Niagara: Scalable Load Balancing on Commodity Switches

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

[1]Princeton University

[2]Google

ABSTRACT

Internet service providers rely on load balancers to distribute client requests for many web services over backend servers. Dedicated load-balancer appliances are expensive and do not scale easily with traffic demand. Instead, future load balancers should be built from smaller commodity components. Rather than rely exclusively on special-purpose load-balancing software, we argue that data center *switches* should be programmed to perform most of the load-balancing function. Commodity switches offer high-speed packet processing, as well as flexible interfaces for installing rules that forward packets. However, hardware switches have small rule tables, and software switches do not forward packets at high speeds. Our Niagara load-balancing architecture combines the per-packet performance of hardware and the large rule-space of software switches. The hardware switches approximate the load-balancing weights for each service, and the software switches correct for small errors in the approximation and ensure connection affinity during weight changes. Our main contributions are algorithms for (i) approximating the weights for each service, (ii) allocating a limited rule table across many services, and (iii) computing incremental updates to the rules when the weights change. Experiments demonstrate that Niagara can load-balance $10,000$ VIPs using only 4000 hardware rules, while having software switches redirect just 3% of traffic.

1. INTRODUCTION

Cloud providers host many services, each replicated on multiple servers for greater throughput and reliability. A load balancer spreads traffic over the backend servers, while ensuring that all packets from the same request reach the same server (i.e., connection affinity). While many providers rely on dedicated load-balancer appliances [1–4], specialized equipment can be costly, hard to scale dynamically, and become a single point of failure. Some providers implement load balancing in software running on multiple commodity servers [5–8] for better flexibility and scale-out. However, load-balancing software on a general-purpose CPU yields low performance and high power requirements per packet if compared to silicon, forcing cloud providers to allocate a large number of servers for a given level of performance.

The emergence of open interfaces to network switches [9, 10] suggests an attractive middle ground—implementing load balancing *directly on the network devices*. The switch chipsets in commodity hardware switches are optimized for high-speed packet processing at reasonable power and cost [10], and modern software switches leverage support in the kernel and network interface cards to achieve good throughput with large rule tables [11, 12]. While commodity switches cannot support all load-balancing features (e.g., matching on URLs or Cookies in application-layer messages), they are a great fit for the common task of distributing traffic by IP addresses and TCP/UDP port numbers. Implementing load balancing on the switches would allow providers to "ride the wave" of advances in switch performance, without the need to provision additional equipment or create and optimize a custom software solution.

Unfortunately, existing switch-based load-balancing solutions [13, 14] do not scale to handle a large number of services, each with many backend servers. A load-balancer application could direct the first packet of each request to a controller, which then installs rules for forwarding the remaining packets of the connection [13]. However, this solution incurs extra delay for the first packet of each request, and controller load and hardware rule-table capacity quickly become bottlenecks. A more scalable alternative proactively installs coarse-grained rules that direct each *block* of client IP addresses to a server replica [14]. However, this approach generates a large number of rules (matching on the client IP prefix and the service's IP address), still consuming too much of the limited rule-table space in commodity hardware switches. Even more rules are needed to ensure connection affinity across changes in the load-balancing policy.

This paper presents the Niagara load balancer that combines the best features of both *hardware switches* (high throughput for a given cost and power consumption) and *software switches* (large rule tables and flexible packet processing). A small number of hardware switches in a geographic region use coarse-grained rules to divide large traffic aggregates over a larger set of software switches that, in turn, direct finer-grain traffic flows to backend servers. Each service has a public IP address and a set of *weights*—the fraction of requests each backend server should receive. Rather than simply using the hardware switches for equal-cost multipath (ECMP) forwarding [8], we propose algorithms that optimize the use of the limited rule-table space to *approximate* the weights accurately and minimize *imbalance*: the portion of traffic that travels through the "wrong" software switches (i.e., a switch in a different cluster than the chosen backend server). Our algorithm divides the hardware rule table across multiple services, and groups services with similar weights, to approximate the weights for many services using fewer rules.

Any practical load-balancing solution must adapt to changes in the load-balancing policy, while ensuring connection affinity. Computing new rules from scratch can cause substantial traffic *churn*, where a large fraction of traffic is reshuffled or bounced across different clusters. Ideally, churn should be proportional to the required change in policy. To achieve this, production deployments often use full-fledged software servers that run algorithms like consistent hashing [15]. To the best of our knowledge, Niagara is the first solution that achieves the properties of consistent hashing *in hardware*. Niagara computes an incremental change to the existing rules to closely approximate the desired traffic distribution, while balancing the transient churn and long-term imbalance. Moreover, even if the amount of total churn is proportional to the change in policy, network administrators may prefer to further limit the churn to a lower acceptable threshold. Niagara automatically creates an update plan to meet the tight traffic churn objectives by breaking one large update into multiple smaller updates. Finally, rather than storing per-flow state in hardware switches, Niagara ensures connection affinity during policy changes through a combination of *rule cloning* [16] in the software switches and a *consistent update* [17] mechanism that updates the switches in phases.

This paper makes four main contributions:

Scalable load balancer using commodity switches: Niagara scales to a large number of services and backend servers, while leveraging the unique strengths of commodity hardware and software switches (§2).

Algorithm for optimizing rule-table space: Niagara balances load accurately, subject to the rule-table capacity of the hardware switches. For each service, we approximate the weights as sums of powers of two and truncate the approximation to use fewer rules (§3). Then, we pack rules for multiple services into a single table and allow sharing of rules across services with similar weights (§4).

Efficient updates: When the policy changes, Niagara computes an incremental update to the rules that optimizes short-term churn and long-term traffic imbalance, and ensures connection affinity without incurring extra rules in the hardware switches (§5).

Realistic prototype: The prototype uses `iptables` on Linux as its unmodified switch target (§6). Linux is widely-deployed on data center servers used as software switches, and runs embedded inside hardware switches.

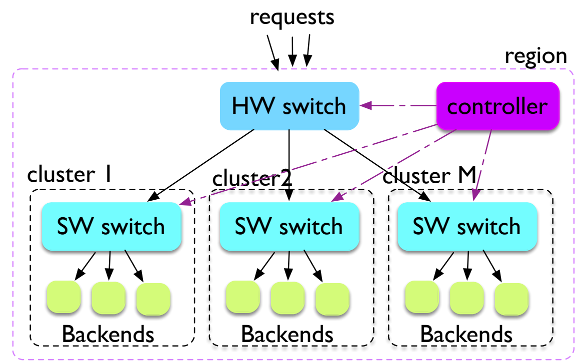


Figure 1: Load balancing using commodity switches.

Experiments demonstrate that Niagara can scale to tens of thousands of services and handle update gracefully (§7). The paper ends with a discussion of related work (§8) and a conclusion (§9).

2. NIAGARA LOAD BALANCER

The Niagara hierarchical load balancer combines the high throughput of hardware with large rule tables of software switches. In this section, we give a high-level overview of Niagara's architecture, formulate the optimization problem for computing the rules in the switches, and outline the five main components of our algorithm.

2.1 Load Balancing on Commodity Switches

Niagara performs load balancing over client requests for multiple replicated services hosted in the same geographic *region*. Each hosted service has a single, virtual IP address (VIP) that corresponds to multiple backend servers, each with its own dedicated IP address (DIP). For services hosted in multiple regions, the provider relies on wide-area load balancing—say, using the Domain Name System (DNS)—to select a particular region (and associated VIP) for each client. Within a region, the backend servers are further grouped into *clusters* that correspond to a data center, or a pod or rack within the same data center, as shown in Figure 1. Backend servers handle client requests and respond directly to clients (DSR, i.e., *direct-server return*).

The switches within a region distribute the client requests over the backend servers in a hierarchical fashion, with high-speed hardware switches dividing traffic over the flexible software switches in each cluster. Today's commodity *hardware* switches have low-cost chipsets that forward traffic at hundreds of Gbps by matching packets against a table of rules. Implemented using Ternary Content Addressable Memory (TCAM), the table can perform wildcard matching on the five-tuple header fields (source and destination IP addresses, source and destination transport ports, and the transport protocol). However, the tables are small, in the thousands or small tens of thousands of rules, and rule updates are slow [18,19]. In contrast, today's *software* switches can forward about 10-20 Gbps per core with large forward-

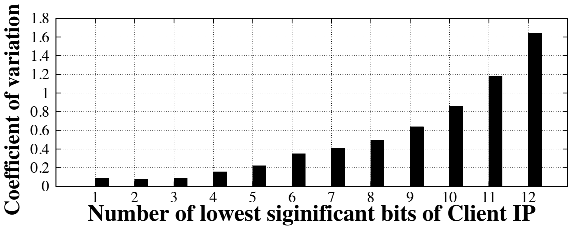


Figure 2: Coefficient of variation of fraction of requests for different bits of the client IP address.

Match		Action
Dest IP (VIP)	**Src IP (client)**	**Next hop (sw. switch)**
63.12.28.34	$*0$	17.12.11.1
63.12.28.34	$*01$	17.12.12.1
63.12.28.34	$*11$	17.12.13.1
63.12.28.42	$*0$	17.12.12.1
63.12.28.42	$*1$	17.12.13.1

Table 1: Approximating weights for two VIPs.

ing tables optimized for exact-match rules, and fast rule updates. Combining a hardware switch with (say) 20 software switches at the lower level leads to a design with low cost, high throughput, high rule capacity, and fast updates.

