Efficient Traffic Splitting on Commodity Switches

Nanxi Kang
Princeton University
nkang@cs.princeton.edu

Monia Ghobadi
Microsoft Research
mgh@microsoft.com

John Reumann
NofutzNetworks Inc.
reumann@nofutznetworks.com

Alexander Shraer
Google
shralex@google.com

Jennifer Rexford
Princeton University
jrex@cs.princeton.edu

ABSTRACT

Traffic often needs to be split over multiple equivalent backend servers, links, paths, or middleboxes. For example, in a load-balancing system, switches distribute requests of online services to backend servers. Hash-based approaches like Equal-Cost Multi-Path (ECMP) have low accuracy due to hash collision and incur significant churn during update. In a Software-Defined Network (SDN) the accuracy of traffic splits can be improved by crafting a set of wildcard rules for switches that better match the actual traffic distribution. The drawback of existing SDN-based traffic-splitting solutions is poor scalability as they generate too many rules for small rule-tables on switches. In this paper, we propose Niagara, an SDN-based traffic-splitting scheme that achieves accurate traffic splits while being extremely efficient in the use of rule-table space available on commodity switches. Niagara uses an incremental update strategy to minimize the traffic churn given an update. Experiments demonstrate that Niagara (1) achieves nearly optimal accuracy using only $1.2\% - 37\%$ of the rule space of the current state-of-art, (2) scales to tens of thousands of services with the constrained rule-table capacity and (3) offers nearly minimum churn.

1. INTRODUCTION

Network operators often spread traffic over multiple components (such as links, paths, and backend servers) that offer the same functionality or service, to achieve better scalability, reliability, and performance. Managing these distributed resources effectively requires a good way to balance the traffic load, especially when different components have different capacity. Rather than deploying dedicated load-balancing appliances, modern networks increasingly rely on the underlying *switches* to split load across the replicas [1–8]. For example, server load-balancing systems [4, 6] use hardware switches to spread client requests for each service over multiple software load balancers, which in turn direct requests to backend servers. Another example is multi-pathing [3, 9, 10], where a switch splits the flows with the same destination over multiple paths.

The most common traffic-splitting mechanism is Equal-Cost Multi-Path (ECMP) [9, 10], which is available in most commodity switches and widely used for load balancing [4, 6] and multi-pathing purposes [1, 2]. ECMP splits a set of flows (typically flows with the same destination prefix) *uniformly* over a group of next-hops based on the hash values of the packet-header fields. Weighted-Cost Multi-Path (WCMP) [3] is an extension of ECMP that supports a weighted splits by repeating the same next-hop multiple times in an ECMP group. ECMP and WCMP both partition the flow space, assuming equal traffic load in each hash bucket. The splitting accuracy of ECMP degrades significantly due to hash collision [11, 12]. Furthermore, ECMP incurs unnecessary traffic shifts during updates. When a next-hop is added or removed in ECMP, any hash function shifts at least 25% to 50% of the flow space to a different next-hop [10].

In this paper, we are interested in designing a generic and accurate traffic-splitting scheme for commodity switches. The emergence of open interfaces to commodity SDN switches such as OpenFlow [13, 14], enables operators to have a controller that installs rules on switch rule-tables to satisfy the load-balancing goals [5, 12, 15]. These rule-tables (*e.g.,* TCAM) are optimized for high-speed packet-header matching, however they have small capacities on the order of a few thousand entries [16–18]. The simplest SDN-based solution [15] directs the first packet of each flow to the controller, which reactively installs an exact-match (microflow) rule on the switch. More efficient approaches [5, 12, 19, 20] proactively install wildcard rules that direct packets matching the same header patterns to the same next-hop, but they do not use the rule-table space efficiently and cannot scale to large networks.

This paper presents Niagara, an efficient traffic-splitting scheme that computes switch rules to minimize traffic *im-*

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

CoNEXT '15 December 01-04, 2015, Heidelberg, Germany
© 2015 ACM. ISBN 978-1-4503-3412-9/15/12...$15.00
DOI: http://dx.doi.org/10.1145/2716281.2836091

balance (*i.e.,* the fraction of traffic sent to the "wrong" next-hop, based on the target load balancing weights), subject to rule-table constraints. Niagara handles multiple *flow aggregates*—sets of flows with the same destination or egress. Each flow aggregate is splitted according to distinct target weights. Our experiments demonstrate that Niagara scales to tens of thousands of flow aggregates and hundreds of next-hops with a small imbalance. After a brief discussion of traffic-splitting use cases and related work (Section 2), we present the traffic-splitting optimization problem and a high-level overview of Niagara (Section 3).

We make the following contributions.

Efficient traffic-splitting algorithm: Niagara approximates load-balancing weights accurately with a small number of wildcard rules. For each flow aggregate, Niagara can flexibly trade off accuracy for fewer rules (Section 4). Niagara packs rules for multiple flow aggregates into a single table, and allows sharing of rules across multiple aggregates with similar weights (Section 5). Given an update, Niagara computes incremental changes to the rules to minimize churn (*i.e.,* the fraction of traffic shuffled to a different next-hop due to the update) and traffic imbalance (Section 6).

Realistic prototype: We implement the Niagara OpenFlow controller and deploy the controller (i) in a physical testbed with a hardware Pica8 switch interconnecting four hosts and (ii) in Mininet [21] with Open vSwitches [22] and a configurable number of hosts. We recently conducted a live demonstration of Niagara at an SDN-based Internet eXchange Point (IXP) in New Zealand [23], where Niagara load balanced DNS and web requests to backend servers in a production environment.

Trace-driven large-scale evaluation: We evaluate the performance of Niagara for server load balancing and multi-path traffic splitting through extensive simulation against real and synthetic data and validate the simulation results subject to the limitations of our prototype (Section 7). Experiments demonstrate that Niagara (1) achieves nearly optimal accuracy outperforming ECMP and other SDN-based approaches, (2) scales to tens of thousands of aggregates using as little as $1.2\% - 37\%$ of the rule space compared to alternative solutions and (3) handles update gracefully with nearly minimal churn.

2. TRAFFIC SPLIT BACKGROUND

2.1 Use cases

We provide three examples that illustrate how hardware switches are used to split traffic over next-hops.

Server load balancing. Cloud providers host many services, each replicated on multiple servers for greater throughput and reliability. Load balancers (*e.g.,* Ananta [4], Duet [6]) rely on hardware switches to spread service requests over servers. Ananta uses switches to forward requests over software load balancers (SLB), which then send requests to backends; Duet requires switches to distribute requests to backends for popular services directly, besides forwarding to SLBs. Depending on the server capacity and deployments (*e.g.,* server allocation in racks, maintenance and failures), a switch is required to spread requests evenly or in a weighted fashion [5, 6]. Both Ananta and Duet use hash-based traffic-splitting schemes (Section 2.3).

Data center multi-pathing. Data center topologies [1–3] offer many equal-length paths that switches can use to increase bisection bandwidth. In a fully symmetric topology, a switch splits traffic of each destination prefix equally over available paths. A recent study [3] found that data-center topologies tend to be asymmetric due to failures and heterogeneous devices. In such a topology, a switch should split traffic in proportion to the capacity of the equal-length paths.

Wide area traffic engineering. Wide Area Networks (WAN) carry a huge amount of inter-datacenter traffic. WAN traffic engineering systems (TE) establish tunnels among data center sites and run periodic algorithms to optimize the bandwidth allocation of tunnels to different applications. The underlying switches should split traffic for each application over tunnels according to the algorithm's results for the best network utilization. Existing TE solutions (*e.g.,* SWAN [24], B4 [25]) use hash-based approaches as their default traffic-splitting schemes (Section 2.3).

2.2 Requirements

Accuracy. Traffic-splitting schemes should be accurate. Commodity servers can handle a limited number of requests; an inaccurate traffic split can easily overload a server, thus incurring long latencies and request failures. In the network, inaccurate splits create congestion and packet loss.

Scalability. The scheme should scale. Data centers host up to tens of thousands of services (*i.e.,* flow aggregates), which are collectively handled by a handful of SLBs (*i.e.,* next-hops); multi-path routing requires an ingress switch to handle hundreds of destination prefixes (*i.e.,* flow aggregates) and dozens of paths (*i.e.,* next-hops). A scalable traffic-splitting scheme should handle the heterogeneity in the numbers of flow aggregates and next-hops, given the constraints in rule-table capacity.

Update efficiency. Failures or changes in capacity require updating the split of flow aggregates. However, transitioning to this new split comes at some cost of reshuffling packets among servers (*i.e.,* churn). This requires extra work to ensure consistent handling of TCP connections already in progress [4, 26, 27]. A good traffic-splitting scheme needs to be updatable with limited churn.

2.3 Prior Traffic-Splitting Schemes

Hash-based approaches. ECMP aims at an equal split over a group of next-hops (*e.g.,* SLBs) by partitioning the flow space into equal-sized hash-buckets, each of which corresponds to one next-hop. WCMP handles weighted splits by repeating next-hops in an ECMP group, thus assigning multiple hash-buckets to the same next-hop. ECMP is available on most commodity switches, which gives rise to its popularity [4, 6, 24, 25]. However, it splits the *flow space* equally, rather than the actual traffic. It is common that certain parts of the flow spaces (*e.g.,* a busy source) contribute more traffic than others [1, 11, 28]; an even partition of the flow space

does not guarantee the equal split of traffic. Moreover, the size of the ECMP table, which is a TCAM with hundreds to thousands rules on commodity switches [3], severely restricts the achievable accuracy of WCMP. Finally, updating an ECMP group unnecessarily shuffles packets among next-hops. It is shown that when a next-hop is added to a $N-1$-member group, at least $\frac{1}{4}+\frac{1}{4N}$ of the flow space are shuffled to different next-hops [10], while the minimum shuffle is $\frac{1}{N}$.

