Efficient Traffic Splitting on SDN Switches

Nanxi Kang[1], Monia Ghobadi[2], John Reumann[3], Alexander Shraer[4], and Jennifer Rexford[1]

[1]Princeton University, {nkang, jrex}@cs.princeton.edu

[2]Microsoft Research, mgh@microsoft.com

[3]Nofutz Networks, reumann@nofutznetworks.com

[4]Google, shralex@google.com

ABSTRACT

Networks often need to balance load over multiple backend servers—or other components like links, paths, or middleboxes—offering the same service. For example, a large cloud provider could easily host tens of thousands of services, each with dozens or hundreds of backend servers. Hash-based approaches like Equal-Cost Multi-Path (ECMP) can achieve a roughly uniform split, or even a weighted split by repeating the same "next-hop" multiple times. However, ECMP has low accuracy for approximating arbitrary weights, scales poorly with many "next-hops," and experiences significant churn when weights change. Instead, a Software-Defined Network (SDN) could split traffic more flexibly by crafting a set of wildcard rules to install in OpenFlow switches. However, existing SDN-based load-balancers either send data packets to the controller, or generate too many rules to approximate the weights. In this paper, we propose Niagara, an SDN-based load-balancing scheme where the controller computes an efficient approximation of the weights for each service, and optimizes the division of the rule-table space across multiple services. Our experiments demonstrate that Niagara improves traffic-splitting accuracy by 77% compared to ECMP, while requiring only 19% of the rules used by Weighted-Cost Multi-Path. Niagara requires 63% fewer rules than previous SDN approaches.

1. INTRODUCTION

Network operators often spread traffic over multiple components (such as links, paths, middleboxes, and backend servers) that offer the same service, to achieve better scalability, reliability, and performance. Managing these distributed resources effectively requires a good way to balance the traffic load, even when different components have different capacity. Rather than deploy dedicated load-balancing appliances, modern networks increasingly rely on the underlying *switches* to split load across the replicas [1–9]. For example, server load-balancing systems [4, 6] use hardware switches to spread client requests over multiple software load balancers (SLBs), which in turn direct requests to backend servers. For each service (or *flow aggregate*), the switch maps each flow to a particular SLB (or *next-hop*). A large cloud provider could easily host tens of thousands of services, with tens or even hundreds of SLBs.

Equal-Cost Multi-Path (ECMP) [10, 11], the most widely used mechanism [1, 2, 4, 6], can split a flow aggregate (typically a destination prefix) *uniformly* over a group of next-hops based on the hash of the packet-header fields. Unequal-Cost Multi-Path (UCMP) or Weighted-Cost Multi-Path (WCMP) [3] handles an uneven weight distribution by repeating the same next-hop multiple times in an ECMP group. However, representing arbitrary weights is expensive (e.g., due to a large number of rules); UCMP and WCMP can only offer a small number of distinct weight distributions. Furthermore, using a hash function leads to 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 [11]. Finally, ECMP hash functions (and the map from hash values to next-hops) are often proprietary, making it difficult for network operators to know which traffic uses which components, complicating network troubleshooting and analysis.

The emergence of open interfaces to SDN switches [12, 13] suggests an attractive alternative, where a controller programs the switch rule table to satisfy a load-balancing goal [5, 14–16]. The rule tables (e.g., TCAM) in commodity hardware switches are optimized for high-speed packet-header matching, making them a great fit for distributing traffic by IP addresses and TCP/UDP port numbers. The controller can also query the traffic counters for each rule to track shifts in the offered load over time. The simplest SDN-based solution [14, 17] directs the first packet of each request to the controller, which reactively installs an exact-match (microflow) rule on the switch. A more efficient approach [5, 15, 16] proactively installs wildcard rules that direct packets matching the same header patterns (e.g., the last few bits of the source IP address) to the same next-hop.

Achieving an accurate and scalable traffic split is challenging. Commodity switches have small rule tables, with thousands or small tens of thousands of rules [18, 19]. Previous solutions [5] do not use the rule-table space efficiently and can only handle a few dozens flow aggregates with several next-hops, while modern networks often require splitting tens of thousands of aggregates over dozens or hundreds of next-hops. This paper presents Niagara, an efficient traffic-splitting algorithm that computes OpenFlow rules that minimize traffic *imbalance* (i.e., the fraction of traffic sent to the "wrong" next-hop, based on the weights), subject to rule-table constraints. Niagara scales to tens of thousands of aggregates and hundreds of next-hops with a small imbalance. After a brief discussion of motivation and related work in the next section (§2), we present the traffic-splitting optimization problem and a high-level overview of Niagara (§3). This paper makes the following contributions:

Accurate traffic-splitting with limited wildcard rules: Niagara approximates load-balancing weights accurately. For each aggregate, Niagara can flexibly trade off accuracy for fewer rules by truncating the approximation (§4).

Efficient sharing of rule-table space across services: Niagara packs rules for multiple flow aggregates into a single table, and allows sharing of rules across multiple aggregates with similar weights (§5).

Minimizing churn during updates: 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) and traffic imbalance. (§6).

We implement the Niagara OpenFlow controller and deploy the controller (i) in a physical testbed with a hardware Pica8 switch interconnecting four hosts (for realistic experiments) and (ii) in Mininet [20] with Open vSwitches and a configurable number of hosts (for experiments scaling the size of the system). We evaluate the performance of Niagara through extensive simulation (§7) and validate the simulation results up to the limit of the hardware testbed. Experiments demonstrate that Niagara reduces imbalance by 77% compared to ECMP with only 19% of the rules of WCMP. Niagara uses 63% less rule space than previous SDN solutions, while achieving the same imbalance. We recently conducted a live demonstration of Niagara at an SDN-based Internet eXchange Point (IXP) in New Zealand [21].

