The Prefilter may employ any data structure that supports exact membership queries. We implement the Prefilter with a Cuckoo hash table to ensure constant-time lookup for incoming packets.

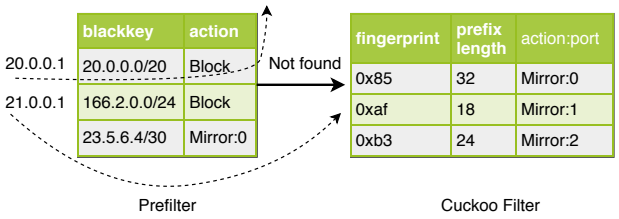


Figure 5.3: A pipeline of a Prefilter and a Cuckoo Filter for 1-field rule set.

5.2.3 Prefilter Update & Rule Selection

We now discuss the selection of rules for replacement in the Prefilter, based on the numbers of insertions computed by the controller. As described previously, the alternative of recomputing of lossy aggregation of a new ruleset in the APC is considerably slower and hence ill-suited to an adaptive approach. it is simpler and faster to update rules at the Prefilter.

5.2.3.1 Rule Identification at Receivers

When a receiver further maps a packet to classes, overselected packets are identified by the absence of a map entry for them. The receiver maintains a list of byte volumes of overselected keys, either exactly or estimated using an approximating data structure such as a Count-Min sketch [100]. Overselection notifications are then generated periodically for keys that exceed the receiver's configured overselection target. For the case with multiple actions, the flow with one action might be wrongly classified to another action, causing duplicated selection of this flow, such flows with top

byte size reported by the receiver will be also inserted to the Prefilter, but they could not be blocked as the Blackkey. Additionally, to reduce overhead in computation and transmission, receivers only transmit some proportion p_f of these notifications in decreasing order of overselection size.

5.2.3.2 Prefilter Update.

The Prefilter updating is similar to a LRU cache [101], where the oldest items in the Prefilter will be evicted if the capacity is exceeded. The goal is to install both the biggest IP addresses/prefixes or IP addresses/prefixes with the highest overselections to Prefilter. Once the controller gets the feedbacks, it checks membership from the aggregation Trie above and identifies the associated Blackkeys or Whitekeys. Those keys will be installed to the Prefilter in decreasing order of overselected traffic volume, up to the Prefilter capacity. Keys remaining in the Prefilter from the previous period are evicted by time counts to accommodate feedback IP addresses/prefixes from the current period.

5.3 Extension to Multiple Dimensions and Actions

The APC installs aggregated rules in the Cuckoo Filter, by hashing on rule key to generate the fingerprint, then inserting both the fingerprint a rule action into the Cuckoo filter. As stated in Section 5.2.1.2, address fields include prefix length through concatenate in order to distinguish prefixes of with the same subset address but different prefix lengths.

5.3.1 Pipeline in Multiple Dimensions

In a Cuckoo Filter, the maximum number of membership checking for a single IPv4 prefix is 32, the case becomes worse for a rule with both source IPv4 prefixes and destination prefixes, the maximum number might be 32×32. The pipeline proposed in this work aims to reduce the number of membership checking. The pipeline contains two Cuckoo Filters, the first is to hold the fingerprints generated by the source prefixes, and the second holds the fingerprints generated by the combination of the source and destination prefixes. The tricky idea is to extract the prefix length of the destination prefix and put it in the first Cuckoo Filter. When a packet arrives, it first tries to match an entry in the first Filter with the source address. Once a match is found, the

prefix length of the destination prefix is obtained, next, the combination of the source prefix and the destination prefix is generated by masking the destination address of the packet and combine it with the source prefix. The combination will be used to query the second Cuckoo Filter. In this design, the maximum number of queries equals that of the first Filter. One example of such pipeline is shown in Figure 5.4.

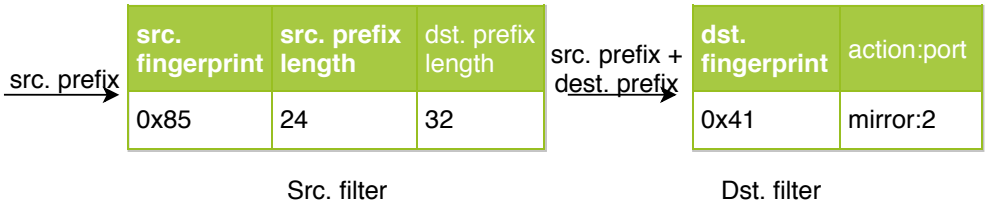


Figure 5.4: A pipeline of Cuckoo Filters for rules with source and destination prefixes.

5.3.2 Lossy Aggregation in Multiple Dimensions

Two-dimensional aggregation for source and destination address fields is performed by generalizing from trees to rectangles partially ordered by inclusion. For each key, we find the smallest rectangle including all rule-keys of minimal distance, then compute the aggregation ratio as the ratio of the number of keys and to non-keys in this rectangle. Aggregation occurs provided $AR \leq T_{AR}$. The approach is tunable through T_{AR} or enlarge the rectangle to include more distant neighbors.. We illustrate with an example in Figure 5.5, where each key bears a tuple comprising the number of non-keys, the number of keys, and the aggregation ratio. For example, the tuple $(1, 2, 1/2)$ indicates 1 non-key, 2 keys and the aggregation ratio of 1/2 for this rectangle. If the threshold T_{AR} is set to be 0.5, we will aggregate the top left and the bottom right rectangles to two keys and 3 non-keys.

5.3.3 Multiple Actions

We now describe how a single action is associated to an lossy aggregate of rules with distinct actions, realized through an extension of Algorithm 7. We explain with with to a single address

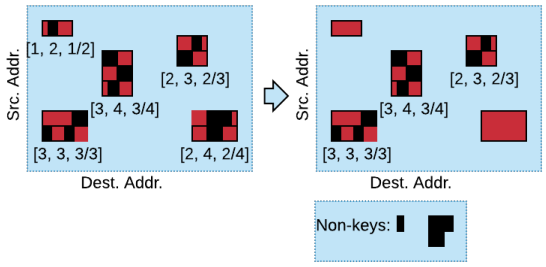


Figure 5.5: An example showing extending the aggregation algorithm to deal with 2-dimensional rules.

dimension; the approach generalizes to multiple dimensions following Section 5.3.2 above. Given a candidate aggregation prefix, compute the number b of non keys, and the number w_j of keys under it for each action j. The aggregation ratio for the prefix is defined as

$$\frac{b + \sum_j w_j - \max_j w_j}{\max_j w_j}. \tag{5.2}$$

The action of maximal w_j is called the dominant action, and is attached to the prefix p if aggregation is performed. After aggregation, some prefixes may become nested in a high-level prefix of a different action, for example, if 10.0.0.0/24 has action 1 and 10.0.0.0/16 has action 2. An example of 2-action aggregation is shown in Figure 5.6. For each non-leaf node bears a 4-tuple whose first elements is the number of non-keys, the next two fields are the number of keys for actions 1 and 2, and the last field is the aggregation ratio. With $T_{AR} = 0.5$, all the nodes are aggregated to the root, as a result, we get one aggregated key (the blue node), one nested prefix (the green node) and one non-key (the black node). To reduce overselection, we would put all the nested prefixes to the Prefilter. However, the number of those prefixes should be small enough to be accepted by the Prefilter while allowing some space for Black keys.

5.4 Distributed Rule Placement

In this section, we describe the algorithm to distribute flow rules to multiple switches for per-flow monitoring. The motivation of employing multiple switches is that a large number of rules

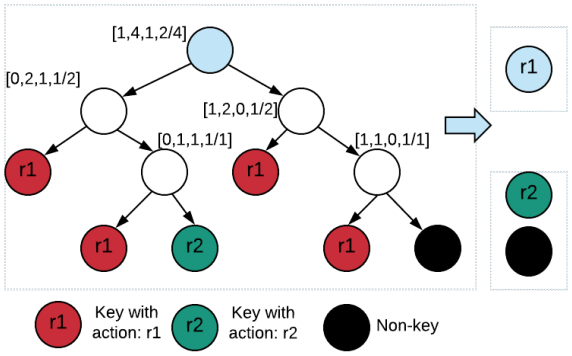


Figure 5.6: An example showing extending the aggregation algorithm to deal with the rules with many actions.

may outstrip the flow table of a single switch. Our approach leverages the controller to place and redistribute the rules in order to balance the memory usage of switches; see subsection 5.4.1. Several improvements of the rule redistribution are proposed to achieve a constant time complexity and a linear space complexity, which is described in subsection 5.4.2.

5.4.1 Rule Placement

The rule placement to multiple switches requires the collaboration between the SDN switches and the controller. Given a set of switches, denoted by S, and a set of flows arriving in sequential order, the switch extracts the flow rule (e.g., 5-tuple) and sends it in a "packetIn" message to the controller when the first packet of each flow arrives. Let r denote the flow rule, and $S[r]$ denote the routing path, the controller obtains $S[r]$, selects a switch in $S[r]$ and install r to this switch.