The simplest load-balancing strategy is for the hardware switches to split all traffic evenly over the clusters using equal-cost multipath (ECMP) forwarding [8], and then rely on the software switches to direct requests for each VIP to the right backends, possibly bouncing flows across clusters. This can result in a large amount of horizontal traffic , especially for popular VIPs with uneven target load distributions (e.g., cluster weights of $\{\frac{1}{2}, \frac{1}{4}, \frac{1}{4}\}$). Instead, the controller could install rules that match on the destination IP address (i.e., the VIP) and some portion of the source IP address (i.e., the client address) to achieve a target load distribution [14]. For example, the rules in Table 1 split the incoming traffic for two VIPs. For the first VIP, the table matches on three different source IP suffixes to approximate the load distribution of $\{\frac{1}{2}, \frac{1}{4}, \frac{1}{4}\}$ among clusters 1, 2, and 3; the traffic of the second VIP is split evenly between clusters 2 and 3.

Fortunately, commodity chipsets [10] can match on any bits in the "five tuple", allowing us to implement a "poor man's hash function" that offers fine-grained control over the load-balancing policy. To split the traffic, we match on the *low-order bits* of the source IP address, which have higher entropy, resulting in a nearly proportional split of the traffic. To justify our approach, we analyze one day of traffic measurements to front-end servers in a commercial cloud deployment. Figure 2 plots the coefficient of variation (i.e., the standard deviation over the mean) for the values of each of the last 12 bits in the source IP address, and these trends persist over time. Values less than 1 are considered a low coefficient of variation. The variation is smallest for the last several bits, which are essentially uniformly randomly set to 0 or 1. Upper bits start to have higher variation because some IP subnets are not fully allocated, meaning that a value of 0 is more common than a value of 1. While we could design our algorithms to account for this skew, we find the last 6-8 bits have very low variation, making it reasonable to assume an even division of traffic when we match on these bits.

Variable	Definition
N	Number of VIPs ($v = 1, \ldots, N$)
M	Number of clusters ($j = 1, \ldots, M$)
C	Hardware switch rule-table capacity
w_{vj}	Target weight for VIP v, cluster j
t_v	Traffic volume for VIP v
e	Tolerable error $\lvert w'_{vj} - w_{vj} \rvert \leq e$
w'_{vj}	Actual weight for VIP v, cluster j
w^H_{wj}	Actual hardware weight for VIP v, cluster j
c_v	Hardware rule-table space for VIP v

Table 2: Table of notation, with inputs listed first.

2.2 Rule Optimization Problem Formulation

The controller needs a good algorithm for computing the rules in the switches, as a function of the per-VIP weights and the per-switch rule-table capacities. For a multi-level hierarchy, the recurring problem is to split traffic from one tier over multiple switches in the next tier; as such, we simplify the discussion by focusing on how a hardware switch should split request traffic for multiple VIPs over multiple clusters. A hardware switch can (at best) approximate the target division of traffic over the clusters, and rely on the software switches to "deflect" misdirected packets to the right cluster. The misdirected traffic consumes extra link bandwidth and experiences longer delay (or higher "stretch") due to the longer path through the network. As such, an important challenge is to minimize the *imbalance*—the fraction of traffic that routes through the wrong software switch.

The VIPs vary in the provisioned server capacity in each cluster, due to differences in the expected request load, server failures and planned maintenance, and the gradual build-out of new clusters. Each VIP v has non-negative weights $\{w_{vj}\}$ for splitting traffic over the M clusters $j = 1, 2, \ldots, M$, where $\sum_j w_{vj} = 1$. (Table 2 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 each VIP can tolerate a given error bound e, where the actual split is w'_{vj} such that $|w'_{vj} - w_{vj}| \leq e$. Ideally, the hardware switch could achieve w'_{vj} with wildcard rules. But small TCAM sizes thwart this, and instead, we settle for the lesser goal of approximating the weights as well as possible (w^H_{vj}), given a limited rule capacity C at the switch.

To compute the weights, the controller solves an optimization problem that allocates c_v rules to each VIP v to achieve weights $\{w^H_{vj}\}$ (i.e., $c_v = numrules(\{w^H_{vj}\})$). VIP v has traffic volume t_v, where some VIPs receive much more requests than others. The goal is to minimize the total traffic imbalance [1] while approximating the weights:

[1]The imbalance only counts the over-approximated weights, which captures the deflected fraction.

Match Dest IP (VIP)	Action Tag (group)
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 (group)	Src IP (client)	Next hop (sw. switch)
1	$*0$	17.12.11.1
1	$*01$	17.12.12.1
1	$*11$	17.12.13.1
2	$*0$	17.12.12.1
2	$*1$	17.12.13.1

Table 3: Sharing rules across VIPs with similar weights.

$$\begin{array}{ll}
\text{minimize } \sum_v (t_v \times \sum_j \max(w^H_{vj} - w'_{vj}, 0)) & s.t. \\
|w'_{vj} - w_{vj}| \le e & \forall v, j \\
w'_{vj} \ge 0,\ w^H_{vj} \ge 0 & \forall v, j \\
\sum_j w'_{vj} = 1,\ \sum_j w^H_{vj} = 1 & \forall v \\
c_v = numrules(\{w^H_{vj}\}) & \forall v \\
\sum_v c_v \le C &
\end{array}$$

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

2.3 Overview of Optimization Algorithm

In solving the optimization problem, we introduce five main algorithmic contributions, starting with the following three ideas:

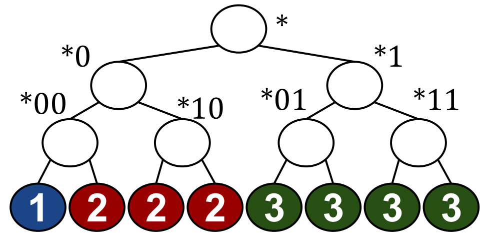
Approximating weights for a single VIP (§3.1): Given weights $\{w_{vj}\}$ for VIP v and error tolerance e, we compute the approximated weights $\{w'_{vj}\}$ and the associated rules for each VIP. The algorithm expands each weight w_{vj} in terms of powers of two (e.g., $\frac{1}{6} \approx \frac{1}{8} + \frac{1}{32}$) that can be approximated using wildcard rules, and decides whether to *over* or *under* approximate each weight to minimize the total number of rules, given the error tolerance.

Truncating the approximation to use fewer rules (§3.2): Given approximation $\{w'_{vj}\}$ and the associated rules, we can fit a *subset* of c_v rules into the hardware table to achieve $\{w^H_{vj}\}$. However, we find that we can achieve an even *smaller* traffic imbalance $t_v \times \sum_j |w^H_{vj} - w'_{vj}|$ by computing the hardware rules and the approximation together. This results in a trade-off curve of traffic imbalance versus the number of hardware rules.

Packing multiple VIPs into a single table (§4.1): Once we know the trade-off curve for each VIP, we can decide how many hardware rules to devote to each VIP to minimize the total traffic imbalance. In each step of the packing algorithm, we devote more rules to the VIP that can 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.

Together, these three parts allow us to make effective use of a small hardware table to divide traffic over the clusters.

A region serving tens of thousands of VIPs, each served by dozen(s) of clusters, can easily overwhelm the small TCAM in today's hardware switches. Fortunately, today's hardware switches have multiple table stages. For example, the popular Broadcom chipset [10] has a table that can match on destination IP prefix and set a meta-data tag that can be matched (along with the five-tuple) in the subsequent TCAM. Niagara can capitalize on this table to map a VIP to a tag—or, more generally, *multiple* VIPs to the same tag. Our fourth algorithmic innovation uses this table:



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

Sharing rules across VIPs with similar weights (§4.2): Multiple VIPs may have similar weights. We associate a tag with a group of VIPs with similar weights. (See Table 3 for an example.) 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 (e.g., ECMP rules) of low priority in TCAM, which are shared by all groups.

Transitioning to new weights (§5): In practice, weights change over time, forcing Niagara to compute *incremental* changes to the rules to control churn, and transition from one set of rules to another while preserving connection affinity.

3. OPTIMIZING A SINGLE VIP

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

3.1 Approximate: Binary Weight Expansion

Naive approach to generating wildcard rules. A possible method to approximate the weights is to pick a fixed IP suffix length k and round every weight to the closest multiple of 2^{-k} such that the approximated weights still sum to 1 [14]. 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 3(a). To generate the corresponding wildcard rules, an approximate weight $b * 2^{-k}$ is represented by b k-bit rules. In practice, these rules may be aggregated by allocating similar suffix patterns to the same weight. The corresponding wildcard rules are listed in Figure 3(b).

Shortcomings of the naive solution. The naive approach falls short, because it 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

$\mathbf{w_{vj}}$	approximation $\mathbf{w'_{vj}}$
$w_{v1} = \frac{1}{6}$	$\frac{1}{8} + \frac{1}{32}$
$w_{v2} = \frac{1}{3}$	$\frac{1}{2} - \frac{1}{8} - \frac{1}{32}$
$w_{v3} = \frac{1}{2}$	$\frac{1}{2}$ (pool)

Table 4: Approximations of weights within $e = 0.02$.

allows *subtraction* as well as longest-match *rule priority*. For example, $\frac{3}{8}$ can be expressed as $\frac{4}{8} - \frac{1}{8}$ which in our example achieves the same approximation with one less rule, as illustrated in Figure 3(c). In this example, 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. Specifically, 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. Table 4 illustrates the approximations for a VIP with weights $\frac{1}{6}$, $\frac{1}{3}$, and $\frac{1}{2}$, under tolerable error $e = 0.02$. 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 first show how to compute the approximations, followed by generating rules based on the approximations.