2. TRAFFIC SPLITTING BACKGROUND

2.1 The Case for Efficient Traffic Splitting

Hardware switches are widely used to spread traffic over equivalent components. For example, popular data center topologies [1,2] offer many equal-length paths that switches can use to increase bisection bandwidth. Similarly, modern load balancers rely on switches to distribute user requests over servers [4–6] or middlebox appliances [7–9].

Requirements of traffic split. An efficient traffic-splitting scheme should be accurate and scalable, and minimize churn during updates. In server load balancing, each SLB runs on commodity servers that can handle a limited number of requests. An inaccurate traffic split can easily overload an SLB, thus incurring long latency and drops for client requests.

Furthermore, traffic splitting is applied to a variety of settings. Cloud providers host up to tens of thousands of flow aggregates (i.e., hosted services) which are split over a small tens of next-hops (i.e., SLBs); multi-pathing routing requires an ingress switch to split hundreds of flow aggregates (i.e., IP prefixes) each over a handful of paths (i.e., next-hops) to an egress switch. A flexible scheme should adapt to handle heterogeneity in the numbers of flow aggregates and next-hops while achieving reasonable accuracy.

Finally, failures or changes in capacity can lead to new weights for splitting a flow aggregate over the next-hops; transitioning to a new weight distribution is not free. In load balancing, the update inevitably shuffles flow space among SLBs (i.e., churn) and requires extra operations for consistent handling of TCP connections already in progress [4, 9, 22,23]. An adaptive scheme should update the split with limited overhead.

Weighted split. Flow aggregates can have very different weight distributions. Here, we use load balancing as an example to illustrate (i) weighted distribution of requests for a service, (ii) distinct weight distributions for different services, and (iii) weight changes over time.

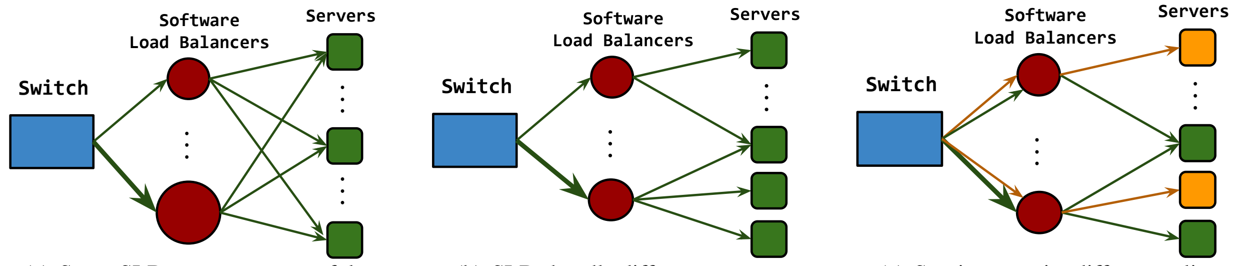
Weighted distributions. Some SLBs are running on more powerful machines making them capable of processing more requests than others (Figure 1(a)). In addition, some SLBs may be geographically closer to more of the backend servers (e.g., in the same rack or podset), leading some SLBs to handle more requests than others (Figure 1(b)).

Distinct distributions. Popular services may have more server replicas in the data center; some service may have a larger deployment in certain racks (Figure 1(c)). These all result in dissimilar numbers of servers for different services at the same SLB and the heterogeneous weight distributions for these services.

Weight changes. Weights are subject to changes of deployment plan and server status (at the timescale of management decisions, i.e., in minutes or hours). Popular services may be gradually rolled out at new servers; maintenance or failure may bring down active servers.

2.2 Related Work

Hash-based approaches. The most common traffic-splitting scheme is ECMP, which assumes equal split over a group of next-hops (e.g., SLBs). ECMP partitions the flow space into equal-sized hash-buckets that each correspond to a next-hop. Packets are hashed on three, or five, or more header fields. Figure 2 shows an example that divides the requests for service 63.12.82.42 over two next-hops 17.12.11.1 and 17.12.12.1. ECMP creates a group (id = 1) by installing two rules in the multipathing table. These two rules map hash value 0 and 1 to 17.12.11.1 and 17.12.12.1, respectively. Then ECMP installs an entry in the forwarding table to direct requests of 63.12.82.42 to this group.



(a) Some SLBs are more powerful (b) SLBs handle different servers (c) Services require different split

Figure 1: Weighted server load balancing examples.

DIP (service VIP)	Next-hops (SLBs)		
	17.12.11.1	17.12.12.1	17.12.13.1
63.12.28.42	$\frac{1}{2}$	$\frac{1}{2}$	0
63.12.28.34	$\frac{1}{6}$	$\frac{1}{3}$	$\frac{1}{2}$

(a) Weights for load balancing two services.

Match	Action
DIP	Group id
63.12.28.42	1
63.12.28.34	2

$\Longrightarrow$

Match		Action
group id	hash value	next-hop
1	0	17.12.11.1
1	1	17.12.12.1
2	0	17.12.11.1
2	1	17.12.12.1
2	2	17.12.12.1
2	3	17.12.13.1
2	4	17.12.13.1
2	5	17.12.13.1

(b) The forwarding table (left) directs packets to an ECMP group; the multipath table (right) forwards packets based on the ECMP group and the hash value.

Figure 2: Hash-based approaches for load balancing.

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
63.12.28.34	1
63.12.28.53	1
63.12.28.27	1
63.12.28.42	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 3: Example wildcard rules for load balancing.

UCMP and WCMP [3] handle unequal split by repeating next-hops in an ECMP group. UCMP achieves $(\frac{1}{6}, \frac{1}{3}, \frac{1}{2})$ split for service 63.12.28.34 with six rules (Figure 2(b)). Given the constrained size of the multipath table (i.e., TCAM), WCMP [3] approximates the desired split with less rules. For example, $(\frac{1}{6}, \frac{1}{3}, \frac{1}{2})$ can be approximated with $(\frac{1}{4}, \frac{1}{4}, \frac{1}{2})$, which takes four rules by only repeating the last-hop twice.