SDN-based approaches. SDN supports programming rule-tables in switches, enabling finer-grained control and more accurate splitting. Aster*x [15] directs the first packet of each flow to a controller, which then installs micro-flow rules for forwarding the remaining packets, making the controller load and hardware rule-table capacity quickly become bottlenecks. MicroTE [12] proactively decides routing for every pair of edge switches (*i.e.,* ToR-to-ToR flows in a data center), but still generates many rules. A more scalable alternative installs coarse-grained rules that direct a consecutive chunk of flows to a common next-hop. A preliminary exploration of using wildcard rules is discussed in [5]. Niagara follows the same high-level approach, but presents more sophisticated algorithms for optimizing rule-table size, while also addressing churn under updates. We discuss [5] in detail in Section 4.1.

Other approaches for multi-pathing. The traffic-splitting problem has been studied extensively in the past in the context of multi-pathing. LocalFlow [29] achieves perfectly uniform splits, but cannot produce weighted splits and may split a flow, causing packet reordering. Conga [30] and Flare [31] load balance flowlets (bursts of packets within a flow) to avoid reordering but require advanced switch hardware support. In comparison, Niagara load balances traffic without packet reordering using off-the-shelf OpenFlow switches. An alternative approach to these schemes is centralized flow scheduling such as Hedera [11]. Hedera reroutes "elephant" flows based on global information. Niagara could provide the default routing scheme for a centralized flow-scheduler which then installs specific flow-rules for elephant flows. The third type of approaches is host-controlled routing, which changes the paths of packets by customizing extra fields in ECMP hash functions [32] or round-robin forwarding to intermediate switches [33]. Niagara does not directly compete with these approaches by design, as it does not touch the end-hosts.

3. NIAGARA OVERVIEW

Niagara generates wildcard rules to split the traffic within the constrained rule-table size. Incoming traffic is grouped into flow aggregates, each of which is divided over the same set of next-hops according to a weight vector. The per-aggregate weight vector is calculated with consideration on the bandwidth of both downstream links and capacity of next-hops. In the load balancing example, incoming packets are grouped by their destination IPs (*i.e.,* services). Traffic of each service is divided over next-hops (*i.e.,* SLBs) according to their capacity (*e.g.,* bandwidth, CPU, the number of back-end servers they connect to). Figure 1(a) shows an example

Match		**Action**
DIP	SIP	Next-hop
63.12.28.42	∗0	17.12.11.1
63.12.28.42	∗	17.12.12.1
63.12.28.34	∗00100	17.12.11.1
63.12.28.34	∗000	17.12.11.1
63.12.28.34	∗0	17.12.12.1
63.12.28.34	∗	17.12.13.1

(a) Load balancing two services.

Match	**Action**
DIP	Tag
63.12.28.42	1
63.12.28.53	1
63.12.28.27	1
63.12.28.34	2
63.12.28.43	2

$\Longrightarrow$

Match		**Action**
Tag	SIP	Next-hop
1	∗0	17.12.11.1
1	∗	17.12.12.1
2	∗00100	17.12.11.1
2	∗000	17.12.11.1
2	∗0	17.12.12.1
2	∗	17.12.13.1

(b) Grouping and load balancing five services.

Figure 1: Example wildcard rules for load balancing.

of wildcard rules generated by Niagara for load balancing. Each rule matches on destination IP to identify the service and source IP to forward packets to the same SLBs. Packets are forwarded based on the first matching rule. In addition to wildcard rules, Niagara leverages the metadata tags supported by latest chip-sets [14] and generates tagging rules to group services of similar weight distributions, thus further reducing the number of rules (Figure 1(b)).

In this section, we formulate the optimization problem for computing wildcard rules in the switch and outline the five main components of our algorithm. For easy exposition of the rule generation algorithm, we use *suffixes of source IP address* and assume a proportional split of the traffic over suffixes (*e.g.,* ∗0 stands for 50% traffic). We relax this assumption in Section 4.1.2.

3.1 Problem Formulation

The algorithm computes the rules in the switch, given the per-aggregate weights and the switch rule-table capacity. A hardware switch should approximate the target division of traffic over the next-hops accurately. The misdirected traffic may introduce congestion over downstream links and overload on next-hops. As such, an important challenge is to minimize the *imbalance*—the fraction of traffic that routes to the "wrong" next-hops.

The weights of each aggregate vary due to differences in resource allocation (*e.g.,* bandwidth), next-hop failures, and planned maintenance. Each aggregate v has non-negative weights $\{w_{vj}\}$ for splitting traffic over the M next-hops $j=1,2,\ldots,M$, where $\sum_j w_{vj}=1$. (Table 1 summarizes the notation.) The traffic split is not always exact, since matching on header bits inherently discretizes portions of traffic. In practice, splitting traffic *exactly* is not necessary, and aggregates can tolerate a given error bound e, where the actual split is w'_{vj} such that $|w'_{vj}-w_{vj}|\leq e$. The value of e depends on the deployment: an aggregate with a few next-hops requires a smaller e value (usually in $[0.001, 0.01]$). Ideally, the hardware switch could achieve w'_{vj} with wildcard rules. But small rule-table sizes thwart this, and instead, we settle

Variable	Definition
N	Number of aggregates ($v = 1, \ldots, N$)
M	Number of next-hops ($j = 1, \ldots, M$)
C	Hardware switch rule-table capacity
w_{vj}	Target weight for aggregate v, next-hop j
t_v	Traffic volume for aggregate v
d_v	Traffic distribution for aggregate v over the flow space
e	Error tolerance $\lvert w'_{vj} - w_{vj} \rvert \leq e$
w'_{vj}	Actual weight for aggregate v, next-hop j
c_v	Hardware rule-table space for aggregate v

Table 1: Table of notation, with inputs listed first.

for the lesser goal of approximating the weights as well as possible, given a limited rule capacity C at the switch.

To approximate the weights, we solve an optimization problem that allocates c_v rules to each aggregate v to achieve weights $\{w'_{vj}\}$ (*i.e.,* $c_v = numrules(\{w'_{vj}\})$). Aggregate v has traffic volume t_v, where some aggregates contribute more traffic than others. We define the total imbalance as the sum of over-approximated weights. The goal is to minimize the total traffic imbalance, while approximating the weights:

$$\begin{aligned} \text{minimize } & \textstyle\sum_v (t_v \times \sum_j E(w'_{vj} - w_{vj}, e)) && s.t. \\ & w'_{vj} \geq 0 && \forall v, j \\ & \textstyle\sum_j w'_{vj} = 1 && \forall v \\ & c_v = numrules(\{w'_{vj}\}) && \forall v \\ & \textstyle\sum_v c_v \leq C \\ \text{where } & E(x,e) = \begin{cases} x & \text{if } x > e \\ 0 & \text{if } x \leq e \end{cases} \end{aligned}$$

given the weights $\{w_{vj}\}$, traffic volumes $\{t_v\}$, rule-table capacity C, and error tolerance e as inputs.

3.2 Overview of Optimization Algorithm

Our solution to the optimization problem introduces five main contributions, starting with the following three ideas:

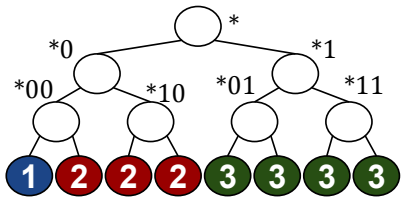
Approximating weights for a single aggregate (Section 4.1): Given weights $\{w_{vj}\}$ for aggregate v and error tolerance e, we compute the approximated weights $\{w'_{vj}\}$ and the associated rules for each aggregate. The algorithm expands each weight w_{vj} in terms of powers of two (*e.g.,* $\frac{1}{6} \approx \frac{1}{8} + \frac{1}{32}$) that can be approximated using wildcard rules.

Truncating the approximation to use fewer rules (Section 4.2): Given the above results, we can truncate the approximation and fit a *subset* of associated rules into the rule table. This results in a *tradeoff curve* of traffic imbalance versus the number of rules.

Packing multiple aggregates into a single table (Section 5.1): We allocate rules to aggregates based on their tradeoff curves to minimize the total traffic imbalance. In each step of the packing algorithm, we allocate one more rule to the aggregate that achieve the highest ratio of the *benefit* (the reduction in traffic imbalance) to the *cost* (number of rules), until the hardware table is full with a total of $C = \sum_v c_v$ rules. Consequently, more rules are allocated to aggregates with larger traffic volume and easy-to-approximate weights.

Together, these three parts allow us to make effective use of a small rule table to divide traffic over next-hops.

Thousands of aggregates with dozens of next-hops can easily overwhelm the small wildcard rule table (*i.e.,* TCAM)



(a) Suffix allocation

Pattern	Action
$*000$	fwd to 1
$*100$	fwd to 2
$*10$	fwd to 2
$*1$	fwd to 3

(b) Naive approach

Pattern	Action	Priority
$*000$	fwd to 1	high
$*0$	fwd to 2	low
$*1$	fwd to 3	low

(c) Use subtraction and priority

Figure 2: Naive and subtraction-based rule generation for weights $\{\frac{1}{6}, \frac{1}{3}, \frac{1}{2}\}$ and approximation $\{\frac{1}{8}, \frac{3}{8}, \frac{4}{8}\}$.

in today's hardware switches. Fortunately, today's hardware switches have multiple table stages. For example, the popular Broadcom chipset [14] has a table that can match on destination IP prefix and set a metadata tag that can be matched (along with the five-tuple) in the subsequent TCAM. Niagara can capitalize on this table to map an aggregate to a tag—or, more generally, *multiple* aggregates to the same tag. Our fourth algorithmic innovation uses this table:

Sharing rules across aggregates with similar weights (Section 5.2): We associate a tag with a group of aggregates with similar weights over the same next-hops. We use k-means clustering to identify the groups, and then generate one set of rules for each group. Furthermore, we create a set of default rules, which are shared by all groups.