3.1.1 Approximate the weights

Pool. When approximating weights, we must ensure that the sum of approximated weights remains 1. This is achieved by selecting a special weight called "pool", whose approximation is derived from the aggregated approximations of other weights, i.e., $w'_{pool} = 1 - \sum_{j \neq pool} w'_{vj}$. In Table 4, we pick the biggest weight $w_{v3} = \frac{1}{2}$ as the pool, and approximate w_{v1} and w_{v2}.

To approximate a single weight w_{vj}, we compute two expansions of power-of-two terms: a lower bound L_j and an upper bound U_j, where $L_j \leq w_{vj} \leq U_j$. Initially, $L_j = 0$ and $U_j = 1$. These two bounds are then *iteratively* tightened by either adding a term to the previous lower bound or subtracting a term from the previous upper bound. During this process, the differences between the bounds and w_{vj} (called error) decrease *exponentially*. Eventually, the computation stops when the error is within the tolerable error. The iterations of computing expansions for $w_{v2} = \frac{1}{6}$ are shown in Table 5.

Using this technique, at each iteration we obtain two approximations for non-pool weights: lower-bound (L) and upper-bound (U). We then choose one of them as the final approximation for the weight. The goal is two-fold: 1) minimize the number of generated rules, and (2) ensure that the error of the "pool" is within the tolerable error. To this end, we introduce two strategies: *exhaustive search* which evaluates all possible choice combinations (one approximation per weight) and picks the best set of approximations, and a *greedy heuristic*, which chooses one of the two bounds to approximate each weight, greedily attempting to minimize the number of rules while always ensuring that the "pool" error is within the tolerable error.

Iteration	L	U	$\mathbf{w_{v1} - L}$	$\mathbf{U - w_{v1}}$
0	0	1	0.1667	0.8333
1	$\frac{1}{8}$	$\frac{1}{4}$	0.0417	0.0833
2	$\frac{1}{8} + \frac{1}{32}$	$\frac{1}{8} + \frac{1}{16}$	0.0104	0.0208
3	–	$\frac{1}{8} + \frac{1}{16} - \frac{1}{64}$	–	0.0052

Table 5: Steps to compute the upper and lower bounds to approximate $w_{v1} = \frac{1}{6}$ with $e = 0.02$. Note that the second iteration U is obtained by adding $\frac{1}{16}$ to first iteration L; third iteration U is obtained by subtracting $\frac{1}{64}$ from second iteration U.

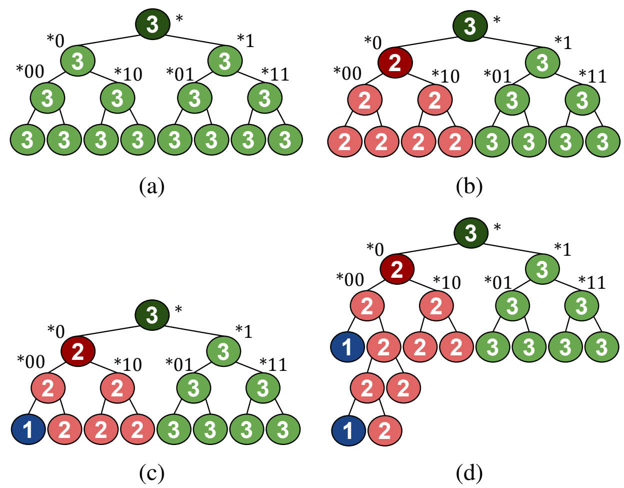


Figure 4: Generate rules using a suffix tree.

3.1.2 Generate rules based on approximations

Given an approximation w'_{vj} for a non-pool weight, we generate its corresponding rules by mapping the power-of-two terms of the approximation to nodes of a suffix tree. Each node in the tree represents a 2^{-k} fraction of traffic, where k is the depth of the node (or, equivalently, the suffix length). Figure 4 visualizes the rule generation steps for our example from Table 4 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 "pool" (Figure 4(a)). Color j represents that the node belongs to w'_{vj}. Each uncolored node implicitly *inherits* the color of its closest ancestor. We use dark color for nodes that are explicitly colored, and light color for the unassigned nodes.

To find the mapping between the approximation terms $w'_{vj} = \sum_k x_{jk}$ and the suffix tree's nodes, we first sort the powers-of-two terms corresponding to non-pool weights in descending order of their absolute values ($|x_{jk}|$). In the ex-

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}
$*$	fwd to 3	(pool)

Table 6: Wildcard rules corresponding to Figure 4(d).

ample of Table 4, the sorted terms are $\frac{1}{2}$, $\frac{1}{8}$, $-\frac{1}{8}$, $\frac{1}{32}$, $-\frac{1}{32}$. Then, one by one, the terms are mapped to nodes as follows. Let x_{jk} be the current term being considered, then:

- If $x_{jk} > 0$, we map it to a node representing $|x_{jk}|$ fraction of traffic with color "pool". The node is then re-colored to j. In this way, weight w'_{vj} gets $|x_{jk}|$ fraction of traffic from the "pool". In the example, we map the term $\frac{1}{2}$ in w'_{v2} to $*0$ (Figure 4(b)).

- If $x_{jk} < 0$, we map it to a node representing $|x_{jk}|$ fraction of traffic with color j. The node is then re-colored to "pool". In this way, weight w'_{vj} gives $|x_{jk}|$ fraction of traffic to the "pool".

- Whenever there exist a pair of terms $0 < x_{j_1k_1} = -x_{j_2k_2}$, instead of transferring a fraction of traffic from w_{vj_2} to the pool and then again from the pool to w_{vj_1}, we can "cancel" these terms out by mapping both to a node corresponding to $|x_{j_1k_1}|$ fraction of traffic with color j_2 and re-coloring it with j_1. In this way w_{vj_1} gets $|x_{j_1k_1}|$ traffic from w_{vj_2}. In the example, we cancel $-\frac{1}{8}$ in w'_{v2} and $\frac{1}{8}$ in w'_{v1} by mapping both to node $*000$ (Figure 4(c)) and assigning color 1 to this node.

Once all terms have been processed, rules are generated based on the explicitly colored nodes. Table 6 shows the rules corresponding to the final colored tree in Figure 4(d).

3.2 Truncate: Fit Rules in Hardware Table

Given the restricted hardware rule-table size, some generated rules might not fit in the hardware. Therefore, we need to separate the rules into hardware rules P^H and software rules P^S. In this sense, P^H achieves a coarse-grained approximation of the weights while $numrules(P^H)$ stays within the rule table size C; rules in P^S are distributed across software switches to further approximate the weights until the tolerable error e is met. As a consequence of this truncation, some packets will hit the "wrong" software switch and need to be "corrected" by P^S. We capture this packet deflection as *imbalance*, defined as $t_v \times \sum_j \max(w^H_{vj} - w'_{vj}, 0)$, where t_v is the expected traffic volume for VIP v and w^H_{vj} is the approximation of weight w_{vj} given by P^H.

Rule-set truncation. A simplistic solution could be to truncate the rule-set generated in Section 3.1 into two parts: C lower-priority rules become P^H and the rest become P^S. For example, if $C = 3$ the rules in Table 6 are truncated into P^H containing the last three rules and P^S containing the top

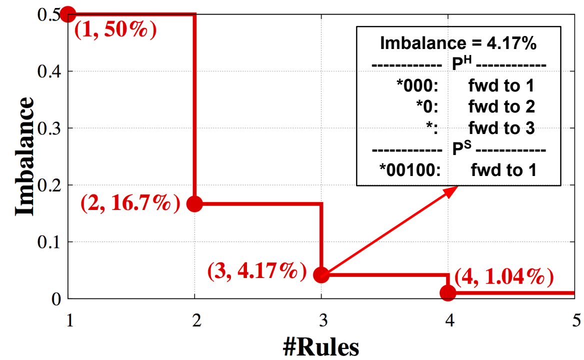


Figure 5: Stairstep curve (imbalance v.s. #rules) for VIP v with weights $w_v = \{\frac{1}{6}, \frac{1}{3}, \frac{1}{2}\}$ and $t_v = 1$.

rule. Note, however, that this approach does not minimize imbalance, since the approximation algorithm did not consider the hardware rule-table size when computing the rules; it is possible that a different set of C rules exists that achieves a smaller imbalance.

Finding P^H for each value of C. Instead of truncating after the rules have already been computed, we determine the partition into P^H and P^S for every possible value of C *during* the computation of weight approximations (having this information for different C values becomes important when we pack rules for different VIPs into the same hardware table in Section 4.1).

The intuition is that truncating the rule-set is equivalent to truncating terms from a weight approximation. We can therefore use "intermediate" lower and upper bound expansions in the iterative computation of approximations described in Section 3.1.1. Furthermore, instead of independently performing algorithm iterations for each weight, we perform iterations for all weights together by tightening either the upper or the lower bound expansion of a single weight at each step. We choose the next bound to tighten to be the one having the maximal error from all currently available bounds of all non-pool weights. After each tightening step, we use the brute-force or heuristic methods in Section 3.1.1 to generate a new set of rules P^H. Therefore each step in this process results in a fresh set of rules. The algorithm completes once both bounds of every weight are within the tolerable error. Note that during this process we may obtain multiple rule-sets with identical size. In this case, we choose the one with smaller imbalance. Hence, the result is one set of hardware rules P^H for each value of C, and the associated imbalance. For each set P^H, we compute the corresponding set P^S by continuing the approximations in P^H. We also optimize P^S such that the stretch of packet deflection is minimized.