The hash-based approaches incur several drawbacks.

Accuracy and Scalability. While ECMP only handles equal weights, the accuracy and scalability of UCMP and WCMP is restricted by the size of multipath table, typically numbering in a small hundreds or thousands on commodity switches [3]. UCMP and WCMP are *infeasible* when the number of flow aggregates exceed the table size. Given a 8-weight distribution (i.e., 8 SLBs), WCMP takes 74 rules on average and 288 rules in the worst case. In other words, with 4,000 rules, WCMP can only handle up to $^{4000}/_{74} \approx 54$ services, which is too small for load balancing in data centers.

Churn. Updating an ECMP group unnecessarily shuffles packets among next-hops.When a next-hop is removed from a N-member group (or a next-hop is added to an $(N-1)$-member group), at least $\frac{1}{4} + \frac{1}{4N}$ of the flow space are shuffled to different next-hops [11], while the minimum churn is $\frac{1}{N}$.

Visibility. In ECMP, hash functions and mappings from hash values to next-hops are proprietary, preventing effective analysis of forwarding behaviors. Network operators have to examine all next-hops when debugging a flow. Furthermore, making this information public does not make the problem much easier. Recently proposed debugging and verification frameworks [24, 25] rely on the assumption that consecutive chunks of flow space are given the same actions to reduce their complexity. ECMP deviates from this assumption by pseudo-randomly casting flows to next-hops.

SDN-based approaches. SDN supports programming rule-tables in switches, enabling finer-grained control and more accurate splitting. Unlike hash-based approaches, the programmable rule table offers full visibility into how packets are forwarded. A simple solution [14, 17] is to direct the first packet of each client request to a controller, which then installs rules for forwarding the remaining packets of the connection. This approach incurs extra delay for the first packet of each flow, and controller load and hardware rule-table capacity quickly become bottlenecks. A more scalable alternative is to proactively install 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 §4.1.

3. NIAGARA OVERVIEW

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
e	Error tolerance $\|w'_{vj} - w_{vj}\| \le e$
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.

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 distribution. 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 weights. Figure 3(a) shows an example of wildcard rules generated by Niagara for load balancing. Each rule matches on destination IP to identify the service and source IP to forward chunks of 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 [13] and generates tagging rules to group services of similar weight distributions, thus further reducing the number of rules (Figure 3(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 §4.1.2.

3.1 Rule Optimization 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 (usually in $[0.001, 0.01]$), where the actual split is w'_{vj} such that $|w'_{vj} - w_{vj}| \le e$. Ideally, the hardware switch could achieve w'_{vj} with wildcard rules. But small rule-table sizes thwart this, and instead, we settle 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., $c_v = \mathit{numrules}(\{w'_{vj}\})$). 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:

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

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 (§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.

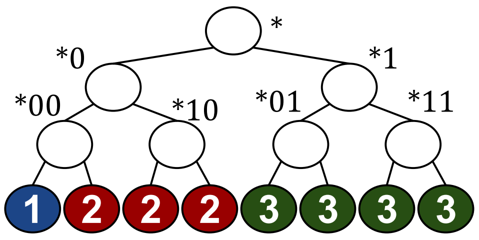
Truncating the approximation to use fewer rules (§4.2): Given the above results, we can truncate the approximation and fit a *subset* of associated rules into the rule table. This results in a *tradeoff curve* of traffic imbalance versus the number of rules.

Packing multiple aggregates into a single table (§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) in today's hardware switches. Fortunately, today's hardware switches have multiple table stages. For example, the popular Broadcom chipset [13] 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 (§5.2): We associate a tag with a group of aggregates with similar weights. We use k-means clustering to identify the



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

groups, and then generate one set of rules for each group. Furthermore, we create a set of default rules of low priority in TCAM, which are shared by all groups.

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

4. OPTIMIZING A SINGLE AGGREGATE

We begin with generating rules to approximate the weight distribution $\{w_{vj}\}$ 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 Weight 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 4(a). To generate the corresponding wildcard rules, an approximate weight $b * 2^{-k}$ 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 4(b).

Shortcomings of the naive solution. The naive approach always expresses b as the "sums" of power of two (for example $\frac{3}{8}$ 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 4(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 $|w'_{vj} - w_{vj}| \leq e$. Each term $x_{jk} = b_{jk} \cdot 2^{-a_{jk}}$, 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 5(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 value of the term based on the errors of both weights. We pick the value that offers the biggest reduction in errors. The current errors of w_{v2} and w_{v3} are $-\frac{1}{3}$ and $\frac{1}{2}$. Let the power-of-two term be x, then the new errors are $-\frac{1}{3} + x$ and $\frac{1}{2} - x$. Hence, the reduction is

$$\begin{aligned}\triangle &= |-\frac{1}{3}| + |\frac{1}{2}| - |-\frac{1}{3} + x| - |\frac{1}{2} - x| \\ &= 2 \times (\min(\frac{1}{3}, x) + \min(\frac{1}{2}, x) - x)\end{aligned}$$

The function is plotted as red line in Figure 6. When $x = 1, \frac{1}{2}$ and $\frac{1}{4}$, the reduction is $-\frac{1}{3}, \frac{2}{3}$ and $\frac{1}{2}$ respectively. In fact, Equation 1 is a concave function, which reaches its maximum value when $x \in [\frac{1}{3}, \frac{1}{2}]$. We choose $\frac{1}{2}$ be the value of the term. 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 5(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 6). For a term that gives

Iteration	$\mathbf{w}'_{v1}$	$\mathbf{w}'_{v2}$	$\mathbf{w}'_{v3}$
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 5: Wildcard rules to approximate $(\frac{1}{6}, \frac{1}{3}, \frac{1}{2})$

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 1 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 of 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 7 visualizes the rule-generation steps for our example from Figure 5(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 7(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 7(b)). Subsequently, $\frac{1}{8}, \frac{1}{32}$ are mapped to $*000, *00100$, which are colored to w_{v1} (Figure 7(c) (d)).