Transitioning to new weights (Section 6): In practice, weights change over time, forcing Niagara to compute *incremental* changes to the rules to control the churn.

4. OPTIMIZE A SINGLE AGGREGATE

We begin with generating rules to approximate the weight vector $\{w_{vj}\}$ of a single aggregate v within error tolerance e. We then extend the method to account for constrained rule-table capacity C.

4.1 Approximate: Binary Expansion

Naive approach to generating wildcard rules. A possible method to approximate the weights [5] is to pick a fixed suffix length k and round every weight to the closest multiple of 2^{-k} such that the approximated weights still sum to 1. For example by fixing $k = 3$, weights $w_{v1} = \frac{1}{6}$, $w_{v2} = \frac{1}{3}$, and $w_{v3} = \frac{1}{2}$ are approximated by $w'_{v1} = \frac{1}{8}$, $w'_{v2} = \frac{3}{8}$, and $w'_{v3} = \frac{4}{8}$. The visualized suffix tree is presented in Figure 2(a). To generate the corresponding wildcard rules, an approximate weight $b \times 2^{-k}$ is represented by b k-bit rules. In practice, allocating similar suffix patterns to the same weight may enable combining some of the rules, hence reducing the number of rules. The corresponding wildcard rules are listed in Figure 2(b).

Shortcomings of the naive solution. The naive approach always expresses b as the "sums" of power of two (for ex-

Iteration	w'_{v1}	w'_{v2}	w'_{v3}
0	0	0	1
1	0	$\frac{1}{2}$	$1-\frac{1}{2}$
2	$\frac{1}{8}$	$\frac{1}{2}-\frac{1}{8}$	$1-\frac{1}{2}$
3	$\frac{1}{8}+\frac{1}{32}$	$\frac{1}{2}-\frac{1}{8}-\frac{1}{32}$	$1-\frac{1}{2}$

(a) Approximation iterations

Pattern	Action	Corresponding terms
$*00100$	fwd to 1	$\frac{1}{32}$ in w'_{v1} and $-\frac{1}{32}$ in w'_{v2}
$*000$	fwd to 1	$\frac{1}{8}$ in w'_{v1} and $-\frac{1}{8}$ in w'_{v2}
$*0$	fwd to 2	$\frac{1}{2}$ in w'_{v2} and $-\frac{1}{2}$ in w'_{v3}
$*$	fwd to 3	1 in w'_{v3}

(b) Wildcard rules

Figure 3: Wildcard rules to approximate $(\frac{1}{6},\frac{1}{3},\frac{1}{2})$

ample $\frac{3}{8}$ is expressed as $\frac{2}{8}+\frac{1}{8}$) and only generates non-overlapping rules. In contrast, our algorithm allows *subtraction* as well as longest-match *rule priority*. In the above example, $\frac{3}{8}$ can be expressed as $\frac{4}{8}-\frac{1}{8}$ to achieve the same approximation with one less rule (Figure 2(c)). The generated rules overlap and the longest-matching rule is given higher priority: $*000$ is matched first and "steals" $\frac{1}{8}$ of the traffic from rule $*0$.

The power of subtractive terms and rule priority. Our algorithm approximates weights using a series of *positive and negative* power-of-two terms. We compute the approximation $w'_{vj}=\sum_k x_{jk}$ for each weight w_{vj} subject to $|w'_{vj}-w_{vj}|\le e$. Each term $x_{jk}=b_{jk}\times 2^{-a_{jk}}$, where $b_{jk}\in\{-1,+1\}$ and a_{jk} is a non-negative integer. For example, $w_{v2}=\frac{1}{3}$ is approximated using three terms as $w'_{v2}=\frac{1}{2}-\frac{1}{8}-\frac{1}{32}$. As we explain later, each term x_{jk} is mapped to a suffix matching pattern. In what follows, we show how to compute the approximations and how to generate the rules.

4.1.1 Approximate the weights

We start with an initial approximation where the biggest weight is 1 and the other weights are 0. The initial approximation for $w_v=(\frac{1}{6},\frac{1}{3},\frac{1}{2})$ is $w'_v=(0,0,1)$ (Figure 3(a)). The errors, namely the difference between the w'_v and w_v, are $(-\frac{1}{6},-\frac{1}{3},\frac{1}{2})$. w_{v1},w_{v2} are under-approximated , while w_{v3} is over-approximated.

We use error tolerance $e=0.02$ for the example. The initial approximation is not good enough; w_{v2} is the most under-approximated weight with an error $-\frac{1}{3}$. To reduce its error, we add one power-of-two term to w'_{v2}. At the same time, this term must be subtracted from another over-approximated weight to keep the sum unchanged. We move a power-of-two term from w_{v3} to w_{v2}.

We decide the term based on the current errors of both weights. The term should offer the biggest reduction in errors. Let the power-of-two term be x. Given the current errors of w_{v2} and w_{v3}, *i.e.,* $-\frac{1}{3}$ and $\frac{1}{2}$, we calculate the new errors as $-\frac{1}{3}+x$ and $\frac{1}{2}-x$. Hence, the reduction is $\triangle=|-\frac{1}{3}|+|\frac{1}{2}|-|-\frac{1}{3}+x|-|\frac{1}{2}-x|=2\times(\min(\frac{1}{3},x)+\min(\frac{1}{2},x)-x)$.

The function is plotted as red line in Figure 4. When

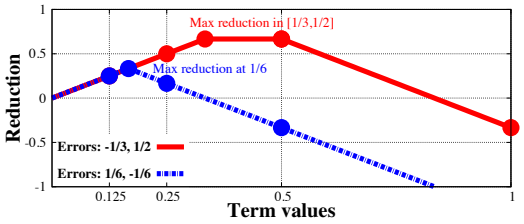


Figure 4: $\triangle$ plots with different errors.

$x=1,\frac{1}{2}$ and $\frac{1}{4}$, the reduction is $-\frac{1}{3},\frac{2}{3}$ and $\frac{1}{2}$ respectively. In fact, Equation $\triangle$ is a concave function, which reaches its maximum value when $x\in[\frac{1}{3},\frac{1}{2}]$. Hence, we choose $\frac{1}{2}$. In a more general case, where multiple values give the maximum reduction, we break the tie by choosing the biggest term. After this operation, the new approximation becomes $(0,\frac{1}{2},1-\frac{1}{2})$ with errors $(-\frac{1}{6},\frac{1}{6},0)$.

We repeat the same operations to reduce the biggest under-approximation and over-approximation errors iteratively. In the example, w_{v3} is perfectly approximated (the error is 0). We only move terms from w_{v2} to w_{v1}. Two terms $\frac{1}{8},\frac{1}{32}$ are moved until all the errors are within tolerance. Eventually, each weight is approximated with an expansion of power-of-two terms (Figure 3(a)).

We make three observations about this process. First, the errors are non-increasing, as each time we reduce the biggest errors. Second, the chosen power-of-two terms are non-increasing, because the terms with the maximum $\triangle$ always lie between two errors (Figure 4). For a term that gives the best $\triangle$ in the current iteration, only smaller terms may have a bigger reduction in the next iteration[1]. Finally, the reduction $\triangle$ is non-increasing, as Equation $\triangle$ is monotonic with both errors and the chosen power-of-two term. *In other words, we gain diminishing return on $\triangle$ for the term-moving operation, as we are getting closer to the error tolerance.*

4.1.2 Generate rules based on approximations

Given the approximation w'_v, we generate rules by mapping the power-of-two terms to nodes in a suffix tree. Each node in the tree represents a 2^{-k} fraction of traffic, where k is the node's depth (or, equivalently, the suffix length). Figure 5 visualizes the rule-generation steps for our example from Figure 3(a) with $w_{v1}=\frac{1}{6}$, $w_{v2}=\frac{1}{3}$, and $w_{v3}=\frac{1}{2}$. When a term is mapped to a node, we explicitly assign a color to the node. Initially, the root node is colored with the biggest weight to represent the initial approximation (Figure 5(a)). Color j means that the node belongs to w'_{vj}. Each uncolored node implicitly *inherits* the color of its closest ancestor. We use dark color for explicitly colored nodes and light color for the unassigned nodes.

We process the terms in the order that they are added to the expansions (*i.e.,* $\frac{1}{2}$, $\frac{1}{8}$, $\frac{1}{32}$). Then, one by one, the terms are mapped to nodes as follows. Let x be the term under consideration, which is moved from weight w_{vb} to w_{va}. We map it to a node representing x fraction of traffic with color b. The node is then re-colored to a. In the example, we map $\frac{1}{2}$ to node $*0$ and color the node with w_{v2} (Figure 5(b)). Sub-

[1]A term may be picked in multiple consecutive iterations.

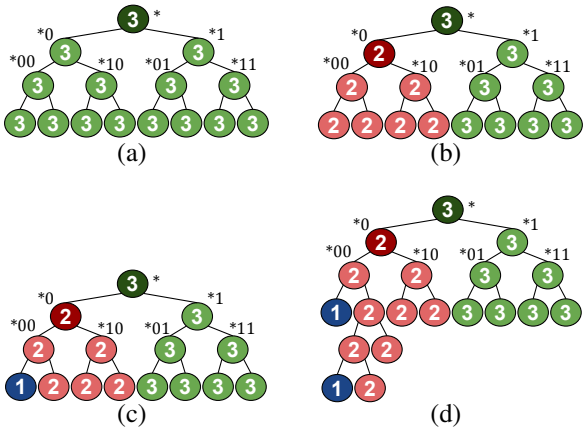


Figure 5: Generate rules using a suffix tree.

sequently, $\frac{1}{8}, \frac{1}{32}$ are mapped to $*000, *00100$, which are colored to w_{v1} (Figure 5(c) (d)).