One basic approach, employed in DCM [102], of determining a switch for the rule installation is to choose the switch, which has the minimum memory occupancy. We use the term *Basic* to denote this algorithm, which is shown in Algorithm 8. *Basic* implements a procedure named $insertRule$: line 2 aims to choose the switch $\widehat{s}$ whose memory usage is the minimum in $S[r]$; line 3 inserts r to $\widehat{s}$; and line 4 updates the memory of $\widehat{s}$, denoted by $M[\widehat{s}]$, by increasing it by 1. However, this algorithm causes the memory to increase more quickly for a subset of switches than others, and therefore, those switches reach its capacity earlier than others. Figure 5.7 shows an

example of the imbalanced rule distribution. In this example, there are five switches, denoted by $s_i, i \in [1, 5]$, and two sets of rules, where the blue arrow denotes the routing path of 18 flows in the first set of flows, and the red one is the routing path of 30 flows in the second set. In stage 1, there exist no rules at the switches. In the second stage, the blue rules (18 flows) arrive at the controller sequentially, and the controller places them to s_1, s_3 and s_4 according to algorithm 8. The outcome of this stage is that each switch bears six rules, which are labeled by the blue number in Figure 5.7. In the third stage, the red rules (30 flows) arrive at the controller. After running algorithm 8, 12 out of the 30 rules are installed to s_2 and s_5, respectively, and six are assigned to s_3, which are labeled as the red number in Figure 5.7. The allocation of the two sets of flows results in less memory usage on both s_1 and s_4 than the other switches.

Algorithm 8 *Basic*

1: **procedure** $insertRule(r, S, M)$
2: $\widehat{s} = arg_s min M[s]$ for $\widehat{s} \in S[r]$,
3: Insert r to $\widehat{s}$
4: $M[\widehat{s}] += 1$

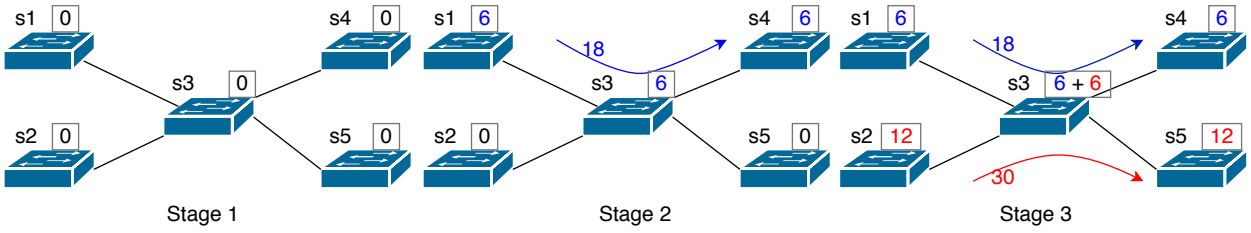


Figure 5.7: One example of *Basic* with 18 blue flow rules and 30 red rules.

Given the drawback of the basic algorithm, we propose to redistribute the rules to balance the memory usage of the switches. The key idea is to evict a flow rule and reinsert it to another switch when a new rule is installed on the switch. The goal of the eviction and the reinsertion aims to

reduce the difference in memories of the two switches. We use the term *Min* to denote the redistribution algorithm. *Min* is shown in Algorithm 9, which presents three procedures: the insertion procedure, denoted by $insertRuleToSwitch$, the redistribution procedure, named $redistributeRule$ and the procedure of removing a rule, named $removeRuleFromSwitch$. $redistributeRule$ takes s, S, M, and T as the inputs, where $T[s]$ is a table which records the switches except s along the routing path of the rules in the flow table of s. Line 7 selects switch s' from $T[s]$ whose memory usage is the minimum. Let the term *intersection rules of* (s, s') denote the rules which have both s and s' on their routing paths and installed in switch s, Line 11 chooses one *intersection rule*, denoted by r', to evict. Lines $12 - 13$ remove r' from s and insert r' to s'. The procedure ends when $M[s]$ is the minimum in memory usage among $T[s]$, as shown in line $8 - 10$. We present one example, described in Figure 5.8, to illustrate the workflow of *Min*. When the $16th$ rule, denoted by r (in red), from the red rules arrives at the controller, r is assigned to switch s_3, resulting in 8 rules at $s3$. The controller evicts r' (in blue), which belongs to the blue rules, and reinserts it to s_1 in stage $2 - 2$. The eviction process is repeated for the remaining ones from the red rules. And the distribution of the rules over the 5 switches is more uniform than Figure 5.7 after all 30 red rules are settled (stage 3), where the difference of rules between the maximum and the minimum switches is 1.

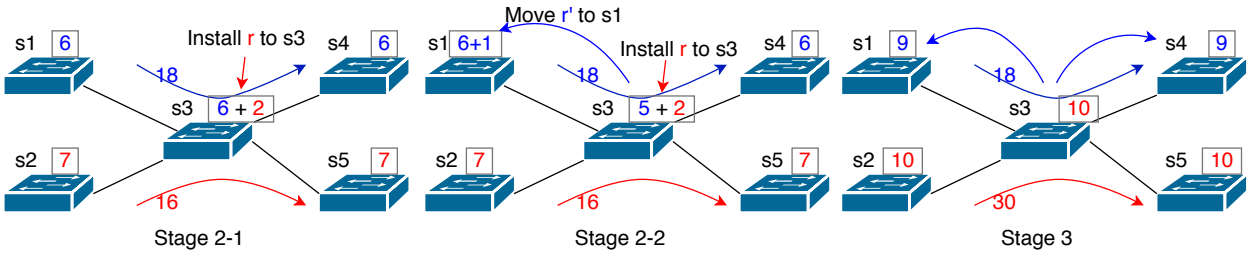


Figure 5.8: One example of *Min* with 18 blue flow rules and 30 red rules.

Algorithm 9 *Min*

1: **procedure** $insertRuleToSwitch(s, r, S, M, T)$
2: Insert r to s
3: $M[s] += 1$
4: $redistributeRule(s, S, M, T)$
5: **procedure** $redistributeRule(s, S, M, T)$
6: $s' = arg_s min M[s]$ for $s \in T[s]$,
7: **if** $M[s'] \geq M[s] - 1$ **then**
8: return
9: Select one *intersection rule* of (s, s'), denoted by r'
10: $removeRuleFromSwitch(s, r', S, M)$
11: $insertRuleToSwitch(s', r', S, M, T)$
12: **procedure** $removeRuleFromSwitch(s, r, M)$
13: remove r from s
14: $M[s] -= 1$
15: **procedure** $insertRule(r, S, M, T)$
16: $\widehat{s} = arg_s min M[s]$ for $s \in S[r]$,
17: $insertRuleToSwitch(\widehat{s}, r, S, M, T)$

5.4.2 Improvements

Min, shown in algorithm 9, encompasses the redistribution of rules among the switches over the network. However, this algorithm is not time-efficient, and as a result, it cannot process the flow rules in real-time. Among the lines in *Min*, Line 7 and line 11 is time-consuming because line 7 needs to iterate over all the switches in $T[s]$, and line 11 needs to choose an *intersection rule* among all the rules. Additionally, the procedure $insertRuleToSwitch$ (line 13) is called recursively, which also increases the time complexity.

One approach to reducing the time complexity of line 7 in algorithm 9 is to compute the minimum value for each switch in parallel by storing $T[s]$ in a hash heap [103], which accomplishes to obtain the minimum in near-constant time. However, this method requires a thread for each switch, which is resource consuming. Our approach is to randomly select a switch among $T[s]$ instead of the one with the minimum memory usage. One issue of this randomization is that we may not choose a switch whose memory usage is lower for the evicted rule to install. In order to solve this, we propose a policy of bidirectional rule eviction. Let the randomly selected switch to

be denoted by s"; the idea is to redistribute a rule from s to s" when the memory usage on s is more significant than s", otherwise, evict a rule from s" and reinsert it to s. We use the term *Rand* to denote the algorithm, which includes the random selection and the bidirectional eviction policy.

Our approach to reducing the computation time of line 11 in algorithm 9 is to store the *intersection rules* in a hash table, which is denoted by H. The key of H is the pair $(s, s")$, and the value is a hash set, denoted by $H[s, s"]$, which stores the *intersection rules* of $(s, s")$. The approach allows obtaining one *intersection rule* r' in constant time. But it requires N^2 number of hash sets for all pairs of switches, where N is the number of switches in the topology. In order to reduce the space complexity, we employ the neighbors of s, denoted by $B[s]$, to replace switch set $T[s]$. Note that $\forall s" \in B[s]$ $s"$ should also be included in $T[s]$. Through this approach, we reduce the number of hash sets to the number of links in the topology. The operation of the hash table includes two procedures: insertion and removal, which are described in Algorithm 10. Lines $1-9$ insert a rule to table H. For details, line 2 extracts the neighbors of s from the routing path S_r; line 7 adds the *intersection rule* to the hash set $H[(s, s')]$; and line 8 inserts the neighbor switch to B. Lines $10-21$ remove a rule from H and update B accordingly. Let the term *Rand-neighbors* to denote *Rand* with neighbors. Note that there is a trade-off between using *Rand* and *Rand-neighbors*: *Rand* requires more memory of the controller than *Rand-neighbors* but can reduce the difference of memories between switches quickly.