Stairstep plot. Figure 5 shows the imbalance as a function of C. Each point in the plot (r, imb) can be viewed as a *cost* for table space r, and the corresponding *gain* in imbalance *imb*. This curve helps us determine the gain a VIP can have from a certain number of allocated hardware rules,

VIP	Weights	Traffic Volume
VIP_1	$w_{11} = \frac{1}{6}, w_{12} = \frac{1}{3}, w_{13} = \frac{1}{2}$	$t_1 = 0.55$
VIP_2	$w_{21} = \frac{1}{4}, w_{22} = \frac{1}{4}, w_{23} = \frac{1}{2}$	$t_2 = 0.45$

Table 7: Weights and traffic volume of VIP_1 and VIP_2.

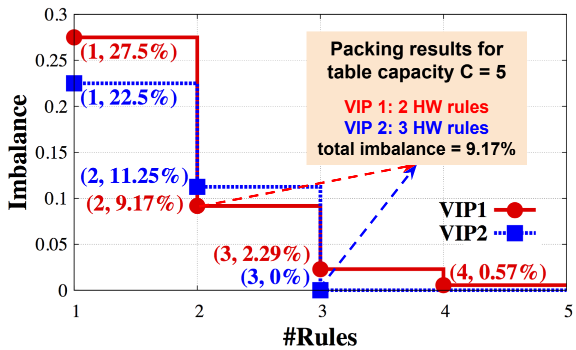


Figure 6: Packing example for VIPs in Table 7.

which is used in packing rules for multiple VIPs into the same hardware table (§4.1).

4. CROSS VIP OPTIMIZATION

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

4.1 Pack: Divide Rules Across VIPs

The stairstep plot in Section 3.2 presents the tradeoff between the number of hardware rules assigned to a VIP and the imbalance the software switches must correct. When dividing rule-table space across multiple VIPs, we use their stairstep plots to determine which VIPs should have more rules, to minimize the total traffic imbalance. For example, Table 7 shows the weight distributions and traffic volumes for two VIPs, with the corresponding stairsteps in Figure 6.

To allocate hardware rules, we greedily sweep through the stairsteps of VIPs in steps. In each sweeping step, we give one more rule to the VIP 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 VIPs (Figure 6) using five hardware rules. We begin with giving each VIP one rule, resulting a total imbalance of 50% $(27.5\% + 22.5\%)$. Then, we decide how to allocate the remaining three rules. Note that VIP_1's per-step gain is 18.33% $(27.5\% - 9.17\%)$, which means that giving one more rule to VIP_1 would reduce its imbalance from 27.5% to 9.17%, while VIP_2's gain is 11.25% $(22.5\% - 11.25\%)$. We therefore give the third rule to VIP_1 and climb one step down along its curve. The per-step gain of VIP_1 becomes 6.88% $(9.17\% - 2.29\%)$. Using the same approach, we give both the fourth and fifth rules to VIP_2, because its per-step gains $(22.5\% - 11.25\% = 11.25\%$ and $11.25\% - 0\% = 11.25\%)$ are greater than VIP_1's. Therefore, VIP_1 and VIP_2 are given two and three hardware 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 VIP_1's stairstep and $(3, 0\%)$ in VIP_2's.

Iteration	L	U
0	–	$\frac{1}{2}$
1	$\frac{1}{2} - \frac{1}{2}$	$\frac{1}{2} - \frac{1}{4}$
2	$\frac{1}{2} - \frac{1}{2} + \frac{1}{8}$	$\frac{1}{2} - \frac{1}{4} - \frac{1}{16}$
3	$\frac{1}{2} - \frac{1}{2} + \frac{1}{8} + \frac{1}{32}$	$\frac{1}{2} - \frac{1}{4} - \frac{1}{16} - \frac{1}{64}$

Table 8: Approximate $w_{v1} = \frac{1}{6}$ with initial upper bound $\frac{1}{2}$. (Compare with Table 5.)

A natural consequence of our packing method is that popular VIPs are allocated more hardware rules, while VIPs with lighter volume may be (mostly) load-balanced in software. Our evaluation (§7) demonstrates that this way of handling "heavy hitters" leads to significant gains.

4.2 Share: Same Rules for Multiple VIPs

In practice, a region may have tens of thousands of VIPs, each with multiple clusters. Given the small TCAM in today's hardware switches, we may not always be able to allocate even one rule to each VIP. Thus, we are interested in *sharing* hardware rules among multiple VIPs. We employ sharing on different levels, creating three types of rules (with decreasing priority): (1) rules specific to a single VIP (described in §3); (2) rules shared among a group of VIPs, and (3) rules shared among *all* VIPs (called *default rules*).[2] Below, we first discuss default rules and then explain how to group similar VIPs.

4.2.1 Default rules shared by all VIPs

Default rules are shared by all VIPs, including those without any other hardware rules. These rules have the lowest priority and match only on the source (client) IP address, rather than the destination (VIP) IP address. There are many ways to create default rules, including approximating a certain weight distribution using algorithm in Section 3. Here we focus on the simplest and most natural one—equal-cost multi-path (ECMP) rules that divide the traffic equally among clusters. Our evaluation (§7) demonstrates that using such rules dramatically reduces the total traffic imbalance.

Assuming there are M clusters where $2^k \leq M < 2^{k+1}$, we construct 2^k ECMP rules matching suffix patterns of length k and distributing traffic evenly among the first 2^k clusters. These ECMP rules provide an initial approximation w^E of the target weight distribution: $w^E_i = 2^{-k}$ for $i \leq 2^k$ and $w^E_i = 0$ otherwise, which can then be improved using more-specific rules, such as per-VIP rules described in Section 3.1. We revisit the example $\{\frac{1}{6}, \frac{1}{3}, \frac{1}{2}\}$. In Table 8, we show how to improve the approximation for $w_{v1} = \frac{1}{6}$, starting from

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

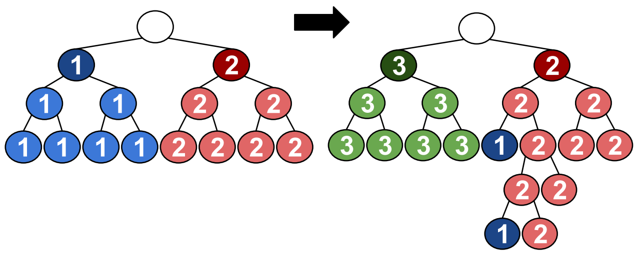


Figure 7: Initial(left) and final(right) suffix trees for $w'_{v1} = \frac{1}{2} - \frac{1}{2} + \frac{1}{8} + \frac{1}{32}$, $w'_{v2} = \frac{1}{2} - \frac{1}{8} - \frac{1}{32}$, $w'_{v3} = \frac{1}{2}$ (pool).

Rules	Pattern	Action
Rules for VIP_v	$*00101$	fwd to 1
	$*001$	fwd to 1
	$*0$	fwd to 3
Shared ECMP rules	$*0$	fwd to 1
	$*1$	fwd to 2

Table 9: Rules for weight distribution $\{\frac{1}{6}, \frac{1}{3}, \frac{1}{2}\}$.

$U = w_1^E = \frac{1}{2}$ (note that the initial lower bound is missing—intuitively, we must improve upon the initial approximation given by ECMP rules). The pool is chosen to be the weight w_{vj} which maximizes $w_{vj} - w_j^E$, i.e., is farthest from its target weight. The pool in our example is w_{v3}.

Figure 7 shows the corresponding suffix tree. Initially, the tree is colored according to ECMP rules. [3] Next, we remove the leading $1/2$ terms from w'_{v1} and w'_{v2}—these terms are already captured by the initial ECMP coloring of the tree. We then process the remaining terms ($-\frac{1}{2}$, $\frac{1}{8}$, $-\frac{1}{8}$, $\frac{1}{32}$, $-\frac{1}{32}$) as explained in Section 3.1.2 and obtain the final rules (Table 9). The total number of rules is five, compared to four rules without using ECMP (Table 6). However, only three of the five rules are "private" to VIP v, as the two ECMP rules are shared among all VIPs. This illustrates that default rules may not save space for one (or even several) VIPs, but will usually bring significant table space savings when the number of VIPs is large, as we demonstrate in Section 7.

4.2.2 Grouping VIPs with similar weights

To further conserve table space, we bundle VIPs having similar weight distributions into groups. We save space by tagging such VIPs with the same group identifier (Table 3).

We use k-means clustering to group VIPs with similar weights. The centroid of each group is computed as the average weight vector of its member VIPs; to prioritize "heavy" VIPs, the average is weighted using t_v (the expected traffic volume of VIP v). We begin by selecting the top-k VIPs with highest traffic volume as the initial centroid of the groups (the choice of k depends on the available hardware table space). Then, we assign every VIP to the group whose centroid vector is closest to the VIP'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., a threshold of 0.01% in our evaluation).

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

5. SEAMLESS CHANGES TO WEIGHTS

Load-balancing policies change over time, due to servers failures, service updates, and cluster maintenance. When the weights for a VIP change, Niagara computes new rules while minimizing the packet deflections caused by (i) churn *during* the update (to ensure connection affinity) and (ii) traffic imbalance *after* the update (due to inaccuracies of approximation in hardware). 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 (§5.1). 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 (§5.2). In both cases, Niagara pushes new rules on software switches (SWSs) before updating the hardware switch (HWS), and uses rule versioning to perform the update consistently. Our approach applies new rules to new connections as soon as possible, while ensuring connection affinity (§5.3).