Once all terms have been processed, rules are generated based on the explicitly colored nodes. Figure 5(b) shows the rules corresponding to the final colored tree in Figure 7(d).

Approximate weights with 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

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

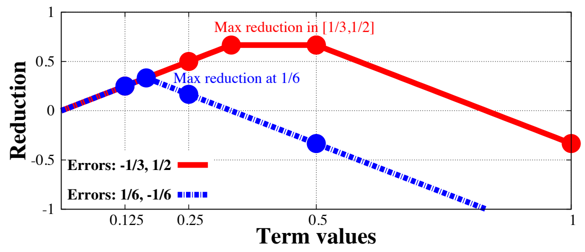


Figure 6: $\triangle$ plots with different errors.

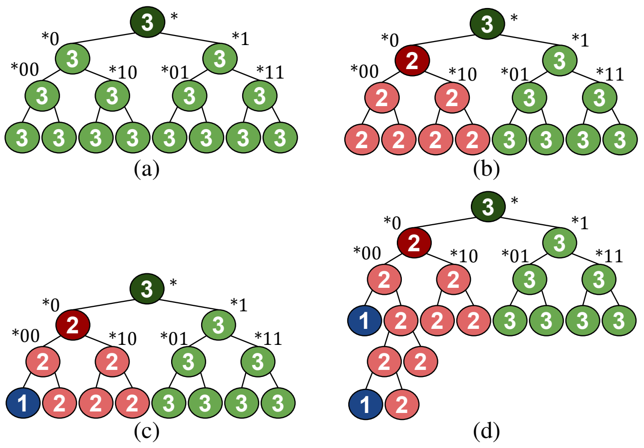


Figure 7: Generate rules using a suffix tree.

algorithm can be extended to handle the unevenness, once the fractions of traffic for suffixes are measured [26, 27]. We still refine the approximation iteratively. In each iteration, a suffix (i.e., a term) is transferred 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 1 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.

4.2 Truncate: Fit Rules in Hardware 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., $t_v \times \sum_j E(w^H_{vj} - w_{vj}, e)$ 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 §4.1 as P^H. For example, when $C = 3$ the rules in Figure 5(b) are truncated into P^H containing the last three rules. Since rules are generated with decreasing $\triangle$ values, which are also the reduction in imbalance, the C lowest-priority rules give the overall biggest reduction of imbalance.

Stairstep plot. Figure 8 shows the imbalance as a function of C. Each point in the plot (r, imb) can be viewed as a

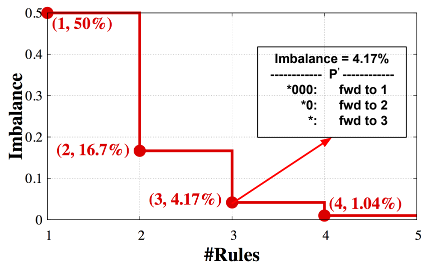


Figure 8: Stairstep curve (imbalance v.s. #rules) for Aggregate v with weights $w_v = \{\frac{1}{6}, \frac{1}{3}, \frac{1}{2}\}$ and $t_v = 1$.

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 (§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

The stairstep plot in § 4.2 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 distributions, traffic volumes and stairsteps of two aggregates.

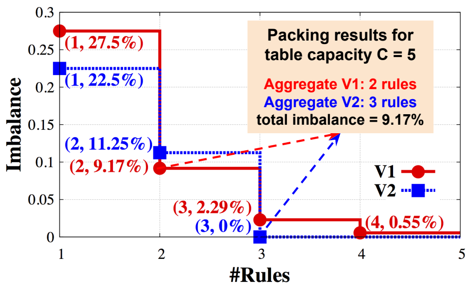
To allocate rules, we greedily sweep through the stairsteps of aggregates in steps. In each sweeping step, we give one more rule to the aggregate with *largest per-step gain* by stepping down one unit along its stairstep. The allocation repeats until the table is full.

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 (§7) demonstrates that this way of handling "heavy hitters" leads to significant gains.

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 Share: Same Rules for Many 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. We employ sharing on different levels, creating three types of rules (with decreasing priority): (1) rules specific to a single aggregate (§4); (2) rules shared among a group of aggregates (§5.2.2), and (3) rules shared among *all* aggregates, called *default rules* (§5.2.1).[2]

5.2.1 *Default rules shared by all aggregates*

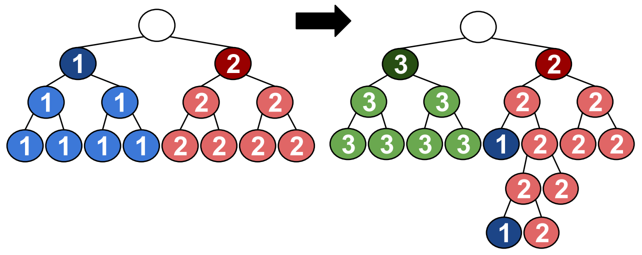
Default rules have the lowest priority and are shared by all aggregates. There are many ways to create default rules, including approximating a certain weight distribution using algorithm in §4. Here we focus on the simplest and most natural one—*uniform default rules* that divide the traffic equally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

Assuming there are M next-hops where $2^k \leq M < 2^{k+1}$, 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 distribution: $w_i^E = 2^{-k}$ for $i \leq 2^k$ and $w_i^E = 0$ otherwise, which can then be improved using more-specific per-aggregate rules.

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 §4.1. Figure 10(a) shows the corresponding suffix tree. Initially, the

[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

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 5(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 (§7).