Combining the improvements with algorithm 9, *Rand-neighbors* is shown in Algorithm 11. Lines $1-6$ present the procedure of $insertRuleToSwitch$: lines $2-3$ insert rule r to switch s and update the memory usage; line 4 redistributes the rules, which is detailed in lines $7-21$; and line 5 adds r to table H. In the procedure of $redistributeRule$, lines $8-11$ restricts the number of recursion in order to decrease the time complexity of the algorithm; line 12 randomly selects $s"$ from the neighbors $B[s]$; line $13-15$ reverses the eviction direction when $M[s"] \geq M[s]-1$; and line $16-20$ remove rule r' from switch s, insert it to $s"$ and also update table H.

The time complexity of both *Rand* and *Rand-neighbers* is the same as *Basic*, which is $O(m)$, where m is the length of the routing path. The time complexity of *Min* is $O(N)$ when *Rand* also

Algorithm 10 Store/remove intersection rules to a hash table

procedure $insertToTable(s, r, S_r, H, B)$
    S'_r = the neighbors of s in S_r
    for $s" \in S'_r$ **do**
        if $s" == s$ **then**
            continue
        $H[(s, s")].add(r)$
        $B[s].add(s")$
procedure $removeFromTable(s, r, S_r, H, B)$
    S'_r = the neighbors of s in S_r
    for $s" \in S'_r$ **do**
        if $s' == s$ or $(s, s") \notin B$ **then**
            continue
        $H[(s, s")].remove(r)$
        if $H[(s, s")]]$ is empty **then**
            del $(s, s")$ from H
            $B[s].remove(s")$

Algorithm 11 *Rand-neighbers*

procedure $insertRuleToSwitch(s, r, S, M, H, B, iteration)$
    Insert r to s
    $M[s] += 1$
    $redistributeRule(s, S, M, T, iteration)$
    $insertToTable(s, r, p, H, B)$
procedure $redistributeRule(s, S, M, B, iteration)$
    if $iteration > 1$ **then**
        return
    $iteration += 1$
    Randomly select a switch from $B[s]$, denoted by $s"$,
    if $M[s"] \geq M[s] - 1$ **then**
        $redistributeRule(s", S, M, B, iteration)$
    Select one *intersection rule* of (s, s'), denoted by r'
    $removeRuleFromSwitch(s, r', S, M)$
    $removeFromTable(s, r', p, H, B)$
    $insertRuleToSwitch(s", r', S, M, B)$
    $insertToTable(s", r', p, H, B)$

employs algorithm 10, where N is the number of switches since finding the minimum switch takes $O(N)$ time. Practically, *Rand* and *Rand-neighbers* will use more time than *Basic* because they need to update the hash table. In addition, m is constant for data centers, and therefore, *Rand* and *Rand-neighbers* can achieve constant time for per-rule placement for data centers.

5.5 Evaluation

In this section, we evaluate the performance of CO2 under a wide range of configurations using realistic policies and traffic traces. Our simulation results show that at a 1% overselection rate for CO2, Cuckoo filters needed 56% more memory to achieve the equivalent false positive rate and Cuckoo hashes 178% more memory for the rule set with single IPv4 prefix. The Cuckoo hashes uses 276% more memory for the rule set with source and destination prefixes.

5.5.1 Experiment Setup

Classification rules. We generate a 500k-rule set on source and destination pairs using ClassBench [104], we will use IP pairs for short in the following paper. This rule set contains 500K distinct source prefixes and 152 distinct destination prefixes, which covers 54 million distinct IP pairs.

Traffic traces. We use a one-hour CAIDA trace [91], which contains anonymized passive traffic traces from CAIDA's passive monitors in 2015. It contains traffic traces from the 'equinix-chicago' high-speed monitor. The trace contains 32 million distinct IP pairs. For each IP pairs in the CAIDA trace, we randomly map it to one of the IP pairs in the ClassBench rules. In this way, all the rules are ensured to have matching traffic. We randomly generate flows which do not match any rule in the rule set to the trace with a given probability, the flow sizes are Pareto ($\alpha = 1.1$) distributed.

5.5.2 CO2 on 1-d Rule Set

We compare CO2 with the *Cuckoo Only* described in 5.2.

Memory saving with overselection. First, we compare CO2 and Cuckoo Only by fixing different memory usage from 0.5 MB to 1.5 MB. 50% of the traffic that do not match any rules in the rule

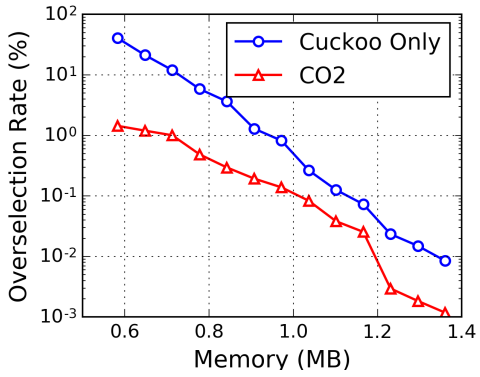


Figure 5.9: Comparison between CO2 and Cuckoo Only with multi-dimension rules.

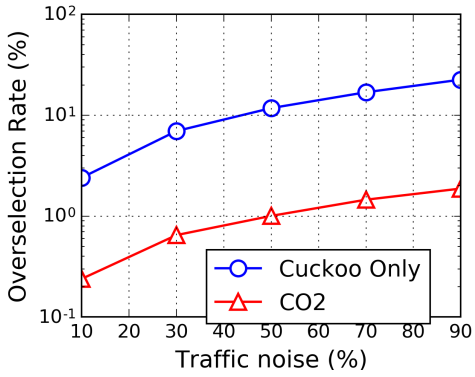


Figure 5.10: The overselection rates of CO2 and Cuckoo Only approach with increasing traffic which matches no rules in the rule set.

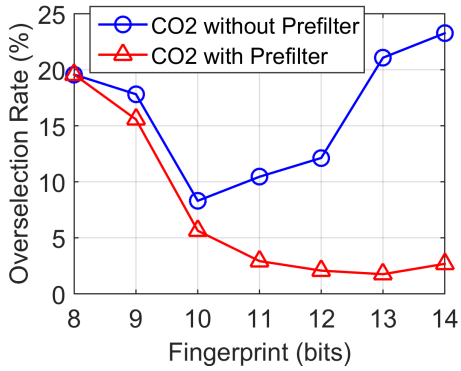


Figure 5.11: The trade-off between aggregation and fingerprint length.

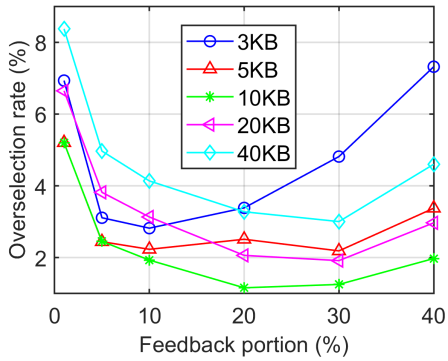


Figure 5.12: The overselection rate with different feedback portions and Prefilter capacities.

set are injected into the original trace. The result is shown in Figure 5.9. We can see from it that the CO2 is always better than Cuckoo Only. As fingerprint length increases, the overselection rate decreases accordingly, and even just with a memory of 0.7MB (the third point in the figure), the overselection rate can be less than 1%. This also indicates that with the same overselection rate, CO2 is more memory saving. From the figure, we can see that at a 1% overselection rate for CO2, Cuckoo filters needed 56% more memory to achieve the equivalent false positive rate and Cuckoo hashes 178% more memory since each entry in a hash table requires 32 bits for a 32-bit IP address

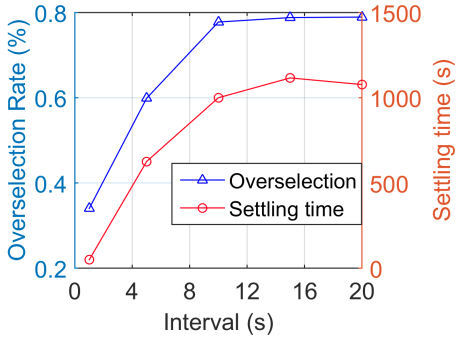


Figure 5.13: The overselection rate and the settling time against the interval.

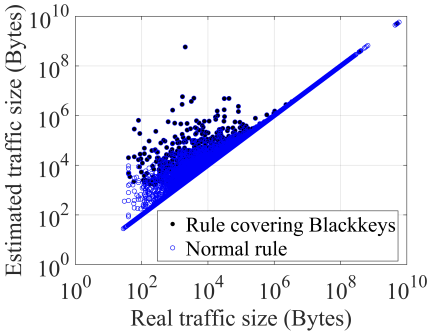


Figure 5.14: The scatter plot of estimated vs. real traffic size per rule.

and CO2 only needs 9 bits. We can conclude that the rule aggregation and the prefiltering perform well in reducing overselection rate and thus achieving the goal of memory saving.

Handling traffic that does not match any rules. We also inject various portions of noise traffic that do not match any rules in the rule set into the traces. The portions are from 10% to 90% to the original traffic volume. We set the memory usage to be 0.7 MB, and the fingerprint length for Cuckoo Only and CO2 are 7 and 13 separately. The result is shown in Figure 5.10, from which we can see that both CO2 and Cuckoo only incur increasing overselection rates when the traffic noise increases, but that of the former approach remains much lower.