Once all terms have been processed, rules are generated based on the explicitly colored nodes. Figure 3(b) shows the rules corresponding to the final colored tree in Figure 5(d).

4.1.3 *Use non-power-of-two terms*

We discuss the case that each suffix pattern may not match a power-of-two fraction of traffic. For example, there may be more packets matching $*0$ than those matching $*1$. Niagara's algorithm can be extended to handle the unevenness, once the fractions of traffic for suffixes are measured [34–37].

We refine the approximation iteratively. In each iteration, a suffix (*i.e.,* a term) is moved from an over-approximated weight to an under-approximated weight to maximize the reduction of errors. The only difference is that the candidate values of this term are no longer powers of two, but all possible fractions denoted by suffixes belonging to the over-approximated weight. We use the concaveness of Equation $\triangle$ to guide our search for the best term value. Instead of brute-force enumeration, we can scan all candidate values in decreasing order, and stop when $\triangle$ starts decreasing.

To illustrate the extended algorithm, we use $w_v = (\frac{1}{6}, \frac{1}{3}, \frac{1}{2})$ as an example and assume an uneven traffic distribution over the flow space shown in Figure 6. We start with the approximation $w'_v = (0,0,1)$ (Figure 7(a)) and move a suffix from the over-approximated weight w_{v3} to the most under-approximated weight w_{v2} in the first iteration. Based on Equation $\triangle$, among all suffixes of w'_{v3}, *1 with term $= \frac{2}{5}$ maximizes $\triangle$ and is moved to w_{v2} (Figure 7(b)). The approximation becomes $(0, \frac{2}{5}, 1-\frac{2}{5})$. In the next iteration, we move suffix *100 with term $= \frac{18}{125}$ to w_{v1}, reducing the approximation error to $w'_v - w_v = (\frac{18}{125} - \frac{1}{6}, \frac{2}{5} - \frac{1}{3}, (1 - \frac{2}{5} - \frac{18}{125}) - \frac{1}{2})$. Finally, moving *111 with term $= \frac{8}{125}$ to w_{v2} completes the approximation. The resulting suffix tree is shown in Figure 7(d).

We also remark that it is not necessary to use suffix matches to approximate traffic volume. As long as the traffic distribution is measured for some bits in the header fields, we could apply the above algorithm to generate patterns matching those bits.

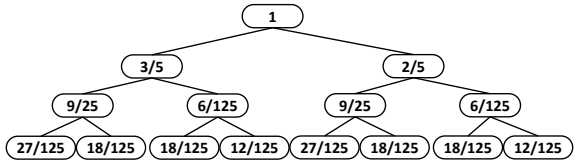


Figure 6: An example traffic distribution with a suffix tree. Each number represents the fraction of traffic matched by the suffix, *e.g.,* *11 matches $\frac{4}{25}$ traffic.

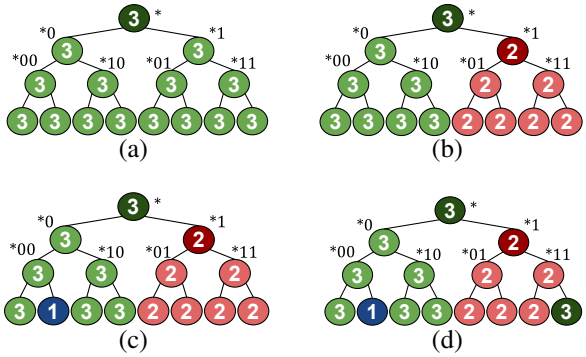


Figure 7: Generate rules using a suffix tree, given the traffic distribution in Figure 6.

4.2 Truncate: Fit Rules in the Table

Given the restricted rule-table size, some generated rules might not fit in the hardware. Therefore, we *truncate* rules to meet the capacity of rule table. We refer to the switch rules as P^H. P^H achieves a coarse-grained approximation of the weights while $numrules(P^H)$ stays within the rule-table size C. We capture the total over-approximation error as *imbalance*, *i.e.,* $t_v \times \sum_j \max(w^H_{vj} - w_{vj}, e)$, where t_v is the expected traffic volume for aggregate v and w^H_{vj} is the approximation of weight w_{vj} given by P^H.

We pick the C lower-priority rules from the rule-set generated in Section 4.1 as P^H. This is because rules are generated with increasing priority and decreasing $\triangle$ values (*i.e.,* the reduction in imbalance). The C lowest-priority rules give the overall biggest reduction of imbalance. For example, when $C = 3$ the rules in Figure 3(b) are truncated into P^H containing the last three rules.

Stairstep plot. Figure 8 shows the imbalance as a function of C. Each point in the plot (r, imb) can be viewed as a *cost* for rule space r, and the corresponding *gain* in reducing imbalance *imb*. This curve helps us determine the gain an aggregate can have from a certain number of allocated switch rules, which is used in packing rules for multiple aggregates into the same switch table (Section 5.1).

5. CROSS AGGREGATES OPTIMIZATION

In this section, we generate rules for multiple aggregates using two main techniques: (1) *packing* multiple sets of rules (each corresponding to a single aggregate) into one rule table and (2) *sharing* the same set of rules among aggregates.

5.1 Pack: Divide Rules Across Aggregates

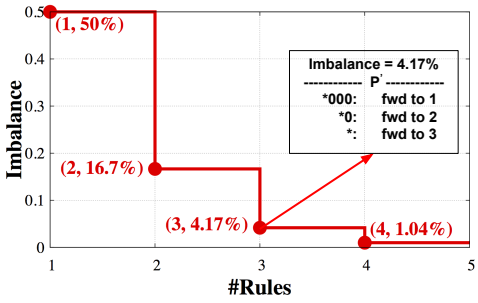


Figure 8: Stairstep curve (imbalance v.s. #rules) for Aggregate v with weights $w_v = \{\frac{1}{6}, \frac{1}{3}, \frac{1}{2}\}$ and $t_v = 1$.

The stairstep plot in Figure 8 presents the tradeoff between the number of rules allocated to an aggregate and the resulting imbalance. When dividing rule-table space across multiple aggregates, we use their stairstep plots to determine which aggregates should have more rules, to minimize the total traffic imbalance. Figure 9 shows the weight vectors, traffic volumes and stairsteps of two aggregates.

To allocate rules, we greedily sweep through the stairsteps of aggregates in steps. In each sweeping step, we give one more rule to the aggregate with *largest per-step gain* by stepping down one unit along its stairstep. The allocation repeats until the table is full.

We illustrate the steps through an example of packing two aggregates v_1 and v_2 using five rules (Figure 9). We begin with allocating each aggregate one rule, resulting in a total imbalance of 50% (27.5% + 22.5%). Then, we decide how to allocate the remaining three rules. Note that v_1's per-step gain is 18.33% (27.5% − 9.17%), which means that giving one more rule to v_1 would reduce its imbalance from 27.5% to 9.17%, while v_2's gain is 11.25% (22.5% − 11.25%). We therefore give the third rule to v_1 and move one step down along its curve. The per-step gain of v_1 becomes 6.88% (9.17% − 2.29%). Using the same approach, we give both the fourth and fifth rules to v_2, because its per-step gains (22.5% − 11.25% = 11.25% and 11.25% − 0% = 11.25%) are greater than v_1's. Therefore, v_1 and v_2 are given two and three rules, respectively, and the total imbalance is 9.17% (9.17% + 0%). The resulting rule-set is a combination of rules denoted by point (2, 9.17%) in v_1's stairstep and (3, 0%) in v_2's.

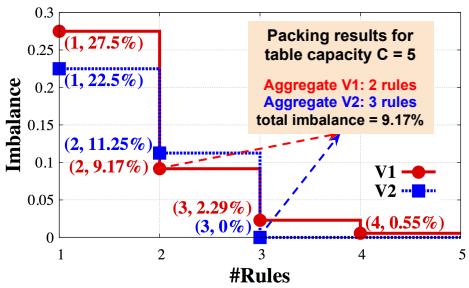
A natural consequence of our packing method is that aggregates with heavy traffic volume and easy-to-approximate weights are allocated more rules. Our evaluation demonstrates that this way of handling "heavy hitters" leads to significant gains.

5.2 Share: Same Rules for Aggregates

In practice, a switch may split thousands of aggregates. Given the small TCAM in today's hardware switches, we may not always be able to allocate even one rule to each aggregate. Thus, we are interested in *sharing* rules among multiple aggregates, which have the same set of next-hops. We employ sharing on different levels, creating three types of rules (with decreasing priority): (1) rules specific to a single aggregate (Section 4); (2) rules shared among a group of aggregates (Section 5.2.2), and (3) rules shared among *all* aggregates, called *default rules* (Section 5.2.1).[2]

Aggregate	Weights	Traffic Volume
v_1	$w_{11} = \frac{1}{6}, w_{12} = \frac{1}{3}, w_{13} = \frac{1}{2}$	$t_1 = 0.55$
v_2	$w_{21} = \frac{1}{4}, w_{22} = \frac{1}{4}, w_{23} = \frac{1}{2}$	$t_2 = 0.45$

(a) Weights and traffic volume of v_1 and v_2.



(b) Packing v_1 and v_2 based on stairsteps.

Figure 9: An example of packing multiple aggregates.