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 (the choice of k depends on the available rule table space). 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 (§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 (§6.2).

6.1 Compute New Rules Incrementally

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 5(b) and the corresponding suffix tree in Figure 7(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 5(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 7(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.

6.2 Bound Churn with Multi-stage Updates

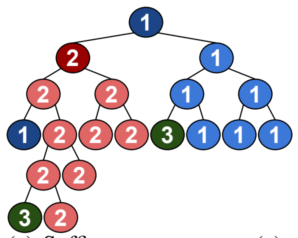
Incurring churn during updates is inevitable. Depending on the deployment, this traffic churn might not be tolerable. Niagara is able to bound the churn by dividing the update process into multiple stages. Given a threshold on accept-

Pattern	Action
∗00100	fwd to 3
∗001	fwd to 3
∗000	fwd to 1
∗0	fwd to 2
∗	fwd to 1

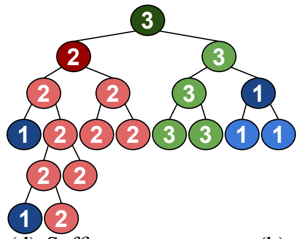
(a) Target rules.

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 corresp. to (a).



(d) Suffix tree corresp. to (b).

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

able churn, Niagara finds a sequence of intermediate rule-sets such that the churn generated by transitioning from one stage to the next is always under the threshold.

Continuing the example in §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 7(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 their update frequency.

7. EVALUATION

In this section, we evaluate the rule-generation algorithm and incremental update.

Weight distribution. The weights of an aggregate depend on various factors such as capacity of next-hops and deployment plans. To reflect this variability, we use three different distribution models to generate weights: Gaussian, Bimodal Gaussian, and Pick Next-hop. Weights of an aggregate v are drawn from these models and normalized so that $\sum_j w_{vj} = 1$. (1) For Gaussian distribution, weights are chosen from $N(4,1)$. Since σ $(= 1)$ is small, the generated weights are close to uniform. It models a setting where an aggregate should be equally split over next-hops. (2) For Bimodal Gaussian distribution, each weight is chosen either from $N(4,1)$ or $N(16,1)$, with equal probability. The generated weights are non-uniform, but aggregates exhibit certain similarity. It models a setting where some next-hops should receive more traffic of an aggregate than others. (3) For Pick Next-hop distribution over M next-hops, we pick a subset of next-hops uniformly at random for one aggregate. Then for those next-hops, we draw the weights from the Bimodal Gaussian distribution. The weights for unchosen next-hops are zero. The generated weights are non-uniform, making it hard to group aggregates. This distribution models a setting where different aggregates should be split over different subsets of next-hops.

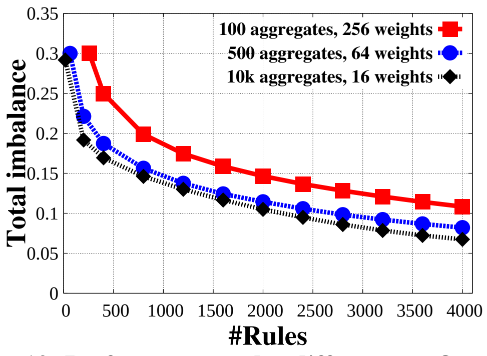


Figure 12: Performance under different configuration.

Traffic distribution. We use (1) uniform traffic distribution and (2) skewed Zipf traffic distribution where the k-th most popular aggregate contributes $1/k$ fraction of the total traffic. The traffic volume is normalized so that $\sum_v t_v = 1$.

Imbalance calculation. We calculate imbalance as

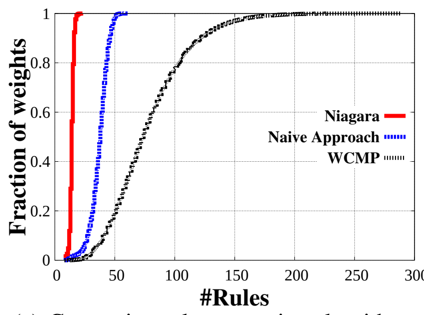
$$\sum_v (t_v \times \sum_j E(w'_{vj} - w_{vj}, 0))$$

instead of $\sum_v (t_v \times \sum_j E(w'_{vj} - w_{vj}, e))$ to avoid the impact of error tolerance e. A total imbalance of 5% to 10% is considered low, depending on the number of next-hops. We use $e = 0.001$ in computing stairsteps of aggregates to terminate the approximation iterations (§4).

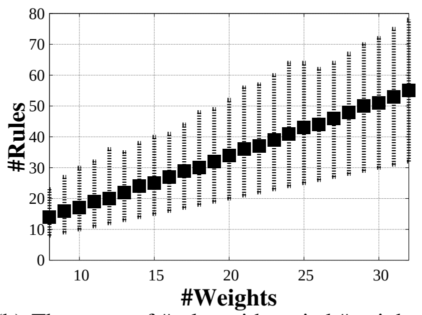
7.1 Rule-Generation Algorithms

All-in-one. Our algorithm scales well under different number of aggregates (N) and number of weights (M). We choose three configurations: (1) $N = 100, M = 256$ models a powerful switch that distributes traffic over its hundreds of outgoing links (or paths); (2) $N = 500, M = 64$ models an enterprise-level load balancer that spreads incoming traffic over a small set of servers; and (3) $N = 10{,}000, M = 16$ models a cloud-scale load balancer that splits requests for tens of thousands of services over the next stage of software load balancers, which in turn direct requests to backends servers behind them [4].