5.5.3 CO2 on 2-d Rule Set

Using the pipeline proposed in subsection 5.3, even with very small fingerprint size, the overselection rate becomes small for both Cuckoo only and CO2, which is shown in Table 5.1. And CO2, which implements the extended aggregation algorithm in subsection 5.3 is able to reduce the overselection rate by up to more than 90%. The Cuckoo Hash table uses 276% more memory than CO2 without overselections.

5.5.4 Breakdown of Memory Savings

We now study the memory savings for each CO2 feature.

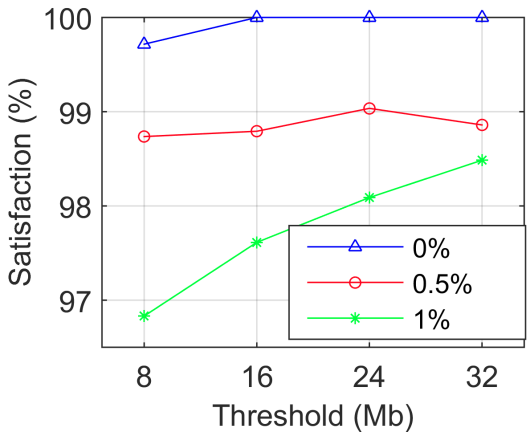


Figure 5.15: Satisfaction vs. threshold Φ for overselection $0\%, 0.5\%$ and 1%.

Fingerprint	5	6	7	8
Cuckoo Only	0.0034	0.0016	9.8110^{-5}	9.1610^{-5}
CO2	0.0004	0.0002	6.8110^{-6}	1.2510^{-6}

Table 5.1: The overselection rates for both Cuckoo Only and CO2 for a 2-d rule set.

The tradeoff between aggregation and fingerprint length. As described in Section 5.2.1.3 under fixed memory there is a tradeoff between overselection reduction obtainable by rule aggregation and by lengthening the fingerprint in the Cuckoo Filter. We study this trade-off experimentally for fingerprint lengths from 8 to 14, the lower bound being set set according to eq. (5.1) in a total memory of 0.7 which gives an overselection rate of 1%. Figure 5.11 shows the resulting overselection rates with and without prefiltering. The minimum overselection rate with the Prefilter (1%) is noticable lower than without (8%), and is less sensitive to the fingerprint length.

A small Prefilter of 10–20KB suffices, with feedback rate of 10%-20%. We study the dependence of the overselection rate on the capacity of Prefilter and the feedback proportions p_f of possible overselection notifications that the receivers send to the controller. From a total switch memory of 0.7MB we explore Prefilter capacities of 3KB, 5KB, 10KB, 15KB and 20KB, together with feedback proportions of 1% – 40% . A fingerprint length of 13 is used in this study. The

results are shown in Figure 5.12. From it, we observethe overselection is smallest for feedback ratios of 20%–30%. The reason why the overselection rate begins to increase at 25% feedback is that a larger feedback ratio makes the Prefilter occupied by small flows. From the figure, we can also see that the overselection rate decreases with the increasing of Prefilter capacity and tends to stable at the memory of 10 KB, therefore, the best Prefilter sizes are 10–20KB.

Dependence on update frequency. Updates from each receiver to Prefilter occur periodically. Shorter update periods incur higher load on the switch, while longer update periods may leave stale rules in the Prefilter. stinactive rules staying in the Prefilter for a long time. We conduct experiment with update periods of 1s, 5s, 10s, 15s and 10s; see Figure 5.13. With larger interval, both the overselection rate and the convergence settling time, which is defined as the time required for the response curve to reach and stay within a range of certain percentage (usually 5% or 2%) of the final value [105]. increase, due to stale rules catching less overselection. We can choose the interval to be within 1-5 seconds for the tradeoff of the update frequency and the performance of the algorithm. If the overselected flow is of small size, it might be ended before it is feedback to the Prefilter. Otherwise, if the overselected flow is of large byte size, before it is installed into the Prefilter, some packets would be already overselected.

5.5.5 CO2 Overhead

We implemented CO2 prototype on Open vSwitch v2.3.0 [106]. We evaluate our prototype on an Ubuntu server on four 12-core, 2.4GHz Opteron processors with 48GB memory. We set up one Open vSwitch instance and 50 hosts in Mininet [107] and group them into 25 client-server pairs. Each pair is connected by the Open vSwitch instance. We run a Netperf's [108] TCP_CRR test between every C/S pair. Each client host sends one million 64-byte packets to the server host. Thus, there are 25 connections connected by Open vSwitch in parallel. We measure the packet forwarding performance and rule update performance of CO2 embedded Open vSwitch against the original one. In our test, the original Open vSwitch can get 1.35M pps forwarding throughput.

Rule update performance. The updated rules are sent from user space to CO2 module in the

kernel via the MMAP based interface. In our test, the CO2 prototype can update 380K rules per second, which ensures it can work well in the highly dynamic environment. As a result, the CO2 module is feasible in the existing Open vSwitch and will not become the bottleneck. Therefore, CO2 can achieve real-time rule update with several ms delays per rule. We can expect a much better performance using hardware.

P4 implementation. CO2 can be implemented using the P4 language [109]. The packet processing at the Prefilters and the Cuckoo Filters are similar, so we only describe how we implement the Cuckoo Filters. We use four registers of size 135869 to store the fingerprints such that 500K rules can be put into the storage with 92% load factor. And we use 156 lines of coding in total for the pipeline implementation. When a packet arrives, the switch generates a fingerprint by hashing on the selected packet header fields and stores it in the metadata. The switch then generates the indexes using a different set of hash functions, matches the fingerprint, reads the output port values and then mirrors the packets to the output port. The controller keeps a copy of all the rules in both the Prefilters and the Cuckoo Filters and can update the entries in these tables accordingly.

5.5.6 The Performance of Rule Distribution

In this subsection, we evaluate the performance of four algorithms – *Basic*, *Min*, *Rand* and *Rand-neighbors*, under the configuration using realistic traffic traces (§ 5.5.6). Our simulation results show that *Rand-neighbors* can increase the maximum number of flow rules over the network compared to *Basic* by 31.27% and 25.20% in two simulations with different switch capacities, respectively.

The simulations are conducted under a fat-tree topology [110], which is commonly used in data centers. The topology used in this work contains 4 core switches, 4 pods with 4 aggregation switches, and top of rack switches in each, respectively.

A 10-min CAIDA trace [91] is chosen as the traffic trace in the evaluation. The trace contains anonymized passive traffic traces from CAIDA's passive monitors in 2015.

Two simulations are conducted with the capacities of the switch to be $10,000$ and $20,000$. The simulation results show that *Rand-neighbors* increases the maximum number of flow rules over

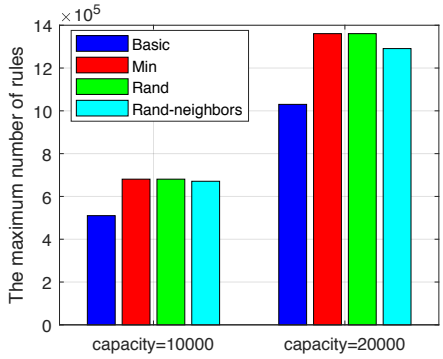


Figure 5.16: The maximum number of flow rules upon one switch reaches its capacity.

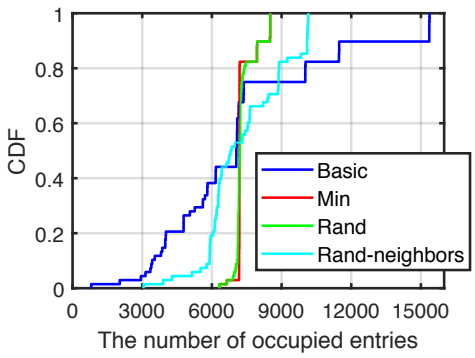


Figure 5.17: The CDF plots of *Basic*, *Min*, *Rand*, and *Rand-neighbers*

	Min	*Rand*	*Rand-neighbers*
Capacity$= 10k$ (increase by %)	33.30	33.11	31.27
Capacity$= 20k$ (increase by %)	32.03	31.97	25.20

Table 5.2: The percentage increase of the maximum number of rules compared to the *Basic*.

the network by 31.27% and 25.20% under the two setups, respectively. Each trial of the simulation ends when there exists one switch reaching its capacity. Upon the end of the trial, we collect the total number of flow rules over the network. The average numbers of the rules of 10 trials for each of the four algorithms are shown in Figure 5.16. The percentage increase in the number of rules is presented in Table 5.2. From both Figure 5.16 and Table 5.2, we can see that *Min* and *Rand* have an increase of approximately 33% compared to *Basic* when the capacity of the switch is $10,000$; and the number is around 31% when the capacity is $20,000$. The increase in the flow rules indicate that *Min*, *Rand* and *Rand-neighbors* are effective in balancing the flow tables over the network compared to *Basic*.

We conduct another simulation with $100,000$ flows and unlimited switch capacity. The CDF plots, shown in Figure 5.17, present the distribution of memory usage (occupied entries in flow table per switch) over the switches. From the CDF plots, we can see that *Min*, *Rand* and *Rand-neighbors* have more uniform distributions than *Basic*. The maximum and the variance of the

Table 5.3: The maximum and variance of #occupied_entries per switch for *Basic*, *Min*, *Rand*, and *Rand-neighbers*.