5.1 Compute New Rules Incrementally

When weights change, Niagara computes new rules to approximate the new set of 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 6 and the corresponding suffix tree in Figure 4(d). In this example, any solution must shuffle at least $\frac{1}{3}$ of the flow space (assuming a negligible tolerable error e), which determines the minimal churn.

Minimize imbalance (recompute hardware and software 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 6 become "fwd to 3" and vice versa. This approach minimizes the traffic imbalance by making the best use of hardware rules. 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 4(d) means that nearly $\frac{2}{3}$ of existing connections must be deflected to their "old" clusters to preserve connection affinity. Second, it requires significant updates to hardware; this slows down the update process, since hardware switches are much slower than software switches in responding to changes. As a result, this approach does not work well when weights change frequently.

Minimize churn (recompute only the software rules).

[3] The root is not colored with "pool" since it is shadowed by the default ECMP rules.

Rules	Pattern	Action
P^S	∗00111	fwd to 1
	∗0011	fwd to 1
	∗01	fwd to 1
P^H	∗00100	fwd to 1
	∗000	fwd to 1
	∗0	fwd to 2
	∗	fwd to 3

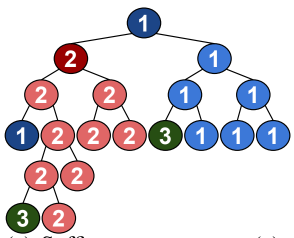
Figure 8: Update P^S only (worst imbalance, least churn).

Rules	Pattern	Action
P^S	∗00100	fwd to 3
P^H	∗001	fwd to 3
	∗000	fwd to 1
	∗0	fwd to 2
	∗	fwd to 1

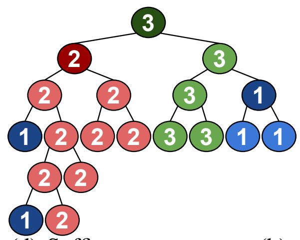
(a) Target rule-set.

Rules	Pattern	Action
P^S	none	
P^H	∗00100	fwd to 1
	∗000	fwd to 1
	∗11	fwd to 1
	∗0	fwd to 2
	∗	fwd to 3

(b) Intermediate rule-set.



(c) Suffix tree corresp. to (a).



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

Figure 9: Rule-sets (and corresponding suffix trees) installed during the transition from $\{\frac{1}{6}, \frac{1}{3}, \frac{1}{2}\}$ **to** $\{\frac{1}{2}, \frac{1}{3}, \frac{1}{6}\}$.

An alternative strawman is to keep the hardware rules "as is," and rely on SWSs to handle the change in weight distribution. This is done by treating the old rules as default rules and using the algorithm in Section 4.2. Figure 8 illustrates this approach by adding three software rules that shift $\frac{1}{4} + \frac{1}{16} + \frac{1}{32}$ of traffic from w_{v3} to w_{v1}. This approach minimizes churn but results in significant imbalance by putting the burden of absorbing the weight change permanently and entirely on software switches. In the example, both the churn and the new imbalance are roughly $\frac{1}{3}$.

Strike a balance (updating hardware and software rules). 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 9(a) and the suffix tree in Figure 9(c). In this case, the algorithm achieves $\frac{1}{32}$ imbalance and $\frac{1}{32} + \frac{3}{8}$ churn (only slightly higher than the minimum of $\frac{1}{3}$).

5.2 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 acceptable 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 from Section 5.1, suppose that the maximum acceptable churn is $\frac{1}{4}$. In this case, the churn created by a direct transition from the old rules in Table 6 to the new rules in Figure 9(a) would be $\frac{1}{32} + \frac{3}{8}$, exceeding the churn threshold. Hence, Niagara finds an intermediate stage shown in Figure 9(b)(d). Specifically, 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 9(d)) is obtained by replacing the subtree ∗11 of the old one (Figure 4(d)) with the new one's (Figure 9(c)). Then, transitioning from the intermediate suffix-tree in Figure 9(d) to the one in Figure 9(c) recolors only $\frac{1}{32} + \frac{1}{8}$ of the flow space (less than the threshold $\frac{1}{4}$) and therefore we can transition directly to the rules in Figure 9(a) after the intermediate stage.

We note that performing a multi-stage update naturally results in lengthy update process for VIPs with frequent weight changes. To mitigate this, Niagara either demotes such VIPs entirely to SWSs or rate limits their update frequency.

5.3 Preserve Connection Affinity

When performing updates, we must ensure that ongoing TCP connections remain pinned to the same backend ("connection affinity") regardless of where the new policy would send the flow. We could wait for old flows to terminate before applying a new policy [14] but this could delay updates indefinitely. The alternative, storing per-flow state in HWS, does not scale. Niagara chooses to track the connection-to-backend mapping at the software layer. Each time a new L4 connection is observed, an SWS maintains its routing decision in a table whose priority supersedes the load-balancing policy, thus pining connection mappings across changes in routing tables. HWS is freed from L4-related tracking tasks. All state tracking is done in abundant DRAM on SWSs.

Connection tracking. The idea of letting SWSs automatically generate a new micro-flow rule for each L4 routing choice follows the local-autonomy principle of DevoFlow [16]. Niagara's local micro-flows gain global significance whenever rules are updated as flows may bounce between switches. At those times, it is important to synchronize local microflows among all SWS. This could be done either via (i) eager periodic broadcast from the switches, (ii) controller-initiated poll-and-broadcast when there is a global policy update, or (iii) lazy schemes in which switches query upon receiving unexpected packets.

Policy versioning. Large sets of forwarding rules are tracked and applied atomically using versions. We tie each packet to the active policy via version tag in the packet. HWS always holds exactly one policy version and labels each routed packet accordingly. SWSs match their version to the routing label on the received packet.

A global policy update consists of the five steps as shown in Figure 10. We first install the new policy version (both

POLICY-UPDATE($version\_id, P^H, P^S$)

1 Install P^H and P^S on SWSs
2 SWSs apply new policy
3 Synchronize connection registry
4 Install P^H on HWS
5 Remove unmatched rules on SWSs

Figure 10: Global policy update scheme

hardware and software rules) on all SWSs (Step 1). These new rules remain shadowed until HWS stamps the new version number into forwarded packets; alternatively, we may instruct SWSs to re-stamp the new version individually (Step 2). Now, new connections are forwarded using the new version while existing connections remain routed as before. Note, all new flows are now being deflected to their target SWS by (another) SWS. Then, we synchronize the connection registries among all SWSs (Step 3) to ensure that any existing connection established under an old version is recognized and forwarded consistently by all SWSs. We then install new hardware rules at HWS (Step 4), so the "new connections" no longer need to be deflected by SWSs. However, connections from previous versions need to be deflected until they terminate. Finally, we garbage collect unmatched rules on SWSs.

In fact, the whole system applies the new policy to incoming packets after Step 2, irrespective of HWS's forwarding behavior. We could choose to never update HWS and things would continue to work. Updating HWS (Step 4) is important to reduce deflection. Not updating HWS keeps forwarding all packets of pre-existing connections to their "correct" SWSs per some previous policy version. This becomes less desirable as old connections die out and traffic churn begins to consist only of the new connections (established under the new policy); Once HWS is updated, only old connections need to be deflected by SWSs. We demonstrate this churn tradeoff between new and old flows when we evaluate the update dynamics of our prototype (§7).

6. NIAGARA PROTOTYPE

We prototype Niagara to show how to apply the output of our algorithm to an actual forwarding system comprising multiple switches (both stateless hardware and stateful software), as well as how to update switches consistently and ensure connection affinity. The source code and experiment setup can be accessed from [20].

System design. Figure 11 shows the network of switches and backends, with a router connecting to clients. All devices attach to a shared L3 network and connect via GRE tunnels. We configure the router to direct all incoming requests to HWS, which then forwards to SWSs and eventually BEs. Return traffic is not tunnelled via SWSs but instead uses direct server return (DSR). We chose to implement both HWS and SWS atop regular Linux servers using *iptables* to reduce our implementation work at the expense of forwarding performance. Iptables can be configured remotely via `ssh` by the controller. Iptables allows the controller to create a collection of routing tables that match on arbitrary packet-header fields and set per-packet metadata. In addition, iptables can be configured to track L4 connections.

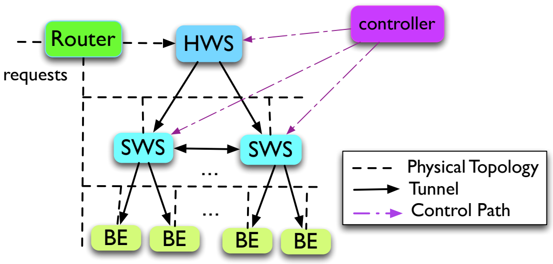


Figure 11: Niagara prototype architecture overview.

Packet processing. The controller begins by creating one routing table at each switch for the current policy version. Each version corresponds to one specific VLAN-tag. Upon receiving a packet, the switch translates the VLAN tag to a per-packet internal metadata *vmark*, and uses it to select the routing table. The rules inside the routing table, which are directly translated from the output of the algorithm, set additional metadata *rmark* denoting the next-hop for the packet. At the network ingress, HWS sets the first *vmark* (a.k.a. VLAN-tag) on any incoming packet.