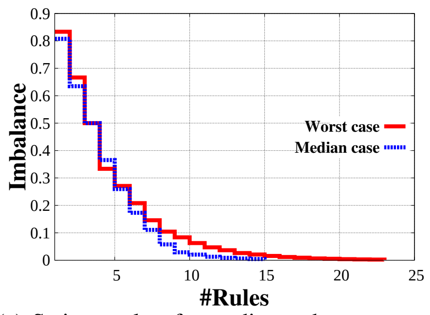
Figure 12 presents the performance of Niagara given Bimodal Gaussian weight distribution and skewed traffic distribution. With 4,000 rules, Niagara achieves an imbalance



(a) Comparing rule-generation algorithms



(b) The range of #rules with varied #weights



(c) Stairstep plots for median and worst cases

Figure 13: Approximating a single aggregate.

of 6.7%, 8% and 10.8% and a maximum over-approximation error [4] of 0.004, 0.0015 and 0.0006 for the three configurations respectively. The leftmost point on the curve denotes how uniform default rules approximate the weights, i.e., ECMP. *ECMP gives roughly* 30% *imbalance for all cases; the over-approximation error even reaches* 0.02. *WCMP is infeasible for any of them.* Niagara improves the imbalance by 64% to 77%.

These three cases, despite different number of aggregates and next-hops, show up similar performance. This is mainly because Niagara takes almost a *linear* number of rules to approximate a given number of weights within the error tolerance. In other words, the number of rules needed to reach the same imbalance is roughly linear in terms of $N \times M$ without grouping, or $G \times M$ with grouping where G is the number of groups. In the plot, we use $G = 50, 150$ and 500 for $N = 100, 500$ and $10,000$ respectively, so $G \times M = 12800, 9600$ and 8000. In addition, as the weight distributions over a smaller number of next-hops are inherently more similar, grouping techniques helps optimize imbalance further. These reasons explain why $N = 10,000, M = 16$ case performs best.

In what follows, we focus on the case of 10,000 aggregates with 16 weights and analyze the contribution of each technique, namely approximating a single aggregate, packing multiple aggregates, sharing default rules among aggregates and grouping.

Approximating a single aggregate. We first examine the number of rules needed to approximate the target weights of a single aggregate. We randomly generate 100,000 distinct sets of weights (8 weights per aggregate). In Figure 13(a), we compare three strategies (§4.1.1): *WCMP*, which repeats next-hop entries in ECMP to approximate weights; *Naive approach*, which rounds weights to nearest multiples of powers of two; *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. The median is 74. It highly depends on the values of weights. In our experiment, a slightly change of weights (e.g., 0.1 to 0.11) cause a dramatic change in number of rules. Naive approach performs slightly better with median 38 rules, but still uses too many rules (61 in the worst case) and too much variation. In comparison, Niagara generated the fewest rules (a median of 14) with small variation. Niagara takes only 19% of the rules used by WCMP and 37% by the naive approach. The reduction demonstrates the effectiveness of using both power-of-two terms and rule priority.

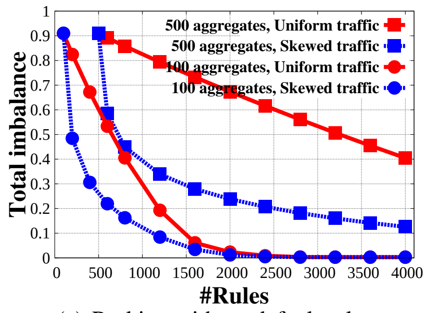
We then repeat the same experiment with a different number of weights M and compare the number of rules (Figure 13(b)). Each marker denotes the median, while vertical bars indicate the minimum and maximum number of rules. Our algorithm performs steadily well under different M values: the number of rules increases linearly, suggesting a roughly *constant* number of rules per weight.

To evaluate our "truncating" technique (§4.2), we use two sets of 8 weights, corresponding to the median and maximum number of rules in Figure 13(b) (14 and 23 rules, respectively), and plot their stairstep curves in Figure 13(c). We observe that given 14 rules, the imbalance of the 'worst case' weight vector is very small (2%). It suggests that we can get quite close to the target weights, even if C is significantly smaller than the number of rules needed to reach the error tolerance.

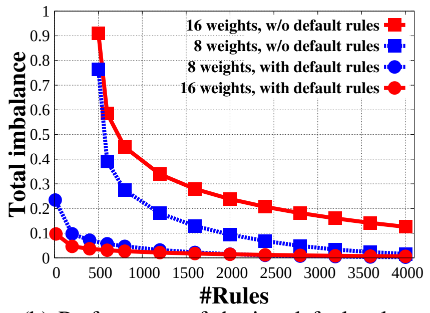
Packing multiple aggregates. Moving on to multiple aggregates, we first evaluate packing (§5.1) assuming aggregates do not share any rules. Each aggregate therefore gets at least one rule. In the experiment, we generate weights from Gaussian model (16 weights per aggregate). Figure 14(a) shows the total imbalance achieved by packing, as a function of rule-table size. The leftmost point on each curve shows the imbalance when every aggregate is given exactly one rule. In all cases, initial imbalance is close to 90%. With Niagara, the imbalance drops linearly for uniform traffic and nearly exponentially for skewed traffic. This is because our packing algorithm prioritizes "heavy" aggregates in rule allocation. By allocating more rules to popular aggregates, we minimize traffic imbalance. For example, packing 100 aggregates with skewed traffic, our algorithm achieves a total imbalance of 1.2% using 2,000 rules. We observed similar results for Bimodal Gaussian and Pick Next-hop weight distributions.