	Basic	Min	Rand	Rand-neighbor
Maximum #occupied_entries	15370	8514	8511	10151
STD	360.04	46.13	46,84	171.04

entries among all switches are shown in Table 5.3. From it, we can see that *Min* and *Rand* have the smallest max values, which are around 8500 for both. They also have the smallest variance, about 46. Rand-neighbor achieves a decrease of 33.96% in the maximum value, and 52.5% in variance compared to *Basic*.

We also study the running time per rule with the four methods. The simulation is conducted with a total of $100,000$ flows and unlimited switch capacity. The results are shown in both Figure 5.18 and Table 5.4. *Rand* and *Rand-neighbers* require approximately 50% more time per rule, and *Min* uses $5x$ time, compared to *Basic*. The results are consistent with the analysis in subsection 5.4.2, where *Rand* and *Rand-neighbers* have the same time complexity with *Basic*, but uses more time than *Basic* in evaluation to manage the hash tables in the algorithm.

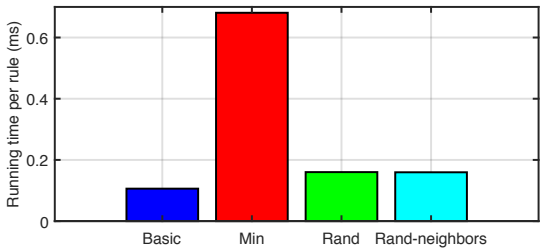


Figure 5.18: The per-rule running time of *Basic*, *Min*, *Rand*, *Rand-neighbers*

5.6 Managing Side-Effects

Although overselection results in a significant reduction in memory consumption at switches, the downstream impact of the overselected traffic on network functions must be assessed.

Table 5.4: The increase of running time per rule of *Min*, *Rand*, and *Rand-neighbers* compared to *Basic*.

	Min	Rand	Rand-neighbor
Increase by (%)	541.89	50.89	50.44

5.6.1 Side-Effects of Overselection

We do this in various classification scenarios and outline measures by which the impact may be limited. We identify three categories of effect: (i) *Load increase:* overselected traffic is processed by downstream network functions and hence increases their processing load, but does not otherwise affect their outcome; (ii) *Perturbation:* overselected traffic perturbs downstream functions in a controlled manner but only by perturbing their operating point; (iii) *Accuracy:* overselection changes measured traffic statistics, potentially requiring compensation or recalibration.

Load Increase Only *Mapping and Middleboxes.* Downstream applications, e.g. running in middleboxes, may maintain an exact database of all packet IP addresses currently mapped to each class; see [111]. These applications can filter out overselected packet by determining whether the IP address of the selected packet lies within that class. When this step is part of the standard packet processing in the middlebox–such as mapping the IP address to a customer for accounting purposes–packets not bearing an IP address in the mapping database are explicitly discarded. In a related scenario, overselected packets are ignored by not being selected. For example, the WAN optimizer always optimizes the HTTP traffic and simply ignores other types of traffic.

Intrusion Prevention Systems. Middlebox may process overselected packet regardless. If an intrusion prevention system (IPS) receives more traffic than expected because of overselection at the switch, it simply inspects more traffic and can still prevent malicious traffic. Thus the effect of overselection is to increase the load on an existing process. In both these cases, the overselection rate must be constrained in order to keep this load within acceptable limits.

Perturbation of Operating Point. *Load balancing.* In load balancing between servers, rules are set in switches to distribute the load amongst servers. We consider two scenarios. Firstly, when

the load balancing control responds to actual fractions distributed to the servers, then this control will adapt to the actual selected traffic, including overselection. This will be the case when the servers communicate information concerning their receiver load to the controller. In the second scenario, we assume that servers are able to redistribute excess load to other servers. In both cases, the perturbation on the system from the expected operating point remains small provided the overselection rate remains small.

Anomaly Detection. Traffic mirroring is often used by network operators to select traffic for anomaly detection and performance diagnosis [68, 112]. In this context, we view overselection as adding in a portion of normal traffic that does not match the intended selection criterion. As such, it adds to the baseline for normal traffic, but does not further affect additive traffic measures for anomaly detection, such as those derived from time series analysis; see e.g.[113].

Accuracy Affects. *Measurement.* Simple measurement applications count packets that match specific criteria, without filtering out overselected packets through mapping as described above. In this case, overselection increases measured packet and bytes counts. System measurements may already employ approximate measurement measurements. Thus the impact of overselection in practice depends on (i) the distribution of the overselection over the set of rules; (ii) the scale of its effects relative to the inherent inaccuracies of the measurement subsystem, and (iii) the accuracy requirements of downstream applications. In the next subsection, we examine points (i) and (ii) in an experimental setting.

5.6.2 Statistics of Overselection

The feedback mechanism is based on controlling the total overselection rate at aggregate level over receivers. Measurement functions typically report statistics at a finer level of granularity. Therefore we now drill down within our previous results to report overselection at a per rule granularity. We found the statistical effects of overselection to be quite limited: 97.5% of rules have a relative byte error of 1% or less. Figure 5.14 shows a scatter plot over rules of measured vs. the bytes and shows that larger relative errors are mostly confined to rules with smaller byte volume.

Impact on Heavy Hitter Detection: We study the impact of overselection on the accuracy of specific measurement function, namely Heavy Hitter (HH) detection [114] [115]. Given a set of S with N elements and an absolute threshold Φ, a heavy hitter is an element whose frequency in S is no smaller than Φ. In a network, the element can be identified by a packet header field, for example, the source IP address. A CountMin (CM) sketch [116] is a space-efficient data structure for approximate aggregation and is hence well suited to HH measurement. A performance measure for detection is the Satisfaction $S = 1 - FP/T$ where FP is the number of the false positives, and T is the total number of selected HHes; see [114]. Figure 5.15 shows the satisfaction for at absolute threshold Φ of $8Mb$, $16Mb$, $24Mb$, and $32Mb$ for total overselection rate of 0%, 0.5%, and 1%. While satisfaction decreases with overselection it remains above 96% for all receivers,

6. SUMMARY AND CONCLUSIONS

In this chapter, we summarize the key points of this dissertation.

We have proposed *IcorLIm*: a framework to detect hash functions. *IcorLIm* comprises two major component: identifying the victim via hypothesis testing and detecting the correlated ECMP groups. Comparing *IcorLIm* to two strawman approaches, it increases the F1 score by 40%.

Besides detecting the hash correlations, we also proposed a novel framework to relieve the load imbalance raised by those correlations. The framework handles the mesh network via a Coprime-based approach, and eliminates the hash correlations for hierarchical networks by color combining. The proposed approaches have been proved to be useful via a set of simulations conducted under various experiment settings. Those experiments have shown that we can reduce the error by at least one magnitude. We have proposed a software-based solution to solve the correlations brought by the shortage of hashes. However, future hardware-based solutions, such as providing more independent hash functions, are valuable for more flexible network designs.

To solve the limited memory in switches, we employed weighted sample-based algorithms: a flexible approach to stream summarization, whose outputs can be readily utilized by downstream applications for queries on ranks and sub-populations. Our approach comprises a new set of algorithms, Priority-Based Aggregation and its variants) of this type. PBA is designed around a single random variable per key aggregate, allowing considerable speed-up in a fixed cache, and it also improves accuracy for a given sample size, compared with state-of-the-art methods.

In addition to sampling, a large number of network functions for network monitoring raises the classification rules. We offered *CO2*, which allows approximate classification at switches to reduce memory usage by introducing a small amount of overselected traffic. We can then filter the overselected traffic at receivers with a little CPU overhead. Our extensive simulations with real traffic traces and rules show that Cuckoo filters needed 56% more memory than *CO2* to achieve the equivalent false positive rate. We also demonstrate the feasibility of *CO2* design with prototypes using Open vSwitch and P4. Although the *CO2* design introduces additional complexity, we

believe it will be feasible to implement with the trend of programmable switches. For example, the P4 specification for programmable switches [109] already supports a pipeline of match-action tables and has been implemented by multiple switch vendors.

REFERENCES

[1] N. Duffield, Y. Xu, L. Xia, N. K. Ahmed, and M. Yu, "Stream aggregation through order sampling," in *Proceedings of the 2017 ACM on Conference on Information and Knowledge Management*, pp. 909–918, ACM, 2017. doi: https://dl.acm.org/doi/10.1145/3132847.3133042.

[2] C. Hopps, "Analysis of an equal-cost multi-path algorithm," RFC 2992, RFC Editor, November 2000.

[3] J. Zhou, M. Tewari, M. Zhu, A. Kabbani, L. Poutievski, A. Singh, and A. Vahdat, "Wcmp: Weighted cost multipathing for improved fairness in data centers," in *Proceedings of the Ninth European Conference on Computer Systems*, p. 5, ACM, 2014.

[4] R. Zhang-Shen and N. McKeown, "Designing a predictable internet backbone with valiant load-balancing," *Quality of Service–IWQoS 2005*, pp. 178–192, 2005.

[5] E. Vanini, R. Pan, M. Alizadeh, P. Taheri, and T. Edsall, "Let it flow: Resilient asymmetric load balancing with flowlet switching.," in *NSDI*, pp. 407–420, 2017.