Connection tracking. IPConntrack in iptables maintains a state table of active local flows, where we save the next-hop information (*rmark*) for the first packet of the connection. SWSs are configured to first check for each incoming packet if it belongs to an existing connection. If so, the packet is immediately forwarded according to the previously-saved *rmark*. Therefore, each flow is routed only once, when adding the flow to the state table; policy changes do not impact ongoing connections.

Since HWS update can reshuffle flow-to-SWS mappings, we need to synchronize connection mappings across SWSs. `Conntrackd` was built as an iptables add-on exactly for this purpose. In our prototype, we configured multicast state replication among SWSs. This multicast group effectively combines the local state tables into one logically shared global connection table, ensuring that packets of the same connection are forwarded to the same BE, even if they traverse different SWSs. To prevent conntrack state from blowing up, we must ensure fast garbage collection as connections expire. To this end, we set up route-exceptions at the BEs (also through iptables) to route all SYN-ACK and FIN packets through SWSs instead of sending them DSR. In practice there are a few more packets that need this exception treatment (e.g., ICMP messages, RST, etc.).

Rule updates. We implement the update mechanism (§5.3) in our prototype. The update first creates the tables in SWSs that contain the complete rule-set of the new version. When all SWSs are primed with the new version, we change the *vmark* at HWS. However, we do not to install the new rules at HWS immediately (§5.3). Instead, we proceed with a later HWS update to minimize traffic churn (§7).

Failures. The current prototype keeps an unbounded history of policy versions to avoid having to deal with wrap-around version numbers and out-of-sync SWSs.

Practical observations. The HWS rule-set is completely stateless, matching only on L3 bits and can be mapped to the tables of a standard packet-forwarding chip like Broadcom's. The use of GRE tunnels is not always necessary (e.g., L2 fabrics) and GRE causes trouble as it reduces MTU size, consumes CPU cycles, and often lacks NIC offload support. In L2 fabrics it may suffice to drive packets to the right SWS by forwarding the packet to the corresponding destination MAC address. Finally, we realize that multicasting the connection table is not going to scale. Instead we propose synchronizing each SWS against a few replicas of a sharded global connection table. Then on policy update, the global controller would initiate a push of connection-table entries from this sharded repository to SWSs, as fallback, SWSs would poll the sharded connection state table on receipt of unexpected packets.

7. EVALUATION

In this section, we evaluate the rule-generation algorithm through simulation and the update mechanism in the prototype. Our algorithm makes effective use of the constrained hardware table. With the additional grouping support, we achieve 2.9% to 11.7% imbalance for $10,000$ VIPs using $4,000$ rules, while the approach that only uses ECMP rules [8] incurs an imbalance as high as 53.3%. Even without grouping, we could load balance 500 VIPs with an imbalance of 3% using $4,000$ rules, much better than 9.7% to 52.1% imbalance by the ECMP-only approach. By further analyzing each technique, we find that our algorithm (i) uses much fewer rules (median 16) than the naive approach (median 22) to approximate a single VIP; (ii) prioritizes popular VIPs in packing rules; (iii) greatly saves hardware rules through using default rules; and (iv) explores similarity among VIPs and can load-balance more VIPs (than the rule table size) with grouping support.

We present the evaluation of our Niagara prototype in handling policy updates. Our prototype achieves a smooth transition from the old policy to the new one, while ensuring connection affinity. Furthermore, we can effectively minimize real-time churn by choosing the timepoint to update the hardware switch. In our experiment, the churn is reduced by 47.2% compared to updating all switches together.

7.1 Rule-Generation Algorithms

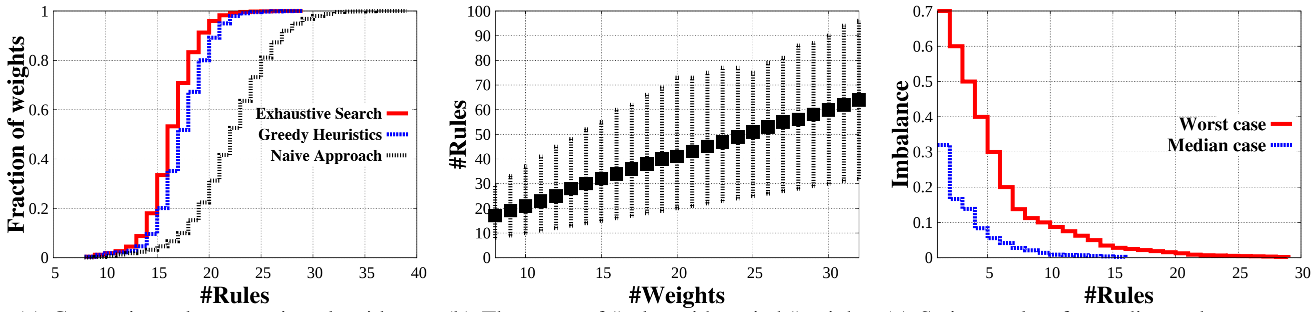
Weight distribution. The cluster weights of a VIP depend on various factors such as size of the cluster, deployment plans, and backend failures. To reflect this variability in our evaluation, we use three different distribution models to generate weights: Gaussian, Bimodal Gaussian, and Pick Cluster. Weights of a VIP 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 is small, the generated weights are close to uniform. It models a setting where a VIP has equal-sized deployment in all clusters. (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 VIPs exhibit certain similarity. It models a setting where a VIP has bigger deployment in some clusters than others. (3) For Pick Cluster distribution over M clusters, we pick a subset of clusters uniformly at random for one VIP. Then for those clusters, we draw the weights from the Bimodal Gaussian distribution. The weights for unchosen clusters are zero. The generated weights are non-uniform, making it hard to group VIPs. This distribution models a setting where different VIPs are served by different subsets of clusters. In the experiment, the number of clusters M is 8 or 16. We set tolerable error e to 0.1%.

Traffic distribution. We evaluate Niagara using both uniform traffic distribution and skewed Zipf traffic distribution where the k-th most popular VIP receives $1/k$ fraction of the total traffic. The traffic is normalized so that $\sum_v t_v = 1$.

All-in-one. Figure 14(c) demonstrates the benefit of all our techniques put together. [4] We load-balance $10,000$ VIPs of different weight distributions with skewed traffic. The number of weights per VIP is 16. For a given number of rules, we classify the VIPs into 100, 200, or 300 groups (picking the option which yields the smallest imbalance). Even with very few hardware rules, the algorithm achieves a reasonably small imbalance. With $4,000$ hardware rules, we reach 2.9% and 6.9% imbalance for the Gaussian and Bimodal Gaussian models respectively, and 11.7% imbalance for Pick Cluster, which is much tougher to group. In what follows, we analyze the contribution of each technique, namely single VIP rule generation, packing multiple VIPs, sharing default rules among VIPs and grouping.

Single VIP rule generation. We first examine the number of rules needed to approximate the target weight vector of a single VIP. We randomly generate $100,000$ distinct weight vectors (8 weights per VIP). In Figure 12(a), we compare three strategies (§3.1.1): *exhaustive search*, which gives an optimal solution with exponential time complexity; *greedy heuristic*, which solves the problem in linear time, and *naive approach*, which only uses positive approximation terms. The exhaustive search generated much fewer rules (a median of 16) than the naive one (a median of 22). It demonstrates that our algorithm greatly reduces the number of rules by using both positive and negative terms and canceling terms through rule priorities. Since the heuristic's performance closely track the exhaustive search strategy, we use the heuristic throughout the remainder of this section. We then repeat the same experiment with a different number of weights M and compare the number of rules (Figure 12(b)).

[4] We calculate imbalance as $\sum_v (t_v \times \sum_j \max(w^H_{vj} - w_{vj}, 0))$ instead of $\sum_v (t_v \times \sum_j \max(w^H_{vj} - w'_{vj}, 0))$ to avoid the impact of e in computing w'_{vj}.



(a) Comparing rule-generation algorithms (b) The range of #rules with varied #weights (c) Stairstep plots for median and worst cases

Figure 12: Single VIP rule generation.

Each marker denotes the median, while vertical bars indicate the minimum and maximum values. We can see that the number of rules increases linearly, suggesting our algorithm performs steadily well under different M values.

To evaluate our "truncating" technique (§3.2), we take two weight vectors of size 8, corresponding to the median and maximum number of rules in Figure 12(b) (16 and 29 rules, respectively), and plot their stairstep curves in Figure 12(c). We observe that given 16 hardware rules, the imbalance of the 'worst case' weight vector is very small (2.5%). It suggests that we can get quite close to the targeted weights, even if C is significantly smaller than the number of rules needed to reach the tolerable error .