Sharing default rules. Sharing default rules offers a further improvement because (i) we no longer need to give each

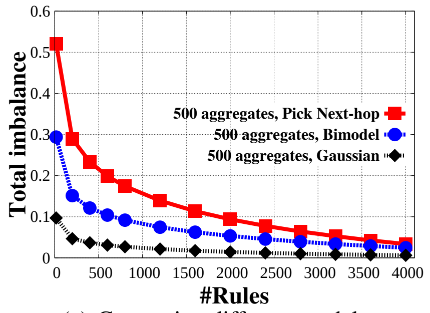
[4] $\max_j \sum_v E(w'_{vj} - w_{vj}, 0)$



(a) Packing without default rules

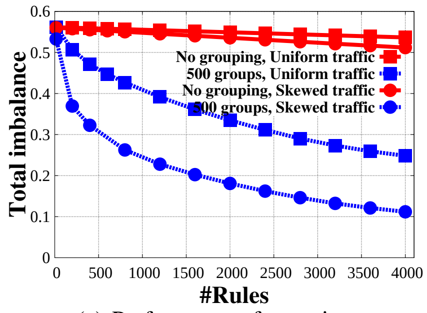


(b) Performance of sharing default rules

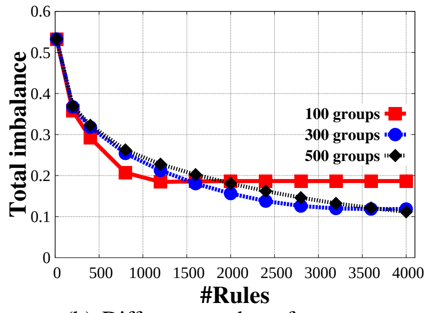


(c) Comparing different models

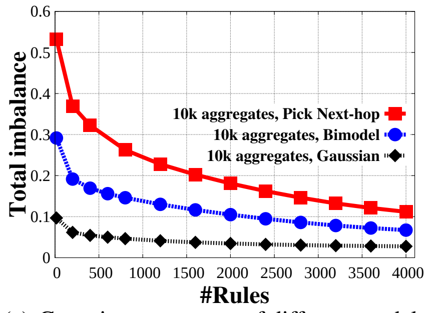
Figure 14: Packing and sharing default rules (16 weights per aggregate).



(a) Performance of grouping



(b) Different number of groups



(c) Grouping aggregates of different models

Figure 15: Grouping $10,000$ aggregates.

aggregate at least one rule during packing and can allocate more rules to heavy aggregates and (ii) default rules provide a good initial approximation and reduce the number of "private" rules for each aggregate. We use uniform default rules in our experiments. Figure 14(b) compares packing 500 aggregates of Gaussian weight distribution, with and without default rules. Using shared default rules achieves a significant reduction in imbalance. The imbalance is reduced from 23.8% to 1.4% for uniform traffic, when $C = 2,000$ and $M = 16$. Moreover, sharing default rules performs better for bigger M values and Gaussian model, as the absolute weights are smaller and closer to uniform. Figure 14(c) compares the performance of sharing default rules for aggregates of different weight distribution models. We achieve the smallest imbalance for Gaussian distribution. Yet, even for Pick Next-hop, the imbalance is 3.3% with $4,000$ rules.

Grouping similar aggregates. Our grouping technique (§5.2.2) groups aggregates with similar weight vectors together. Among the weight distributions, Pick Next-hop is the hardest one to group. Figure 15(a) presents the result of packing $10,000$ aggregates (16 weights per aggregate) of Pick Next-hop model. We cannot pack these aggregates without grouping (there are fewer available rules than aggregates, and all aggregates are equally important). Uniform default rules are not a good initial approximation either (56% initial imbalance). Given $4,000$ rules, the imbalance still exceeds 50%. However, with grouping, the imbalance drops to 24.8% and 11.2% with $4,000$ rules.

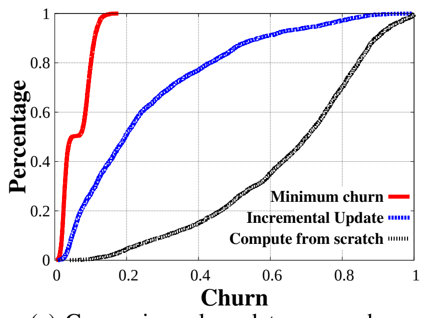
We examine how the number of aggregate groups affects imbalance. We notice that there is a tradeoff between grouping accuracy and approximation accuracy: when the aggregates are classified into more groups, the distance between each aggregate's target weight vector and the centroid vector of its group is reduced, making the grouping more accurate. However, the approximation is less accurate for a bigger number of groups.

Figure 15(b) illustrates this tradeoff comparing the imbalance with 100, 300, and 500 groups. When there are less than 500 rules, classifying the aggregates 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. For larger rule-table sizes, using more groups becomes advantageous, since the distance between each aggregate and its group's centroid, which 'represents' the aggregate during packing, decreases. For example, given $1,500$ rules, 300-group outperforms 100-group.

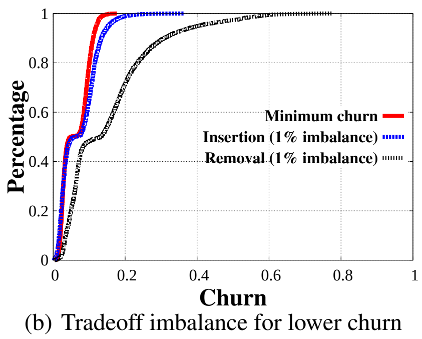
We compare the effectiveness of grouping for different weight models (Figure 15(c)). For a given number of rules, we classify the aggregates into 100, 300, or 500 groups (picking the option which yields the smallest imbalance). With very few rules, the algorithm achieves a reasonably small imbalance. With $4,000$ 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%.

Time. We recorded the running time of the algorithm [5]

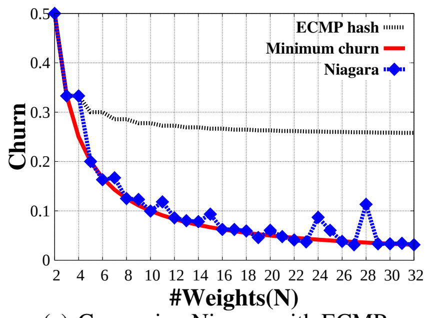
[5]The algorithm assumes that terms can be non-power-of-two.