[6] V. R. K. Addanki, "Method and system for management of flood traffic over multiple 0: N link aggregation groups," Apr. 22 2014. US Patent 8,705,551.

[7] A. Hendel, "Mutable hash for network hash polarization," Mar. 19 2015. US Patent App. 14/026,725.

[8] Cisco, "Cef polarization." https://www.cisco.com/c/en/us/support/docs/ip/express-forwarding-cef/116376-technote-cef-00.html.

[9] Dell, "Dell networking configuration guide for the mxl 10/40gbe switch i/o module 9.9(0.0)." http://www.dell.com/support/manuals.

[10] S. Jain, A. Kumar, S. Mandal, J. Ong, L. Poutievski, A. Singh, S. Venkata, J. Wanderer, J. Zhou, M. Zhu, *et al.*, "B4: Experience with a globally-deployed software defined wan," *ACM SIGCOMM Computer Communication Review*, vol. 43, no. 4, pp. 3–14, 2013.

[11] A. Roy, H. Zeng, J. Bagga, and A. C. Snoeren, "Passive realtime datacenter fault detection and localization," in *14th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 17)*, pp. 595–612, 2017.

[12] A. Roy, R. Das, H. Zeng, J. Bagga, and A. C. Snoeren, "Understanding the limits of passive realtime datacenter fault detection and localization," *IEEE/ACM Transactions on Networking*, 2019.

[13] Y. Li, R. Miao, C. Kim, and M. Yu, "Lossradar: Fast detection of lost packets in data center networks," in *Proceedings of the 12th International on Conference on emerging Networking EXperiments and Technologies*, pp. 481–495, ACM, 2016.

[14] C. Tan, Z. Jin, C. Guo, T. Zhang, H. Wu, K. Deng, D. Bi, and D. Xiang, "Netbouncer: Active device and link failure localization in data center networks.," in *NSDI*, pp. 599–614, 2019.

[15] C. Yu, C. Lumezanu, A. Sharma, Q. Xu, G. Jiang, and H. V. Madhyastha, "Software-defined latency monitoring in data center networks," in *International Conference on Passive and Active Network Measurement*, pp. 360–372, Springer, 2015.

[16] R. Khan, R. Bolla, M. Repetto, R. Bruschi, and M. Giribaldi, "Smart proxying for reducing network energy consumption," in *2012 International Symposium on Performance Evaluation of Computer & Telecommunication Systems (SPECTS)*, pp. 1–8, IEEE, 2012.

[17] Dell, "Dell configuration guide for the s4048-on system 9.9(0.0)." https://www.dell.com/support/manuals/us/en/19/force10-s4048-on/s4048_on_9.9.0.0_config_pub-v1/rtag7?guid=guid-9bda04b4-e966-43c7-a8cf-f01e7ce600f4&lang=en-us.

[18] Cisco, "Data center access design with cisco nexus 5000 series switches and 2000 series fabric extenders and virtual portchannels.." https://itnetworkingpros.files.wordpress.com/2014/04/c07-572829-01_design_n5k_n2k_vpc_dg.pdf.

[19] Y. Huo, X. Li, W. Wang, and D. Liu, "High performance table-based architecture for parallel crc calculation," in *The 21st IEEE International Workshop on Local and Metropolitan Area Networks*, pp. 1–6, IEEE, 2015.

[20] A. Joglekar, M. E. Kounavis, and F. L. Berry, "A scalable and high performance software iscsi implementation.," in *FAST*, vol. 5, pp. 267–280, 2005.

[21] E. Stavinov, "A practical parallel crc generation method," *Circuit Cellar-The Magazine For Computer Applications*, vol. 31, no. 234, p. 38, 2010.

[22] J. Zhou and Z. Ji, "Hashing technique to optimally balance load within switching networks," 2017.

[23] Netgear, "What is a virtual lan (vlan) and how does it work with my managed switch?." https://kb.netgear.com/21574/What-is-a-virtual-LAN-VLAN-and-how-does-it-work-with-my-managed-switch.

[24] X. Wu, D. Turner, C.-C. Chen, D. A. Maltz, X. Yang, L. Yuan, and M. Zhang, "Netpilot: automating datacenter network failure mitigation," *ACM SIGCOMM Computer Communication Review*, vol. 42, no. 4, pp. 419–430, 2012.

[25] M. Al-Fares, S. Radhakrishnan, B. Raghavan, N. Huang, and A. Vahdat, "Hedera: Dynamic flow scheduling for data center networks.," in *NSDI*, vol. 10, pp. 19–19, 2010.

[26] K. He, E. Rozner, K. Agarwal, W. Felter, J. Carter, and A. Akella, "Presto: Edge-based load balancing for fast datacenter networks," in *ACM SIGCOMM Computer Communication Review*, vol. 45, pp. 465–478, ACM, 2015.

[27] N. McKeown, "Software-defined networking," *INFOCOM keynote talk*, vol. 17, no. 2, pp. 30–32, 2009.

[28] Facebook, "Broadcom's openflow data plane abstraction.." https://www.broadcom.com/products/ethernet-connectivity/software.

[29] M. Alizadeh, T. Edsall, S. Dharmapurikar, R. Vaidyanathan, K. Chu, A. Fingerhut, F. Matus, R. Pan, N. Yadav, G. Varghese, *et al.*, "Conga: Distributed congestion-aware load balancing for datacenters," in *ACM SIGCOMM Computer Communication Review*, vol. 44, pp. 503–514, ACM, 2014.

[30] N. G. Duffield and B. Krishnamurthy, "Efficient sampling for better OSN data provisioning," *CoRR*, vol. abs/1612.04666, 2016.

[31] C. Lo, D. Frankowski, and J. Leskovec, "Understanding behaviors that lead to purchasing: A case study of pinterest," in *Proceedings of the 22Nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD '16, (New York, NY, USA), pp. 531–540, ACM, 2016.

[32] X. Liu, L. Golab, W. Golab, I. F. Ilyas, and S. Jin, "Smart meter data analytics: Systems, algorithms, and benchmarking," *ACM Trans. Database Syst.*, vol. 42, pp. 2:1–2:39, Nov. 2016.

[33] M. Yu, L. Jose, and R. Miao, "Software defined traffic measurement with opensketch," in *Proceedings of the 10th USENIX Conference on Networked Systems Design and Implementation*, nsdi'13, (Berkeley, CA, USA), pp. 29–42, USENIX Association, 2013.

[34] "NetFlow." `https://www.ietf.org/rfc/rfc3954.txt`.

[35] A. Roy, H. Zeng, J. Bagga, G. Porter, and A. C. Snoeren, "Inside the social network's (datacenter) network," in *ACM SIGCOMM Computer Communication Review*, vol. 45, pp. 123–137, ACM, 2015.

[36] Y. Li, R. Miao, C. Kim, and M. Yu, "Flowradar: a better netflow for data centers," in *13th USENIX Symposium on Networked Systems Design and Implementation (NSDI 16)*, pp. 311–324, USENIX Association, 2016.

[37] A. Andreyev, "Introducing data center fabric, the next-generation Facebook data center network," 2014. https://code.facebook.com/posts/360346274145943/introducing-data-center-fabric-the-next-generation-facebook-data-cent

[38] C. Cranor, T. Johnson, O. Spatscheck, and V. ladislav Shkapenyuk, "Gigascope: A stream database for network applications," in *Proc ACM SIGMOD*, June 2003.

[39] M. Moshref, M. Yu, R. Govindan, and A. Vahdat, "Scream: Sketch resource allocation for software-defined measurement," in *Proceedings of the 11th ACM Conference on Emerging Networking Experiments and Technologies*, CoNEXT '15, (New York, NY, USA), pp. 14:1–14:13, ACM, 2015.

[40] J. Vitter, "Random sampling with a reservoir," *ACM Trans. Math. Softw.*, vol. 11, 1985.

[41] P. Gibbons and Y. Matias, "New sampling-based summary statistics for improving approximate query answers," in *SIGMOD*, ACM, 1998.

[42] C. Estan and G. Varghese, "New directions in traffic measurement and accounting," in *Proc. of SIGCOMM*, pp. 323–336, 2002.

[43] E. Cohen and H. Kaplan, "Summarizing data using bottom-k sketches," in *Proceedings of the ACM PODC'07 Conference*, 2007.

[44] E. Cohen and H. Kaplan, "Tighter estimation using bottom-k sketches," in *Proceedings of the 34th VLDB Conference*, 2008.

[45] B. Rosén, "Asymptotic theory for successive sampling with varying probabilities without replacement, I," *The Annals of Mathematical Statistics*, vol. 43, no. 2, pp. 373–397, 1972.

[46] E. Cohen, "Stream sampling for frequency cap statistics," in *Proceedings of the 21th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD '15, (New York, NY, USA), pp. 159–168, ACM, 2015.

[47] E. Cohen, G. Cormode, and N. Duffield, "Don't let the negatives bring you down: sampling from streams of signed updates," *In SIGMETRICS*, vol. 40, no. 1, pp. 343–354, 2012.

[48] E. Cohen, N. Duffield, H. Kaplan, C. Lund, and M. Thorup, "Algorithms and estimators for accurate summarization of internet traffic," in *Proceedings of the 7th ACM SIGCOMM Conference on Internet Measurement*, IMC '07, (New York, NY, USA), pp. 265–278, ACM, 2007.