5.2.1 *Default rules shared by all aggregates*

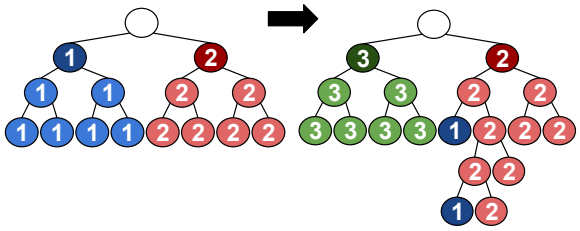
Default rules have the lowest priority and are shared by all aggregates. There are many ways to create default rules, including approximating a certain weight vector using algorithm in Section 4. Here we focus on the simplest and most natural one—*uniform default rules* that divide the traffic equally among next-hops.

Assuming there are M next-hops where $2^k \le M < 2^{k+1}$, we construct 2^k default rules matching suffix patterns of length k and distributing traffic evenly among the first 2^k next-hops. [3] These rules provide an initial approximation w^E of the target weight vector: $w^E_i = 2^{-k}$ for $i \le 2^k$ and $w^E_i = 0$ otherwise, which can then be improved using more-specific per-aggregate rules. If aggregates do not use the same set of next-hops, the default rules will only balance over the common set of next-hops and the per-aggregate rules will rebalance the loads of the rest of next-hops.

We revisit the example $(\frac{1}{6}, \frac{1}{3}, \frac{1}{2})$. The initial approximation $w^E = (\frac{1}{2}, \frac{1}{2}, 0)$. $w_{v1} = \frac{1}{6}$ is over-approximated with error $\frac{1}{3}$; $w_{v3} = \frac{1}{2}$ is under-approximated with error $-\frac{1}{2}$; we move $\frac{1}{2}$ from w_{v1} to w_{v3}. The rest operations are similar to Section 4.1. Figure 10(a) shows the corresponding suffix tree. Initially, the tree is colored according to the uniform default rules. Next, we refine the approximation and obtain terms $\frac{1}{2}$, $\frac{1}{8}$, $\frac{1}{32}$ and the final rules (Figure 10(b)). The total number of rules is five, compared to four rules without using default rules (Figure 3(b)). However, only three of the five rules are "private" to aggregate v, as the two default rules are shared among all aggregates. This illustrates that default rules may not save space for one (or even several) aggregates, but will usually bring significant table space savings when the number of aggregates is large (Section 7).

[2]Default rules do not require extra grouping table.

[3]When M is the power of two, the uniform default rules gives an equivalent split to ECMP.



(a) Initial (left) and final (right) suffix trees for $w'_{v1} = \frac{1}{2} - \frac{1}{2} + \frac{1}{8} + \frac{1}{32}$, $w'_{v2} = \frac{1}{2} - \frac{1}{8} - \frac{1}{32}$, $w'_{v3} = \frac{1}{2}$ (pool).

Rules	Pattern	Action
Rules for aggregate v	$*00101$	fwd to 1
	$*001$	fwd to 1
	$*0$	fwd to 3
Shared default rules	$*0$	fwd to 1
	$*1$	fwd to 2

(b) Rules that approximate v.

Figure 10: Generate rules for $\{\frac{1}{6}, \frac{1}{3}, \frac{1}{2}\}$ given default rules

5.2.2 *Grouping aggregates with similar weights*

To further save the table space, we group aggregates and tag aggregates in each group with the same identifier.

We use k-means clustering to group aggregates with similar weights. The centroid of each group is computed as the average weight vector of its member aggregates; to prioritize "heavy" aggregates, the average is weighted using t_v (the expected traffic volume of aggregate v). We begin by selecting the top-k aggregates with highest traffic volume as the initial centroid of the groups, where the choice of k depends on the available rule table space (Section 7). Then, we assign every aggregate to the group whose centroid vector is closest to the aggregate's target weight vector (using Euclidean distance). After assignment, we re-calculate group centroids. The procedure is repeated until the overall distance improvement is below a chosen threshold (*e.g.,* 0.01% in our evaluation).

Putting it all together. Niagara's full algorithm first (i) groups similar aggregates, then (ii) creates one set of default rules (*e.g.,* uniform rules) that serve as the initial approximation for all the groups, (iii) generates per-group stairstep curves, and finally (iv) packs groups into a rule table.

6. GRACEFUL RULE UPDATE

Weights change over time, due to next-hop failures, rolling out of new services, and maintenance. When the weights for an aggregate change, Niagara computes new rules while minimizing (i) churn due to the difference between old and new weights and (ii) traffic imbalance due to inaccuracies of approximation. Niagara has two update strategies, depending on the frequency of weight changes. When weights change frequently, Niagara *minimizes churn* by incrementally computing new rules from the old rules (Section 6.1). When weights change infrequently, Niagara minimizes *traffic imbalance* by computing the new set of rules from scratch and installs them in stages to limit churn (Section 6.2).

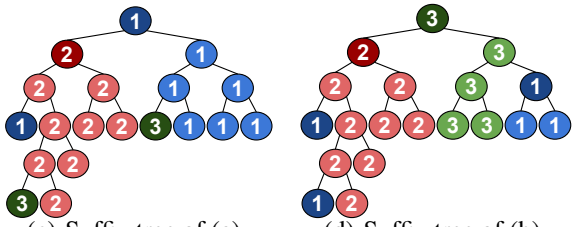
6.1 Incremental Rule Computation

Pattern	Action
$*00100$	fwd to 3
$*100$	fwd to 3
$*000$	fwd to 1
$*0$	fwd to 2
$*$	fwd to 1

(a) Target rules.

Pattern	Action
$*00100$	fwd to 1
$*000$	fwd to 1
$*11$	fwd to 1
$*0$	fwd to 2
$*$	fwd to 3

(b) Intermediate rules.



(c) Suffix tree of (a). (d) Suffix tree of (b).

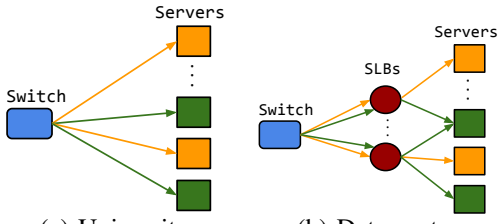
Figure 11: Rule-sets (and corresponding suffix trees) installed during the transition from $\{\frac{1}{6}, \frac{1}{3}, \frac{1}{2}\}$ to $\{\frac{1}{2}, \frac{1}{3}, \frac{1}{6}\}$.

When weights change, Niagara computes new rules to approximate the updated weights. New rules not only determine the new imbalance, but also the traffic churn during the transition. We use an example of changing weights from $\{\frac{1}{6}, \frac{1}{3}, \frac{1}{2}\}$ to $\{\frac{1}{2}, \frac{1}{3}, \frac{1}{6}\}$ to illustrate the computation of new rules. Initial rules are given in Table 3(b) and the corresponding suffix tree in Figure 5(d). In this example, any solution must shuffle at least $\frac{1}{3}$ of the flow space (assuming a negligible error tolerance e), namely the minimal churn is $\frac{1}{3}$.

Minimize imbalance (recompute rules from scratch). A strawman approach to handle weight updates is to compute new rules from scratch. In our example, this means that action "fwd to 1" in Table 3(b) become "fwd to 3" and vice versa. This approach minimizes the traffic imbalance by making the best use of rule-table space. However, it incurs two drawbacks. First, it leads to heavy churn, since recoloring $\frac{1}{2} + \frac{1}{8} + \frac{1}{32}$ fraction of the suffix tree in Figure 5(d) means that nearly $\frac{2}{3}$ of traffic will be shuffled among next-hops. Second, it requires significant updates to hardware, which slow down the update process. As a result, this approach does not work well when weights change frequently.

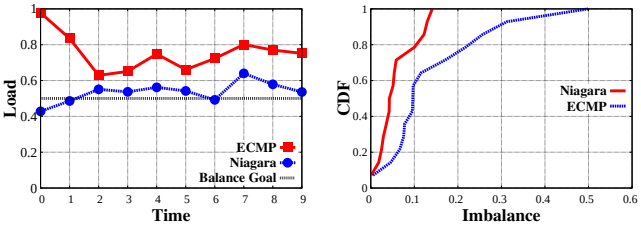
Minimize churn (keep rules unchanged). An alternative strawman is to keep the switch rules "as is". This approach minimizes churn but results in significant imbalance and overloads on next-hops. In the example, both the churn and the new imbalance are roughly $\frac{1}{3}$.

Strike a balance (incremental rule update). The above two approaches illustrate two extremes in computing the new rules. Niagara intelligently explores the tradeoff between churn and imbalance by iterating over the solution space, varying the number of old rules kept. In the example, keeping two old rules ($*000$ fwd to 1, and $*0$ fwd to 2) leads to the rule-set shown in Figure 11(a) and the suffix tree in Figure 11(c). The imbalance is $\frac{1}{32}$, the same with computation from scratch; the churn is $\frac{1}{32} + \frac{3}{8}$, which is slightly higher than the minimum churn $\frac{1}{3}$, as suffixes $*00100, *011, *11$ are re-colored to 1. In practice, when computing new rules for an aggregate, Niagara does not use more rules than the old ones.



(a) University (b) Data center

Figure 12: Load balancer architecture.



(a) Server load (single VIP) (b) Imbalance

Figure 13: Accuracy of uniform server load balancing.

6.2 Multi-stage Updates

Incurring churn during updates is inevitable. Depending on the deployment, this traffic churn might not be tolerable. Niagara is able to bound the churn by dividing the update process into multiple stages. Given a threshold on acceptable churn, Niagara finds a sequence of intermediate rulesets such that the churn generated by transitioning from one stage to the next is always under the threshold.