(a) Comparing rule update approaches



(b) Tradeoff imbalance for lower churn



(c) Comparing Niagara with ECMP

Figure 16: Incremental Update.

on a Ubuntu server with Intel Xeon E5620 CPU (2.4 GHz, 4 core, Model 44, 12 MB cache). Our implementation is single threaded and written in C++. It takes less than 10ms to compute the stairstep curves for a 16-weight distribution ($e = 0.001$). The time of packing grows linearly with the number of aggregates and is dominated by the computation of stairsteps, which can be easily parallelized. For grouping, k-means clustering takes at most 8 sec. to complete, depending on traffic and weight distributions. Skewed traffic and similar weight distributions lead to faster convergence of the clustering results and fewer iterations.

7.2 Incremental Update

We evaluate Niagara's incremental rule computation. Given the old weights for an aggregate, we randomly clear one non-zero weight and renormalize the rest to obtain the new weights, or vice versa (called "removal" and "insertion"). We examine the churn and imbalance of the update.

Effect of keeping old rules. We first compare Niagara with computing from scratch under the same error tolerance ($e = 0.001$). A re-computation results in *minimum* number of rules, but incur huge churn. In contrast, Niagara's update algorithm may lead to extra rules as it keeps some old rules to reduce churn (§6.1). To quantify the effects, we run Niagara against insertion update on $5,000$ sets of 16 weights. For each set of weights, Niagara choose the rules which minimize the churn *without exceeding* the number of rules needed by the re-computation. Figure 16(a) plots the CDF of the churn among $5,000$ sets of weights from Bimodal distribution. The incremental update approach (blue) dramatically reduce the churn; it incurs 20% churn for 50% of the test cases, where the recomputation (black) incurs about 70% churn for the same portion. This suggests that the old rules serve as a good initial approximation for the new weights, thus saving extra rules. We observe similar effect for other weight distributions as well.

Trade imbalance for lower churn. Although Niagara already offers a fairly small amount of churn with minimum number of rules, yet we inevitably miss those rules sets of bigger sizes that gives smaller churn. Truncating those rule sets gives low churn, but leads to increase in imbalance. We plot the improved CDFs of churn in Figure 16(b), when a maximum imbalance of 1% is allowed. The CDF for insertion is much better than the curves in Figure 16(a) and almost coincides with the curve of minimum churn. This suggests that a small tradeoff in imbalance will greatly reduce the churn. Removal update performs slightly worse than insertion, because the old rules related to the cleared weights need to be removed and result in increased churn.

Compare with ECMP and WCMP. The theoretical lower bound of churn for ECMP is $\frac{1}{4} + \frac{1}{4N}$ for removing one member from a N-sized group (or adding one member to $(N-1)$-sized group), where the minimum churn is $\frac{1}{N}$. The churn for WCMP depends on the number of operations on ECMP groups. To compare ECMP and Niagara, we use uniform weight distribution, e.g., N weights of $\frac{1}{N}$. For each N value, Niagara choose the rules that gives the minimum churn, while i) staying within the number of rules needed by a re-computation and ii) incurring less than 1% imbalance. Figure 16(c) presents the comparison of Niagara and ECMP. The blue line with diamonds shows Niagara's performance for insertion update. It 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 typical rule-set of 30 rules for 16 weights, if we enumerate the number of lower-priority rules kept in the new rule-set, the incremental computation takes about $30 \times 10\text{ms} = 300\text{ms}$ to complete, which is in the same order of magnitude as rule insertion and modification on switches (3.3ms to 18ms [28, 29]). It is sufficient for updates on the timescale of management tasks. For planned updates, we can also pre-compute the new rule-set in advance.

8. CONCLUSION

Niagara advances the state-of-the-art in traffic splitting on switches by demonstrating a new approach that programs the hardware rule table to closely approximate the desired load distribution, trading off accuracy for table capacity. Experiments demonstrate that Niagara effectively utilizes a typical $4,000$ rule switch chip to split $10,000$ aggregates resulting in 6.7% imbalance, while ECMP incurs an imbalance of 30% and WCMP cannot even reach this scale.

9. 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," ACM SIGCOMM, 2009.

[2] M. Al-Fares, A. Loukissas, and A. Vahdat, "A scalable, commodity data center network architecture," ACM 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," ACM 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 *ACM 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 *ACM SIGCOMM*, 2014.

[7] J. W. Anderson, R. Braud, R. Kapoor, G. Porter, and A. Vahdat, "xOMB: Extensible Open Middleboxes with Commodity Servers," ACM 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] S. Rajagopalan, D. Williams, H. Jamjoom, and A. Warfield, "Split/merge: System support for elastic execution in virtual middleboxes," in *Usenix NSDI*, 2013.

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

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

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

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

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

[15] FlowScale. `http://www.openflowhub.org/display/FlowScale`.

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

[17] M. Bredel, Z. Bozakov, A. Barczyk, and H. Newman, "Flow-based load balancing in multipathed layer-2 networks using openflow and multipath-tcp," HotSDN, 2014.

[18] 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`.

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

[20] B. Lantz, B. Heller, and N. McKeown, "A network in a laptop: Rapid prototyping for software-defined networks," in *HotNets*, ACM, 2010.

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

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

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

[24] A. Khurshid, X. Zou, W. Zhou, M. Caesar, and P. B. Godfrey, "Veriflow: Verifying network-wide invariants in real time," USENIX NSDI, 2013.

[25] P. Kazemian, G. Varghese, and N. McKeown, "Header space analysis: Static checking for networks," USENIX NSDI, 2012.

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

[27] M. Yu, L. Jose, and R. Miao, "Software defined traffic measurement with opensketch.," in *NSDI*, 2013.

[28] 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.

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