[49] J. Hájek, "Limiting distributions in simple random sampling from a finite population," *Publications of Mathematical Institute of Hungarian Academy of Sciences, Series A*, vol. 5, pp. 361–374, 1960.

[50] B. Rosén, "Asymptotic theory for order sampling," *J. Statistical Planning and Inference*, vol. 62, no. 2, pp. 135–158, 1997.

[51] N. Duffield, M. Thorup, and C. Lund, "Priority sampling for estimating arbitrary subset sums," *J. Assoc. Comput. Mach.*, vol. 54, no. 6, 2007.

[52] P. S. Efraimidis and P. G. Spirakis, "Weighted random sampling with a reservoir," *Inf. Process. Lett.*, vol. 97, no. 5, pp. 181–185, 2006.

[53] N. Alon, Y. Matias, and M. Szegedy, "The space complexity of approximating the frequency moments," *Journal of Computer and System Sciences*, vol. 58, no. 1, pp. 137–147, 1999.

[54] P. Indyk, "Stable distributions, pseudorandom generators, embeddings and data stream computation," in *Proc. of the 41st Symposium on Foundations of Computer Science*, 2000.

[55] W. B. Johnson, J. Lindenstrauss, and G. Schechtman, "Extensions of lipschitz maps into banach spaces," *Israel Journal of Mathematics*, vol. 54, no. 2, pp. 129–138, 1986.

[56] G. Cormode and S. Muthukrishnan, "An improved data stream summary: The count-min sketch and its applications," *J. Algorithms*, vol. 55, pp. 29–38, 2004.

[57] A. C. Gilbert, S. Guha, P. Indyk, Y. Kotidis, S. Muthukrishnan, and M. J. Strauss, "Fast, small-space algorithms for approximate histogram maintenance," in *Proceedings of the Thiry-fourth Annual ACM Symposium on Theory of Computing*, STOC '02, (New York, NY, USA), pp. 389–398, ACM, 2002.

[58] A. Andoni, R. Krauthgamer, and K. Onak, "Streaming algorithms from precision sampling," Tech. Rep. 1011.1263, arXiv, 2010.

[59] M. Monemizadeh and D. P. Woodruff, "1-pass relative-error lp-sampling with applications," in *Proc. 21st ACM-SIAM Symposium on Discrete Algorithms*, ACM-SIAM, 2010.

[60] H. Jowhari, M. Saglam, and G. Tardos, "Tight bounds for Lp samplers, finding duplicates in streams, and related problems," in *PODS*, pp. 49–58, 2011.

[61] V. Braverman and R. Ostrovsky, "Zero-one frequency laws," in *Proceedings of the Forty-second ACM Symposium on Theory of Computing*, STOC '10, (New York, NY, USA), pp. 281–290, ACM, 2010.

[62] Z. Liu, A. Manousis, G. Vorsanger, V. Sekar, and V. Braverman, "One sketch to rule them all: Rethinking network flow monitoring with univmon," in *Proceedings of the 2016 Conference on ACM SIGCOMM 2016 Conference*, SIGCOMM '16, (New York, NY, USA), pp. 101–114, ACM, 2016.

[63] E. Cohen, "Stream sampling for frequency cap statistics," in *Proceedings of the 21th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, pp. 159–168, ACM, 2015.

[64] K. Keys, D. Moore, and C. Estan, "A robust system for accurate real-time summaries of internet traffic," *ACM SIGMETRICS Performance Evaluation Review*, vol. 33, 2005.

[65] S. Palkar and e. Lan, "E2: a framework for nfv applications," in *SOSP*, ACM, 2015.

[66] Z. A. Qazi and e. Tu, "Simple-fying middlebox policy enforcement using sdn," vol. 43, ACM, 2013.

[67] P. Patel, D. Bansal, and etc., "Ananta: Cloud Scale Load Balancing," in *SIGCOMM*, 2013.

[68] Y. Zhu and N. K. etc., "Packet-level telemetry in large datacenter networks," in *SIGCOMM*, 2015.

[69] J. Sherry and e. Hasan, "Making middleboxes someone else's problem: network processing as a cloud service," *SIGCOMM*, vol. 42, no. 4, 2012.

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

[71] P. Bodík, I. Menache, and etc., "Surviving failures in bandwidth-constrained datacenters," in *SIGCOMM*, 2012.

[72] "http://blog.ipspace.net/2014/06/trident-2-chipset-and-nexus-9500.html."

[73] "P4 language consortium." `p4.org`.

[74] Q. Dong, S. Banerjee, J. Wang, and D. Agrawal, "Wire speed packet classification without tcams: A few more registers (and a bit of logic) are enough," *SIGMETRICS Perform.*, 2007.

[75] O. Rottenstreich and J. Tapolcai, "Lossy compression of packet classifiers," in *ANCS*, May 2015.

[76] A. X. Liu, C. R. Meiners, and E. Torng, "Tcam razor: A systematic approach towards minimizing packet classifiers in tcams," *IEEE/ACM Trans. Netw.*, 2010.

[77] B. Fan, D. G. Andersen, M. Kaminsky, and M. D. Mitzenmacher, "Cuckoo filter: Practically better than bloom," CoNEXT, 2014.

[78] B. Chazelle, J. Kilian, and etc., "The bloomier filter: An efficient data structure for static support lookup tables," SIAM, 2004.

[79] Y. Wang and e. Pan, "Namefilter: Achieving fast name lookup with low memory cost via applying two-stage bloom filters," in *INFOCOM*, 2013.

[80] H. Lim, B. Fan, D. G. Andersen, and M. Kaminsky, "Silt: A memory-efficient, high-performance key-value store," SOSP, ACM, 2011.

[81] H. Song, S. Dharmapurikar, and etc., "Fast hash table lookup using extended bloom filter: An aid to network processing," SIGCOMM, 2005.

[82] L. Mchale and e. Case, "Stochastic pre-classification for sdn data plane matching," ICNP, IEEE Computer Society, 2014.

[83] X. Zhao, Y. Liu, L. Wang, and B. Zhang, "On the aggregatability of router forwarding tables," in *INFOCOM*, March 2010.

[84] Y. Liu, B. Zhang, and L. Wang, "Fifa: Fast incremental fib aggregation," in *INFOCOM, 2013 Proceedings IEEE*, April 2013.

[85] C. Meiners, A. Liu, and E. Torng, "Bit weaving: A non-prefix approach to compressing packet classifiers in tcams," *Networking, IEEE/ACM Transactions on*, 2012.

[86] B. Everitt and A. Skrondal, *The Cambridge dictionary of statistics*, vol. 44. Cambridge University Press Cambridge, 2002.

[87] M. Al-Fares, A. Loukissas, and A. Vahdat, "A scalable, commodity data center network architecture," in *ACM SIGCOMM Computer Communication Review*, vol. 38, pp. 63–74, ACM, 2008.

[88] A. Singh, J. Ong, A. Agarwal, G. Anderson, A. Armistead, R. Bannon, S. Boving, G. Desai, B. Felderman, P. Germano, *et al.*, "Jupiter rising: A decade of clos topologies and centralized control in google's datacenter network," in *ACM SIGCOMM computer communication review*, vol. 45, pp. 183–197, ACM, 2015.

[89] Wikipedia, "F1 score." https://en.wikipedia.org/wiki/F1_score.

[90] Wikipedia, "Precision and recall." https://en.wikipedia.org/wiki/Precision_and_recall.

[91] Caida, "Caida's passive traces dataset." http://www.caida.org/data/passive/passive_dataset_download.xm

[92] N. Spring, R. Mahajan, and D. Wetherall, "Measuring isp topologies with rocketfuel," in *ACM SIGCOMM Computer Communication Review*, vol. 32, pp. 133–145, ACM, 2002.

[93] J. Y. Yen, "An algorithm for finding shortest routes from all source nodes to a given destination in general networks," *Quarterly of Applied Mathematics*, vol. 27, no. 4, pp. 526–530, 1970.

[94] D. Williams, *Probability with Martingales*. Cambridge University Press, 1991.

[95] M. Pătraşcu and M. Thorup, "The power of simple tabulation hashing," *J. ACM*, vol. 59, pp. 14:1–14:50, June 2012.

[96] A. Metwally, D. Agrawal, and A. El Abbadi, "Efficient computation of frequent and top-k elements in data streams," in *Proceedings of the 10th International Conference on Database Theory*, ICDT'05, (Berlin, Heidelberg), pp. 398–412, Springer-Verlag, 2005.

[97] A. Feldmann, J. Rexford, and R. Caceres, "Efficient policies for carrying web traffic over flow-switched networks," *IEEE/ACM Transactions on Networking*, pp. 673–685, December 1998.

[98] A. Singh, J. Ong, and etc., "Jupiter rising: A decade of clos topologies and centralized control in google's datacenter network," *SIGCOMM*, 2015.

[99] S. Dharmapurikar, P. Krishnamurthy, and D. E. Taylor, "Longest prefix matching using bloom filters," in *SIGCOMM*, 2003.

[100] G. Cormode and S. Muthukrishnan, "An improved data stream summary: The count-min sketch and its applications," *J. Algorithms*, vol. 55, 2005.

[101] T. R. Puzak, "Analysis of cache replacement-algorithms," 1985.