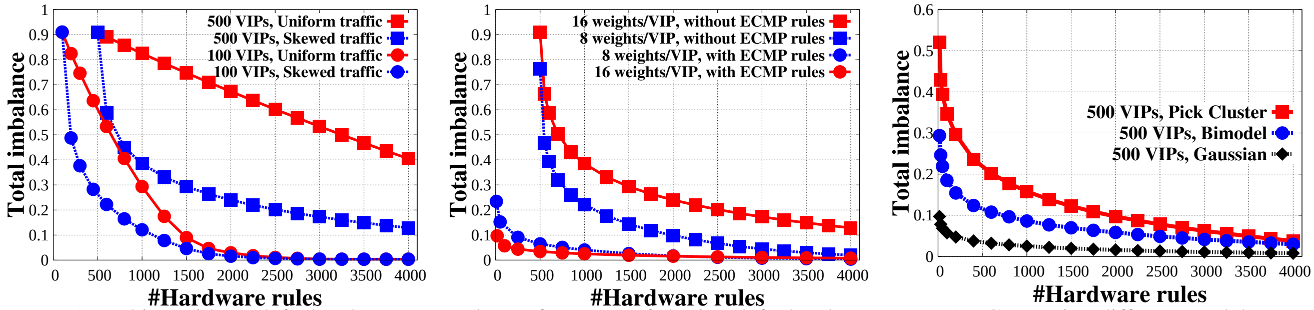
Packing multiple VIPs. Moving on to multiple VIPs, we first evaluate packing (§4.1) assuming VIPs do not share any rules. Each VIP therefore gets at least one hardware rule. In the experiment, we generate weights from Gaussian model (16 weights per VIP). Figure 13(a) shows the total imbalance achieved after packing, as a function of hardware table size. The leftmost point on each curve shows the imbalance when every VIP is given exactly one rule. In all cases, initial imbalance is close to 90%. Observe that the imbalance drops linearly for uniform traffic and nearly exponentially for skewed traffic, suggesting that our packing algorithm uses hardware rules efficiently. Furthermore, skewed traffic leads to a much faster drop, as our packing algorithm prioritizes "heavy" VIPs in rule allocation. By allocating more rules to popular VIPs, we minimize traffic imbalance. For example, packing 100 VIPs with skewed traffic and $2,000$ hardware rules, our algorithm achieves a total imbalance of 1.5%. We observed similar results for Bimodal Gaussian and Pick Cluster weight distributions.

Sharing default rules. Sharing default rules offers a further improvement because (i) we no longer need to give each VIP at least one rule during packing and can allocate more rules to heavy VIPs, and (ii) default rules provide a good initial approximation and reduce the number of "private" rules for each VIP. We use ECMP default rules in our experiments. Figure 13(b) compares packing 500 VIPs of Gaussian weight distribution, with and without default rules. With the same number of rules, using shared default rules achieves a significant reduction in imbalance. For example, the imbalance is reduced from 22% to 4%, when $C = 1,000$ and $M = 8$. Moreover, we observe that sharing default rules performs better for bigger M values and the Gaussian model, as the weights are closer to uniform. Figure 13(c) compares the performance of sharing default rules for VIPs of different weight distribution models. We achieve the smallest imbalance for Gaussian distribution. Yet, even for Pick Cluster, the imbalance is less than 4% with $4,000$ rules.

Grouping similar VIPs. Our grouping technique (§4.2.2) clusters VIPs with similar weight vectors together. Among the weight distributions, Pick Cluster is the hardest one to group. Figure 14(a) presents the result of packing $10,000$ VIPs (16 weights per VIP) of Pick Cluster model. When the traffic distribution is uniform, we cannot pack these VIPs without grouping (there are fewer available rules than VIPs, and all VIPs are equally important). ECMP default rules are not a good initial approximation either (53% initial imbalance). Given $4,000$ rules, the imbalance still exceeds 50%. However, with grouping, the imbalance drops to 26% with $4,000$ rules. When the traffic is skewed, the imbalance decreases from 20% to 12% with $4,000$ rules.

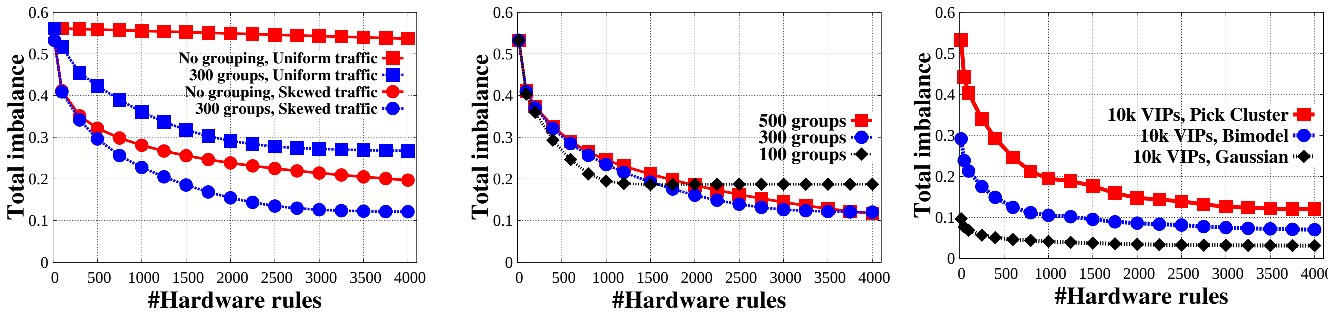
We examine next how the number of VIP groups affects imbalance. We notice that there is a tradeoff between grouping accuracy and approximation accuracy: when the VIPs are classified into more groups, the distance between each VIP's target weight vector and the centroid vector of its group is reduced, making the grouping more accurate. However, the approximation is less accurate for a bigger number of groups. Figure 14(b) illustrates this tradeoff comparing the imbalance with 100, 300, and 500 groups. When there are less than 500 rules, classifying the VIPs into 100 groups performs best since it is easier to pack 100 groups than, e.g., 300 groups, while the centroids of groups still give a reasonable approximation for VIPs. For larger hardware rule tables, using more groups becomes advantageous, since the distance between each VIP and its group's centroid, which 'represents' the VIP during packing, decreases. For example, given $1,500$ rules, 300-group outperforms 100-group.

Time. We recorded the running time of the algorithm 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 Python 2.7.3. It takes less than 30



(a) Packing without default rules (b) Performance of sharing default rules (c) Comparing different models

Figure 13: Packing and sharing ECMP default rules (16 weights per VIP).



(a) Performance of grouping (b) Different number of groups (c) Grouping VIPs of different models

Figure 14: Grouping.

sec. to compute the stairsteps curves of 100 VIPs ($e = 0.1\%$) and then perform packing using 4000 hardware rules. The time grows linearly with number of VIPs and is dominated by the computation of stairsteps, which can be easily parallelized. For grouping, k-means clustering takes 30 sec. to 480 sec. to complete, depending on traffic and weight distributions. Skewed traffic and similar weight distributions across VIPs lead to faster convergence of the clustering results and fewer iterations.

7.2 Rule-update Mechanism

We evaluate our rule update mechanism in our prototype. The setup includes one HWS, two SWSs (SW1 and SW2), and two BEs (BE1 and BE2) serving a single VIP v. We connect BE1 to SW1 and BE2 to SW2. Thus, BE1 is the only backend of SW1 and similarly for SW2. Each SWS sends all requests to its only backend unless the packets should be deflected. We inject client traffic destined to VIP v into the network, and monitor the bytes received at BEs as well as packet deflection.

In the experiment, we transition from weights $\{w_{v1} = \frac{3}{4}, w_{v2} = \frac{1}{4}\}$ to weights $\{w_{v1} = \frac{1}{4}, w_{v2} = \frac{3}{4}\}$. Both the old policy and the new policy achieve weights using hardware rules. The old policy map $*00$ to BE2 and the rest to BE1; the new policy. change the mapping of $*01$ and $*10$ to BE2. During the update, the existing connections of $*01$ and $*10$ should be pinned to BE1, but new connections should be directed to BE2. We start eight TCP connections to VIP v matching patterns 000, 001,..., 111, where connections end asynchronously and new connections of the same pattern start afterwards. Then, we update switches and keep recording the packets received by BEs and traffic churn during the update.

Figure 15 shows three runs of the experiment, where the only difference is the timing of HWS update:

Update HWS and SWSs together (top): At the beginning, the eight TCP connections create $3:1$ throughput ratio at BE1 and BE2. No packets are deflected. At 90 sec. we update HWS and both SWSs. As a consequence, for active flows 001, 010, 101, and 110, HWS sends their packets to SW2 and SW2 directs them to BE1. Therefore, although the throughput of BEs do not change, we see a sudden increase in traffic churn consisting of old flows. This traffic churn gradually disappears, as these flows finish. Finally, the throughput ratio at BE1 and BE2 becomes $1:3$.

Update HWS after all old flows end (center): If we update HWS *after* all old flows end, we see no traffic churn immediately after updating SWSs (at 90 sec.), since packets from old flows still hit their original SWSs. However, churn increases as new flows arrive. For example, when a new flow 001 starts, HWS sends its packets to SW1 based on the old rules; SW1 applies the new rules, and redirects packets to BE2. Eventually, HWS is updated after old flows end (160 sec.), stopping the deflection of new flows.

Update HWS at an optimized time (bottom): Since new flows keeps expanding and old flows are shrinking, we can find a "sweet-spot" that minimizes the traffic churn. In the

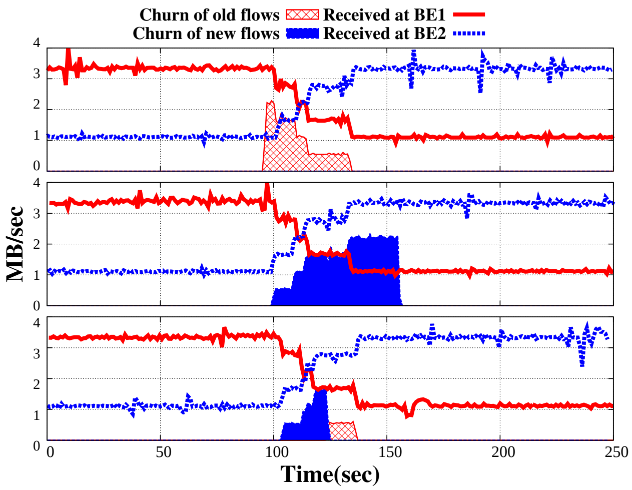


Figure 15: Top: update of SWSs and HWS together; Center: update HWS after old flows finish; Bottom: update HWS at an optimized time.

example, we update HWS at 125 sec. The churn contains only new flows before the update and old flows afterwards.