Continuing the example in Section 6.1, we limit maximum acceptable churn to $\frac{1}{4}$. The churn for the direct transition from the old rules to the new rules is $\frac{1}{32}+\frac{3}{8}$, exceeding the threshold. Hence, we need to find an intermediate stage so that both the transition from the old rules to the intermediate rules and from the intermediate rules to the new rules do not exceed the threshold.

To compute the intermediate rules, we pick the pattern $*11$, which is the *maximal* fraction of the suffix tree that can be recolored within the churn threshold. The intermediate tree (Figure 11(d)) is obtained by replacing the subtree $*11$ of the old one (Figure 5(d)) with the new one's (Figure 11(c)). The intermediate rules are computed accordingly. Then, transitioning from the intermediate suffix-tree in Figure 11(d) to the one in Figure 11(c) recolors only $\frac{1}{32}+\frac{1}{8}$ ($<\frac{1}{4}$) of the flow space and therefore we can transition directly to the rules in Figure 11(a) after the intermediate stage.

We note that performing a multi-stage update naturally results in lengthy update process for aggregates with frequent weight changes. To mitigate this, Niagara may rate limit the update frequency of aggregates.

7. EVALUATION

This section presents the evaluation of Niagara in two scenarios: server load balancing and multi-path traffic splitting. We conduct both trace-driven analysis and synthetic experiments to demonstrate Niagara's splitting accuracy, scalability and update efficiency.

7.1 Niagara for Server Load Balancing

We evaluate Niagara's accuracy against real packet traces and load balancing configuration from a campus network. We further use large-scale synthetic data-center load balancing configuration to examine its scalability and update efficiency. Before diving into the results, we first describe the experiment setup and data for the two scenarios.

Setup. We use two different load balancer architectures (Figure 12). In the campus network, the switch directly forwards VIP requests to backend servers. VIPs are deployed on different servers, hence the switch cannot use default rules that are intended to be shared by all aggregates (*i.e.,* VIPs). In the data center network, the switch directs requests to an intermediate layer of Software Load Balancers (SLBs), which encapsulate packets to a pool of backend servers. All VIP requests are distributed over the same set of SLBs, although the weights for each VIP can be different depending on the deployment of backend servers behind SLBs.

University traces and configuration. The campus network hosts around 50 services (*i.e.,* VIPs). Each VIP is served by 2 to 5 backends. VIP requests should be evenly distributed over backends. We collected a 20-minute Netflow traces from the campus border router and extracted the top 14 popular VIPs from the traces for our evaluation as the other VIPs saw only negligible traffic.

Synthetic weight distribution. In a large-scale data center network, the weights of a VIP depend on various factors such as capacity of next-hop servers and deployment plans. To reflect this variability, we use three different distribution models to choose VIP weights: Gaussian, Bimodal Gaussian, and Pick Next-hop. Weights of a VIP v are drawn from these models and normalized such that $\sum_j w_{vj}=1$.

Gaussian distribution. Weights are chosen from $N(4,1)$. Since the variance is small, the generated weights are close to uniform. This distribution models a setting where requests should be equally split over next-hops.

Bimodal Gaussian distribution. Here, each weight is chosen either from $N(4,1)$ or $N(16,1)$, with equal probability. The generated weights are non-uniform, but VIPs exhibit certain similarity. This distribution models a setting where some next-hops can handle more VIP requests than others.

Pick Next-hop distribution. In this model, we pick a subset of next-hops uniformly at random for each VIP. For the chosen next-hops, we draw the weights from the Bimodal Gaussian distribution and set the weights for the remaining unchosen next-hops to zero. The generated weights are non-uniform, making it hard for grouping. This case models a setting where different VIPs should be split over different subsets of next-hops.

Synthetic VIP traffic volume distribution. We use a Zipf traffic distribution where the k-th most popular VIP contributes $1/k$ fraction of the total traffic. The traffic volume is normalized so that $\sum_v t_v = 1$.

Metrics. We calculate *imbalance_lb* as $\sum_v(t_v \times \sum_j E(w'_{vj}-w_{vj},0))$, where t_v is the traffic volume of VIP v, w_{vj} is the desired fraction of loads on next-hop j by

VIP v and w'_{vj} is the actual load. A total imbalance $\leq 10\%$ is considered low.

7.1.1 Accuracy

We assume that the hardware switch directly forwards VIP requests to the backend servers (Figure 12(a)). The collected traffic traces exhibit stable traffic distribution over last 8 bits of source IP. In the experiment, *we run Niagara once with the profiled traffic distribution.*

We slice the 20-min trace into 2-min timeframes and compute the load of each backend using Niagara and ECMP. The ECMP hash function is SHA. We first examine one VIP with two backends each with 50% target load. Figure 13(a) shows the load of one of the backends. ECMP gives extremely unbalanced backend loads as part of the flow space contributes more traffic than the rest. On average, 80% of the load is absorbed by this backend and the total imbalance is $80\% - 50\% = 30\%$. In contrast, Niagara achieves a roughly balanced load with 1% imbalance. Figure 13(b) presents the CDF of imbalance for all VIPs. Even for uniform load balancing, ECMP still has a much longer imbalance tail than Niagara, because it merely splits the flow space equally regardless of the actual traffic distribution.

7.1.2 Rule Efficiency and Scalability

Next, we focus our attention to server load balancing in large-scale data center network setting (such as Duet [6] and Ananta [4]) with tens of thousands of VIPs, where hardware switches forward VIP requests to SLBs, which further distribute requests over backend servers (Figure 12(b)).

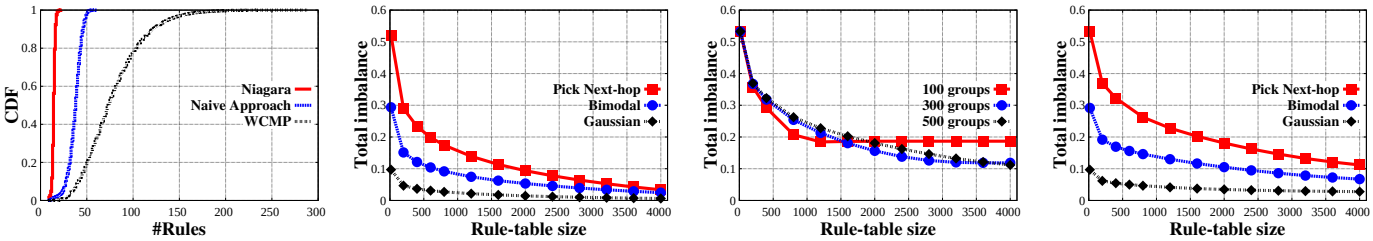
Approximate weights for a single VIP. We examine the number of rules needed to approximate the target weights of a single VIP assuming a balanced distribution of traffic over flow space. We randomly generate 100000 distinct sets of 8 weights (*i.e.,* 8 SLBs) with error tolerance $e = 0.001$. Figure 14(a) compares the CDF of the performance of three strategies (Section 4.1.1): *WCMP*, which repeats next-hop entries in ECMP, *Naive approach*, which rounds weights to the nearest multiples of powers of two and Niagara, which uses expansions of power-of-two terms to approximate weights. WCMP performs the worst and needs as many as; 288 rules to reach the error tolerance. Its performance is very sensitive to the values of the target weights. A slight change of weights (*e.g.,* from 0.1 to 0.11) may cause a dramatic change in number of rules. In fact, we see similar results for less tight error tolerance as well. The naive approach performs slightly better with a median of 38 rules, but still uses more rules (61 in the worst case) compared to Niagara. In comparison, Niagara generates the fewest rules (median is 14) with small variation. Niagara's performance is largely due to using both power-of-two terms and exploiting rule priorities to have both additive and subtractive terms.

Load balance multiple VIPs. Moving on to multiple VIPs, we use 16 weights per VIP (*i.e.,* 16 SLBs) and draw weights from the three synthetic models. We assume all VIPs share a set of uniform default rules. Figure 14(b) shows the total imbalance achieved by packing and sharing default rules for 500 VIPs, as a function of rule-table size. The leftmost point on each curve shows the imbalance given by the default rules (*i.e.,* ECMP). The initial imbalance for Gaussian, Bimodal and Pick Next-hop are $10\%, 30\%$ and 53% respectively. With Niagara, as the rule-table size increases, the imbalance drops nearly exponentially, reaching 3.3% at 4000 rules for Pick Next-hop model. This performance is due to the packing algorithm prioritizing "heavy-flows" when bumping up against rule-table capacity. Allocating rules to heavier-traffic sections of flow-space naturally minimizes imbalance given a fixed number of rules.

Our grouping technique (Section 5.2.2) groups VIPs with similar weight vectors. The maximal number of VIP groups affects approximation accuracy. When the VIPs are classified into more groups, the distance between each VIP's target weight vector and the centroid vector of its group is reduced, thus creating more groups containing only VIPs of more similar weights. However, as soon as rule capacity is reached, finer-grained VIP groups actually reduce overall performance because each group can push a small number of rules into the switch. Depending on number of groups, there is a tradeoff between grouping accuracy and approximation accuracy. When the VIPs are classified into more groups, the distance between each VIP's target weight vector and the centroid vector of its group is reduced, making the grouping more accurate. However, the approximation is less accurate for a bigger number of groups given limited rule capacity. Figure 14(c) illustrates this tradeoff by comparing the imbalance of classifying 10000 VIPs into 100, 300, and 500 groups. When there are less than 500 rules, classifying the VIPs into 100 groups performs best, because it is easier to pack 100 groups and the centroids of groups still give a reasonable approximation for aggregates. As rule-table sizes increase, using more fine-grained VIP groupings is advantageous, since the distance between each aggregate and its group's centroid, which "represents" the aggregate during packing, decreases. For example, given 1500 rules, 300-group outperforms 100-group.