[102] Y. Yu, C. Qian, and X. Li, "Distributed and collaborative traffic monitoring in software defined networks," in *Proceedings of the third workshop on Hot topics in software defined networking*, pp. 85–90, ACM, 2014.

[103] P. E. Black, "Dictionary of algorithms and data structures," tech. rep., 1998.

[104] D. E. Taylor and J. S. Turner, "ClassBench: A Packet Classification Benchmark," *Transactions on Networking*, vol. 15, no. 3, 2007.

[105] Wikipedia, "Settling time."

[106] "Open vSwitch.." `http://www.openvswitch.org/`.

[107] "Mininet.." `http://www.mininet.org/`.

[108] "The Netperf Homepage.." `http://www.netperf.org/`.

[109] P. Bosshart, D. Daly, and etc., "P4: Programming protocol-independent packet processors," *SIGCOMM*, 2014.

[110] M. Al-Fares, A. Loukissas, and A. Vahdat, "A scalable, commodity data center network architecture," *SIGCOMM Comput. Commun. Rev.*, vol. 38, Aug. 2008.

[111] T. Zink and M. Waldvogel, "Efficient hash tables for network applications," *SpringerPlus*, vol. 4, 2015.

[112] N. Handigol, B. Heller, and etc., "I know what your packet did last hop: Using packet histories to troubleshoot networks," in *NSDI*, 2014.

[113] A. Lakhina, M. Crovella, and C. Diot, "Diagnosing network-wide traffic anomalies," SIGCOMM '04, ACM, 2004.

[114] M. Moshref, M. Yu, and etc., "Dream: dynamic resource allocation for software-defined measurement," *SIGCOMM*, 2015.

[115] G. Cormode and e. Korn, "Finding hierarchical heavy hitters in data streams," in *VLDB*, 2003.

[116] G. Cormode and S. Muthukrishnan, "An improved data stream summary: the count-min sketch and its applications," *Journal of Algorithms*, vol. 55, no. 1, pp. 58–75, 2005.

APPENDIX A

FIRST APPENDIX

A.1 Upper Bound for p-values

of Theorem **??**.

$$\begin{aligned} p_i &= P(\widehat{Y}_i \geq \widehat{X}_i) = P(e^{\theta\widehat{Y}_i} \geq e^{\theta\widehat{X}_i}) \\ &\leq E(e^{\theta\widehat{Y}_i})e^{-\theta\widehat{X}_i} \leq \min_{\theta>0} e^{-\theta\widehat{X}_i} \prod_{j}^{M} E(e^{\theta(x_j=i)/M}) \\ &\leq \min_{\theta>0} e^{-\theta\widehat{X}_i}\left(\frac{1}{L}e^{\theta/M} + \frac{L-1}{L}\right)^M \end{aligned} \tag{A.1}$$

where θ is an arbitrary positive number.

By taking the derivative of function $e^{-\theta\widehat{X}_i}(\frac{1}{L}e^{\theta/M} + \frac{L-1}{L})^M$, we obtain

$$\theta = M\log\left(\frac{(L-1)\widehat{X}_i}{1-\widehat{X}_i}\right) \tag{A.2}$$

By substituting θ with the right part of (A.2), we get the upper bound in (**??**).

□

A.2 Theorem Concerning CRC

of Theorem 1. To prove Theorem 1, we only need to prove $crc_i(x)\%2 = crc_i(y)\%2$ leads to $crc_i(x \oplus z_1)\%2 = crc_i(y \oplus z_1)\%2$, and $crc_i(x)\%2 = crc_i(y)\%2$ leads to $crc_i(x \oplus z_2)\%2 = crc_i(y \oplus z_2)\%2$.

Let $z_1 = crc_i(t)$, based on the rolling property of the CRC function, we get

$$crc_i(x \oplus z_1) = crc_i((t \ll b_x)|x) = crc_i(t \ll b_x) \oplus crc_i(x)$$

where b_x is the binary length of x.

We also have,

$$crc_i(y \oplus z_1) = crc_i(t \ll b_x) \oplus crc_i(y)$$

If $crc_i(x)$ and $crc_i(y)$ are both even (odd), $crc_i(t \ll b_x) \oplus crc_i(x)$ and $crc_i(t \ll b_x) \oplus crc_i(y)$ are both even or odd, that is, $crc_i(x \oplus z_1)$ and $crc_i(y \oplus z_1)$ are both even or odd.

It is the same for $crc_i(x \oplus z_2)\%2 = crc_i(y \oplus z_2)\%2$

□

A.3 Theorem Concerning Coprimes

of Theorem 2. Let U, W, U' and W' denote the distribution of the hashing output of $H\%m_1$, $H\%m_2$, $H\%p_1\%m_1$ and $H\%p_2\%m_2$, respectively. When $\widehat{H} \gg q_1q_2$, $\forall i' \in [0, q_1 - 1]$, $j' \in [0, q_2 - 1]$, $i \in [0, m_1 - 1]$, and $j \in [0, m_2 - 1]$, we have,

$$P(U' = i', W' \approx j') = \frac{1}{q_1q_2}$$

When condition 1 is not satisfied in Theorem 2, there exists hash correlation brought by $q_1\%m_1$ and $q_2\%m_2$ because the mapping from $q_1(q_2)$ to $m_1(m_2)$ has a remainder. This mapping causes the non-uniform distribution of q_1 integers over m_1 slots, where $q_1\%m_1$ slots get $q_1/m_1 + 1$ integers, and $m_1 - q_1\%m_1$ slots get q_1/m_1 integers. We denote this non-uniformity by the term *the approximation error*. Note that when q_1 increases, the difference between $(q_1/m_1 + 1)/q_1$ and $(q_1/m_1)/q_1$ can be reduced, and thus, we can quantify the approximation error by Equation A.3.

$$err = (q_1\%m_1)/q_1 + (q_2\%m_2)/q_2 \tag{A.3}$$

where err denotes the approximation error.

When condition 1 is satisfied, $err \approx 0$. Let $S(i) = \{i'|i'\%i = 0\}$, and $S(j) = \{j'|j'\%j = 0\}$, we have,

$$
\begin{aligned}
P(W=j|U=i) &= \frac{\sum_{i' \in S(i), j' \in S(j)} P(U'=i', W'=j')}{\sum_{i' \in S(i)} P(U'=i')} \\
&= \frac{(q_1/m_1)(q_2/m_2)(1/(q_1 q_2))}{(q_1/m_1)(1/q_1)} \\
&= 1/m_2 = P(W=j)
\end{aligned} \tag{A.4}
$$

uncorrelated criterion achieved.

□

A.4 Theorem Concerning WCMP Mapper

For simplicity, we only prove the case when there are two egress ports in the WCMP group.

of Theorem 3. We introduce the variables used in this proof. Suppose the integer weights of those two ports are w_1 and w_2, and let $W = w_1 + w_2$. q is the Coprime value chosen for this WCMP group. Let w_1^e, w_2^e, w_1^w and w_2^w as the normalized weights generated by applying ECMP mapper and WCMP mapper. And $r = q \, mod \, W$.

If we can prove $(w_1/W - w_1^e)^2 \geq (w_1/W - w_1^w)^2$, and $(w_2^e - w_2/W)^2 \geq (w_2^w - w_2/W)^2$, we can prove that WCMP mapper achieves a smaller K-S statistic than ECMP mapper.

$$
w_1^e = \frac{w_1 * q/W}{q}
$$

$$
w_2^e = \frac{w_2 * q/W + r}{q}
$$

$$
w_1^w = \frac{w_1 * q/W + r/2}{q}
$$

$$
w_2^w = \frac{w_2 * q/W + r/2 + (r \neq 0)}{q}
$$

$$(w_1/W - w_1^e)^2 = (\frac{w_1 * q/W - w_1 * q/W}{q})^2$$

$$(w_1/W - w_1^w)^2 = (\frac{w_1 * q/W - w_1 * q/W - r/2}{q})^2$$

Let $e_1 = \frac{q^2}{w_1^2} * ((w_1/W - w_1^e)^2 - (w_1/W - w_1^w)^2)$, we have

$$\begin{aligned} e_1 &= (q/W - q/W)^2 - (q/W - q/W - r/2)^2 \\ &= \{2(q/W - q/W) - r/2\} * r/2 \\ &= (2r - r/2) * r/2 \geq 0 \end{aligned}$$

$$(w_2^e - w_2/W)^2 = (\frac{w_2 * q/W + r - w_2 * q/W}{q})^2$$

$$(w_2^w - w_2/W)^2 = (\frac{w_2 * q/W + r/2 + (r \neq 0) - w_2 * q/W}{q})^2$$

Let $e_2 = \frac{q^2}{w_2^2} * ((w_2^e - w_2/W)^2 - (w_2^w - w_2/W)^2)$, we have

$$\begin{aligned} e_2 &= (q/W + w_2 r - q/W)^2 \\ &- (q/W + w_2 r/2 + w_2(r \neq 0) - q/W)^2 \\ &= 2bw_2(r - (r/2 + (r \neq 0))) + w_2^2(r^2 - (r/2 + (r \neq 0))^2) \end{aligned}$$

Since $r \geq r/2 + (r \neq 0)$, $e_2 \geq 0$.

Done. □