Figure 14(d) shows the effectiveness of grouping for different weight models. Given the number of rules, we classify the VIPs into 100, 300, or 500 groups (picking the option which yields the smallest imbalance). At 4000 rules, we reach 2.8% and 6.7% imbalance for the Gaussian and Bimodal Gaussian models respectively, and 11.1% imbalance for Pick Next-hop, which is much tougher to group. In contrast, ECMP incurs imbalance of $9.6\%, 29.1\%$ and 53.2% (the leftmost point), respectively.

Time. The algorithm performs well on a standard Ubuntu server (Intel Xeon E5620, 2.4 GHz, 4 core, 12MB cache). The prototype single-threaded C++ implementation completes the computation of the stairstep curves for a 16-weight vector ($e = 0.001$) in $10ms$. The time of packing grows linearly with the number of aggregates and is dominated by the computation of stairstep curve, which could be parallelized. The grouping function using k-means clustering takes at most 8 sec. to complete. If the traffic distribution is skewed and VIPs use similar weight distributions the algorithm tends to converge faster and requires fewer iterations. We do not expect to update aggregate groups frequently: if two aggre-



(a) CDF of #Rules per VIP (b) 500 VIPs with default rules (c) Group sizes (d) Grouping 10000 VIPs

Figure 14: Weighted server load balancing for multiple VIPs.

gates are grouped together, they must have similar deployment in the network and are unlikely to be changed dramatically in a short term.

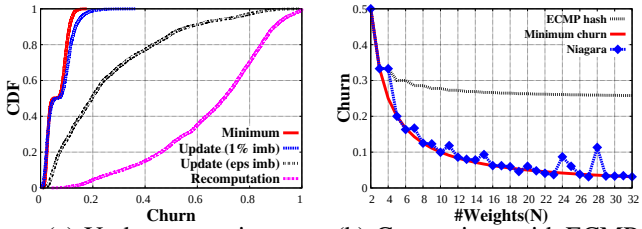
7.1.3 *Incremental Update*

We evaluate the churn and imbalance caused by Niagara's incremental update strategy. Given the old weight vector, we randomly clear one non-zero weight and renormalize the rest to obtain new weights, or vice versa, simulating a server failure or addition. The minimum churn is the weight of the failed (or added) server.

Incremental update with low churn and imbalance. Our performance baseline is an approach where the load balancer recomputes all forwarding rules from scratch in response to a weight change. This baseline approach completely ignores churn and prior assignments by recalculating all rules. This strategy does minimize the number of rules, however at the expense of incurring unnecessary traffic churn. In contrast, the incremental update algorithm in Niagara is aware of the cost of switching flows from one next-hop to another and tries to minimize churn. It keeps partial rules from the old rule-set and computes a small number of new rules to achieve the new weights while staying within bounded rule-space capacity. Figure 15(a) plots the CDF of the churn among 5000 weight vectors drawn from Bimodal distribution. The full recomputation approach (pink curve) incurs about 70% churn in 50% of test cases while Niagara's incremental update approach (black curve) only incurs 20% churn for half of test cases. This suggests that Niagara's intuition that an old rule-set serves as a good approximation for updated weights holds up in practice. Furthermore, this observation holds across the weight models used in this study.

Although Niagara's strategy explained above already reduces churn, it can be further improved by allowing a small margin for imbalance. The above strategy ignores larger rules-sets (than the minimum) that gives less churn. Based on this observation, we evaluate an alternative update strategy which installs truncated rules of larger rule-sets with up to 1% imbalance. The resulting curve (blue line in Figure 15(a)) almost overlaps with the curve of minimum churn (red). This confirms that an allowance for small imbalance will greatly reduce churn during updates.

Comparison with hash-based approaches. The theoretical lower bound of churn for ECMP, *i.e.,* assuming a perfect balanced traffic distribution over the flow space, is $\frac{1}{4}+\frac{1}{4N}$ for removing one member from a N-sized group (or adding one



(a) Update strategies (b) Comparison with ECMP

Figure 15: Incremental Update.

member to $(N-1)$-sized group) contrasting to the minimum churn of $\frac{1}{N}$ [9, 10]. We compare the churn of ECMP and Niagara using a uniform weight distribution, *e.g., N* weights of $\frac{1}{N}$. We create random server failure and additions as described in the previous experiment. For each value of N, Niagara generates the rule-set with minimum churn, while (1) staying within the number of rules needed by recomputation and (2) incurring less than 1% imbalance. Figure 15(b) presents the comparison of Niagara and ECMP. Niagara's performance (the blue line with diamonds) closely follows the curve of minimum churn; the fluctuation in performance (*e.g.,* $N = 24, 28$) is due to the differences in approximating $\frac{1}{N}$. Niagara gives a much smaller churn than ECMP for $N \geq 5$. When $N = 32$, Niagara reduces the churn by 87.5% compared to ECMP.

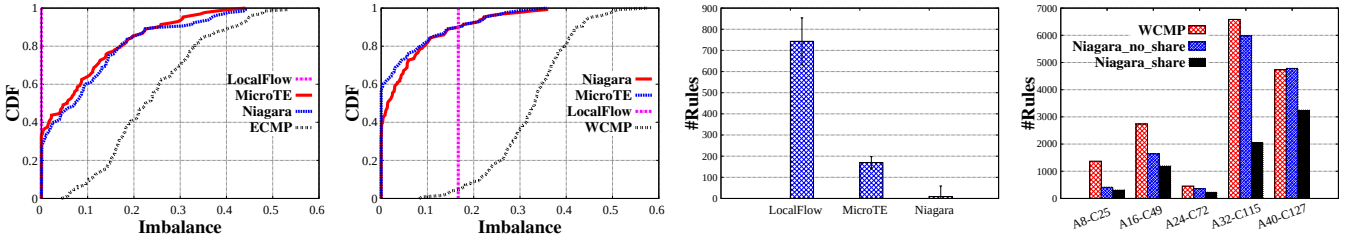
Time. Given a rule-set of 30 rules, if we enumerate the number of lower-priority rules kept in the new rule-set, the incremental computation takes about 30×10ms $= 300$ms to complete, which is in the same order of magnitude as rule insertion and modification on switches (3.3ms to 18ms [38,39]). This is sufficient for updates on the timescale of management tasks. For planned updates, we can also precompute the new rule-set in advance.

7.2 Niagara for Multi-pathing

This section presents Niagara's performance for splitting traffic over multiple equivalent outgoing links by simulating real data center traces [28] on both symmetric and asymmetric topologies [3].

Metrics. We calculate *imbalance_mp* as $\sum_i \max(0, F_i - \frac{W_i}{\sum_k W_k})$, where F_i is the fraction of traffic sent on i-th link and W_i is the weight of i-th link (*i.e.,* the relative bandwidth capacity). It characterizes the total oversubscription when the switch operates at its full bandwidth capacity.

Accuracy in symmetric topology. We simulate 1-hour real packet traces [28] to a popular /16 prefix on a single



(a) Symmetric topology (Imb) (b) Asymmetric topology (Imb) (c) Rules for one dst prefix (d) Rules for multiple prefixes

Figure 16: Multipathing

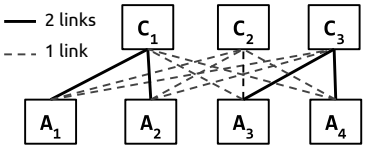


Figure 17: Topology: $N_C = 3, N_A = 4, L_C = 6, L_A = 4$

switch with 4 equal-capacity outgoing links. We slice the trace into 30-second time frames and calculate the imbalance within each time frame.

We compare the splitting performance of Niagara, ECMP, MicroTE [12] and LocalFlow [29]. As MicroTE schedules forwarding paths for ToR-to-ToR flows, we assume that each /24 prefix in the traces correspond to a ToR and compute the utilization and imbalance accordingly. Figure 16(a) shows the CDFs. ECMP performs much worse than Niagara, as it only splits the flow space equally without taking into account the actual flow sizes. ECMP gives $< 10\%$ imbalance in around 10% of the time frames. In comparison, Niagara achieves $< 10\%$ imbalance in 61% of the time frames. MicroTE and Niagara offer similar splitting performance. We notice that Niagara incurs high imbalance for some of the time frames (*e.g.,* 15% time frames have $> 20\%$ imbalance). Upon close examination of the traces, we found that these time frames contain large "elephant" flows; Niagara could not achieve balanced split as it does not split a single flow over multiple links to avoid packet reordering. This also explains why LocalFlow, which splits flows, performs the best.

Accuracy in asymmetric topology. We experiment with a simple asymmetric topology in Figure 17, where there are three core switches and four aggregation switches with 4 links each. We look at the traffic splitting at A_1. A_1 can split traffic destined to A_2 evenly on the 4 uplinks, as A_1 and A_2 have the same bandwidth capacity to all core switches. For traffic to A_3, although A_1 has two links connected to C_1, it cannot send more traffic to C_1 than C_2 or C_3, because C_1 only has one link to A_3. Therefore, A_1 should split traffic destined to A_3 in proportion to $\frac{1}{2} : \frac{1}{2} : 1 : 1$ (*i.e.,* $w = (\frac{1}{6}, \frac{1}{6}, \frac{1}{3}, \frac{1}{3})$) over the 4 uplinks.

Figure 16(b) shows the imbalance CDF for splitting traffic for A_3 at A_1. It is no surprise that Niagara gives a much better result than WCMP. Niagara offers similar performance to MicroTE. For smaller imbalance ($< 2\%$), Niagara performs slightly worse than MicroTE, because it schedules bulks of flows (matching wildcard patterns) rather than ToR-to-ToR flows. This allows Niagara to use much fewer rules than MicroTE. Both Niagara and MicroTE offer $< 10\%$ imbalance for 82% of timeframes. LocalFlow's imbalance is steady at 16.6%, as it always splits traffic evenly.

Rule efficiency. We compare the number of rules generated by Niagara, MicroTE and LocalFlow to split the flows of a single destination prefix evenly (Figure 16(c)). LocalFlow uses the most rules: 743 on average and 854 in the worst case, because it needs finer-grained rules, which even match on bits outside 5-tuple for splitting a single flow, to balance link loads. MicroTE uses fewer rules (149 rules on average and 198 in the worst case) but still significantly more than Niagara, because it schedules ToR-to-ToR traffic. Niagara uses an average of 9 rules (59 in the worst case), which is 1.2% of the rule consumption of LocalFlow and 6% of MicroTE. In fact, *the rule consumption of MicroTE and LocalFlow heavily depends on the traffic pattern* (e.g., *active flows and active ToR pairs), making them hard to scale and less accurate when splitting multiple destination prefixes is needed.* Consider a rule-table with 4000 rules, LocalFlow and MicroTE can at most handle 5 and 26 flow aggregates given similar traffic patterns. In contrast, Niagara can handle more than 400 aggregates.

To compare the number of rules needed to balance multiple flow aggregates between Niagara and WCMP we generate large, asymmetric topologies to examine the total number of rules installed at an aggregation switch. A typical asymmetric topology contains two layers of switches: N_C core switches and N_A aggregation switches. Each core switch has at most L_C links to the aggregation layer; each aggregation switch has at most L_A links to the core layer. The connection algorithm in [3] is used to interconnect two layers of switches. The result is an asymmetric topology that maximizes bisection bandwidth among aggregation switches. We set $L_C = 64$ and $L_A = 192$ and vary the values of $N_C \in [1, L_A]$ and $N_A = 8, 16, 24, 32$. Figure 16(d) compares the number of rules generated by (1) WCMP, (2) Niagara_no_share, where there is no shared default rules and (3) Niagara_shared, where uniform default rules are used. We found that Niagara_share always outperforms WCMP. This figure also shows the rule-saving benefits of shared default rules.

8. CONCLUSION

Niagara advances the state-of-the-art in traffic splitting on switches by demonstrating a new approach that takes a resourceful approach to install carefully optimized flow-rules into hardware switches to closely approximate the desired load distribution and minimize traffic churn during weight changes given the limited rule table capacity.

9. ACKNOWLEDGMENT

We would like to thank the CoNEXT reviewers, our shepherd Pelsser Cristel, Srinivas Narayana, Josh Bailey, Kelvin Zou, Sarthak Grover and Robert MacDavid for their feedback on earlier versions of this paper. This work was supported by the NSF under grant NeTS-1409056.

10. REFERENCES

[1] A. Greenberg, J. R. Hamilton, N. Jain, S. Kandula, C. Kim, P. Lahiri, D. A. Maltz, P. Patel, and S. Sengupta, "VL2: A Scalable and Flexible Data Center Network," SIGCOMM, 2009.

[2] M. Al-Fares, A. Loukissas, and A. Vahdat, "A scalable, commodity data center network architecture," SIGCOMM, 2008.

[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," EuroSys, 2014.

[4] P. Patel, D. Bansal, L. Yuan, A. Murthy, A. Greenberg, D. A. Maltz, R. Kern, H. Kumar, M. Zikos, H. Wu, C. Kim, and N. Karri, "Ananta: Cloud scale load balancing," in *SIGCOMM*, 2013.

[5] R. Wang, D. Butnariu, and J. Rexford, "OpenFlow-based server load balancing gone wild," in *USENIX Hot-ICE*, 2011.

[6] R. Gandhi, H. Liu, Y. Hu, G. Lu, J. Padhye, L. Yuan, and M. Zhang, "Duet: Cloud scale load balancing with hardware and software," in *SIGCOMM*, 2014.

[7] J. W. Anderson, R. Braud, R. Kapoor, G. Porter, and A. Vahdat, "xOMB: Extensible Open Middleboxes with Commodity Servers," ACM/IEEE ANCS, 2012.

[8] A. Gember, A. Akella, A. Anand, T. Benson, and R. Grandl, "Stratos: Virtual Middleboxes as First-Class Entities," Tech. Rep. TR1771, University of Wisconsin-Madison, 2012.

[9] D. Thaler and C. Hopps, "Multipath Issues in Unicast and Multicast Next-Hop Selection." RFC 2991, Nov. 2000.

[10] C. Hopps, "Analysis of an Equal-Cost Multi-Path Algorithm." RFC 2992, Nov. 2000.

[11] M. Al-Fares, S. Radhakrishnan, B. Raghavan, N. Huang, and A. Vahdat, "Hedera: Dynamic flow scheduling for data center networks," USENIX NSDI, 2010.

[12] T. Benson, A. Anand, A. Akella, and M. Zhang, "MicroTE: fine grained traffic engineering for data centers," in *CoNEXT*, 2011.

[13] N. McKeown, T. Anderson, H. Balakrishnan, G. Parulkar, L. Peterson, J. Rexford, S. Shenker, and J. Turner, "OpenFlow: Enabling innovation in campus networks," *SIGCOMM CCR*, 2008.

[14] Broadcom, "High capacity StrataXGS Trident II Ethernet switch series." http://www.broadcom.com/products/Switching/Data-Center/BCM56850-Series.

[15] N. Handigol, M. Flajslik, S. Seetharaman, R. Johari, and N. McKeown, "Aster*x: Load-balancing as a network primitive," in *ACLD*, 2010.

[16] M. Appelman and M. D. Boer, "Performance analysis of OpenFlow hardware," tech. rep., University of Amsterdam, Feb. 2012. http://www.delaat.net/rp/2011-2012/p18/report.pdf.

[17] D. Y. Huang, K. Yocum, and A. C. Snoeren, "High-fidelity switch models for software-defined network emulation," HotSDN, 2013.

[18] O. Rottenstreich and J. Tapolcai, "Lossy compression of packet classifiers," ACM/IEEE ANCS, 2015.

[19] FlowScale. http://www.openflowhub.org/display/FlowScale.

[20] SciPass. http://globalnoc.iu.edu/sdn/scipass.html.

[21] N. Handigol, B. Heller, V. Jeyakumar, B. Lantz, and N. Mckeown, "Reproducible network experiments using container based emulation," in *CoNEXT*, 2012.

[22] "Production quality, multilayer open virtual switch." http://openvswitch.org/.

[23] "GLIF 2014 demos." http://www.glif.is/meetings/2014/demos.

[24] C.-Y. Hong, S. Kandula, R. Mahajan, M. Zhang, V. Gill, M. Nanduri, and R. Wattenhofer, "Achieving High Utilization with Software-Driven WAN," in *SIGCOMM*, 2013.

[25] S. Jain, A. Kumar, S. Mandal, J. Ong, L. Poutievski, A. Singh, S. Venkata, J. Wanderer, J. Zhou, M. Zhu, J. Zolla, U. Hölzle, S. Stuart, and A. Vahdat, "B4: Experience with a globally-deployed software defined wan," in *SIGCOMM*, 2013.

[26] M. Reitblatt, N. Foster, J. Rexford, C. Schlesinger, and D. Walker, "Abstractions for network update," in *SIGCOMM*, 2012.

[27] A. R. Curtis, J. C. Mogul, J. Tourrilhes, P. Yalagandula, P. Sharma, and S. Banerjee, "DevoFlow: Scaling flow management for high-performance networks," in *SIGCOMM*, 2011.

[28] T. Benson, A. Akella, and D. A. Maltz, "Network traffic characteristics of data centers in the wild," IMC, 2010.

[29] S. Sen, D. Shue, S. Ihm, and M. J. Freedman, "Scalable, optimal flow routing in datacenters via local link balancing," in *CoNEXT*, 2013.

[30] M. Alizadeh, T. Edsall, S. Dharmapurikar, R. Vaidyanathan, K. Chu, A. Fingerhut, V. T. Lam, F. Matus, R. Pan, N. Yadav, and G. Varghese, "CONGA: Distributed congestion-aware load balancing for datacenters," in *SIGCOMM*, 2014.

[31] S. Kandula, D. Katabi, S. Sinha, and A. W. Berger, "Flare: Responsive Load Balancing Without Packet Reordering," in *CCR*, 2007.

[32] A. Kabbani, B. Vamanan, J. Hasan, and F. Duchene, "Flowbender: Flow-level adaptive routing for improved latency and throughput in datacenter networks," in *CoNEXT*, 2014.

[33] J. Cao, R. Xia, P. Yang, C. Guo, G. Lu, L. Yuan, Y. Zheng, H. Wu, Y. Xiong, and D. Maltz, "Per-packet load-balanced, low-latency routing for clos-based data center networks," in *CoNEXT*, 2013.

[34] M. Moshref, M. Yu, R. Govindan, and A. Vahdat, "DREAM: dynamic resource allocation for software-defined measurement," in *SIGCOMM*, 2014.

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

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

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

[38] A. Lazaris, D. Tahara, X. Huang, E. Li, A. Voellmy, Y. R. Yang, and M. Yu, "Tango: Simplifying SDN Control with Automatic Switch Property Inference, Abstraction, and Optimization," in *CoNEXT*, 2014.

[39] X. Jin, H. H. Liu, R. Gandhi, S. Kandula, R. Mahajan, M. Zhang, J. Rexford, and R. Wattenhofer, "Dynamic scheduling of network updates," in *SIGCOMM*, 2014.