8. RELATED WORK

Hierarchical load balancers: Ananta [8] is a hierarchical load balancer that uses ECMP in hardware switches to spread traffic over custom software multiplexers. In contrast, Niagara optimizes the rules in the hardware switches for more accurate load balancing, and leverages commodity software switches. Niagara also has novel algorithms for incremental rule updates while preserving connection affinity.

Load balancing using coarse-grained rules: Previous work [14] introduced an algorithm for computing coarse-grained rules for splitting traffic over multiple backends (the "naive approach" in Section 3.1). Niagara's algorithm makes more effective use of rule-table space by using both positive and negative terms, and introduces novel techniques for truncating, packing, and sharing, and for incremental updates.

Network support for connection affinity: Niagara ensures connection affinity (§5.3) by extending and combining rule cloning [16] and per-flow consistent updates [17]. The update mechanism, coupled with Niagara's algorithm for computing incremental rule changes, results in efficient and seamless updates to the load-balancing policy.

9. CONCLUSION

Niagara advances the state-of-the art in software-based load-balancing by demonstrating a new approach that combines hardware and software switches. Hardware is programmed to closely approximate the desired load distribution, trading off accuracy for hardware table capacity, while software switches correct any residual traffic imbalance.

Niagara effectively utilizes limited hardware resources: a typical 4k rule switch chip can load balance 10k VIPs. This is 4k rules well-spent, as it reduces the traffic redirection across switches by 77% compared to previous ECMP-only plus software solutions. In practical terms, Niagara increases effective throughput by 37%, or in other words, effective link utilization increases from 65% to 89%.

Only programming hardware to forward to backend-servers directly without any detour through software will produce better link utilization. However, hardware-only load-balancing suffers from long policy-update delays; to avoid disrupting existing flows, flows must quiesce before their routing can be changed. Niagara accepts traffic redirection during updates as inevitable. Instead of avoiding traffic reshuffling, Niagara bounds and minimizes it. Compared to instant policy updates, typical of pure software approaches, Niagara reduces the amount of traffic redirection during the update by 47.2%. Unlike pure hardware-rules-only load-balancing, Niagra can promptly apply updates, without waiting for all old flows to expire.

10. REFERENCES

[1] "F5 networks, BIG-IP." `http://www.f5.com/products/big-ip`.

[2] Brocade, "ADX application delivery switches." `http://www.brocade.com/products/all/application-delivery-switches`.

[3] Citrix, "Netscaler application delivery controller." `http://www.citrix.com/products/netscaler-application-delivery-controller/`.

[4] Barracuda, "Load balancer application delivery controller." `https://www.barracuda.com/products/loadbalancer`.

[5] HAProxy. `http://haproxy.1wt.eu/`.

[6] "Linux virtual server." `http://www.linuxvirtualserver.org/`.

[7] Embrane, "Heleos-powered load balancer." `http://www.embrane.com/products/load-balancer`.

[8] 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.

[9] 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 Computer Communnications Review*, vol. 38, pp. 69–74, Apr. 2008.

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

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

[12] "Intel DPDK: Data plane development kit." `http://dpdk.org/`.

[13] N. Handigol, M. Flajslik, S. Seetharaman, R. Johari, and N. McKeown, "Aster*x: Load-balancing as a network primitive," in *Architectural Concerns in Large Datacenters (ACLD)*, 2010.

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

[15] D. Karger, E. Lehman, T. Leighton, R. Panigrahy, M. Levine, and D. Lewin, "Consistent hashing and random trees: Distributed caching protocols for relieving hot spots on the world wide web," in *ACM STOC*, 1997.

[16] 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.

[17] M. Reitblatt, N. Foster, J. Rexford, C. Schlesinger, and D. Walker, "Abstractions for network update," in *ACM SIGCOMM*, pp. 323–334, Aug. 2012.

[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, pp. 43–48, 2013.

[20] "Extended version of this paper upon request."

APPENDIX

A. UNEVEN TRAFFIC DISTRIBUTION

In the paper, we assume uniform requests distribution over the last octet of src_ip address, i.e., $*0$ denotes half portion of the total requests. This is the ideal case. In reality, we often observe unbalanced distribution. For example, the network receives more requests from *src_ip* $= *0$ than those from *src_ip* $= *1$. We notice, however, this trend is stable thus predictable. That is, the *ratio* of number of requests matching $0*$ compared to $1*$ remains a *constant* over the time.

In this section, we discuss how to extend Niagara's single VIP algorithm to handle the constantly skewed request distribution. [5] Formally, let $req(p)$ be the proportion of requests received from suffix pattern p (e.g., $req(*0) = 0.6$). We should notice that $req(p)$ decreases exponentially in the order the length of p [6].

Consider an example function $req(p) = 0.6^a * 0.4^b$, where a and b are the number of 0s and 1s in p. We use the recurring example ($w_1 = \frac{1}{6}, w_2 = \frac{1}{3}, w_3 = \frac{1}{2}$) and $e = 0.02$ to illustrate how to generate the rules.

Similar to the standard algorithm, we pick w_3 as the pool. But for each non-pool weight, we generate rules *along with* approximating its values before proceeding to the next weight. In the example, we approximate $w_1 = \frac{1}{6}$ (Table 10) by finding two suffixes, whose $req(\cdot)$ values bound w_1 closely. With the given *req* function, the bounds for w_1 is in the form of $0.6^a * 0.4^b$. Therefore, the lower bound and upper bound for $\frac{1}{6}$ are 0.4^2 ($req(*11)$) and 0.6^3 ($req(*000)$) respectively, i.e.,

$$req(*11) \leq \frac{1}{6} \leq = req(*000)$$

As the lower bound already reaches tolerable error, we continue computing only the upper bound. We could either adding another suffix to the lower bound (e.g., $req(*11) + req(*111100)$) or subtracting a suffix from the upper bound (e.g., $req(*000) - req(*11000)$). The smaller one becomes the new upper bound. Therefore, the final approximation for w_1 is

$$req(*11) \leq w_1 \leq req(*11) + req(*111100)$$

Here, we pick the upper bound for w_1; the rules are ($*111100$, fwd to 1) and ($*11$, fwd to 1); the corresponding nodes in the suffix tree are colored accordingly.

Prior to approximating w_2, we update $req(\cdot)$ function to account for the suffixes already colored with w_1, since they cannot be colored to other weights any more. Specifically, for every positive term $req(p)$, we update p's ancestor suffix (say q) by doing $req(q) = req(q) - req(p)$. For $req(*11)$ in the example, we update $req(*1) = req(*1) - req(*11)$ and $req(*) = req(*) - req(*11)$. For every negative term $req(p)$, we add $req(p)$ back to p's ancestor suffix.

Then, we repeat the similar approximation for $w_2 = \frac{1}{3}$ to compute the lower and upper bound. We start with $req(*10) = 0.4 * 0.6$ and $req(*00) = 0.6^2 - 0.6^2 * 0.4^4$ ($req(*00)$ is updated because $*111100$ is colored with w_1) as lower and upper bounds. As the upper bound already reaches tolerable error, we tighten the lower bound only. We could either adding another suffix to the lower bound (e.g., $req(*10) + req(*0001)$) or subtracting a suffix from the upper bound (e.g., $req(*00) - req(*110100)$). The bigger one becomes the new lower bound. The final result is

$$req(*00) - req(*110100) \leq w_2 \leq req(*00)$$

We pick the lower bound; the rules are ($*110100$, fwd to pool) and ($*00$, fwd to 2). We then combine the rules for w_1 and w_2 to get the final rule list (Figure 11).

B. RULE MINIMIZATION WITH APPROXIMATION

Section 3.1.1 presents how to approximate an arbitrary weight with two bounds: lower-bound and upper-bound. Each bound consists of a series of positive and negative powers-of-two terms. To approximate multiple weights, we should pick one bound as the approximation of each non-pool weight and ensure the approximation error on the pool weight does not exceed tolerable error. The goal is to minimize the resulting number of rules. In this section, we fill in the details on the possible picking strategies to achieve the optimization goal.

There are two picking strategies: exhaustive search and greedy heuristics. An exhaustive search enumerates all combination of lower-bound and upper-bound approximations for non-pool weights. Among all combinations whose error for pool weight is within tolerable error, it picks the one with minimum number of rules. Therefore, the brute-force approach gives optimal solutions, but takes exponential order of time to complete.

In contrast, the greedy heuristics targets at polynomial time complexity with tradeoff in optimality. The algorithm is shown in Figure 16. The heuristics picks the bound for weights in multiple iterations. In each iteration, it chooses one bound for one non-pool weight to minimize the current number of rules without exceeding the tolerable error.

Let L_i and U_i be the lower-bound and upper-bound for weight w_i. Let a_j be the index of weight that is chosen by the j-th iteration. Then w_{a_j} and w'_{a_j} are the weight and approximation chosen by j-th iteration. At the beginning of j-th iteration, the heuristics first decides the current error of pool weight e^p, i.e.,

$$e^p = w'_{pool} - w_{pool} = -\sum_{i<j} (w'_{a_i} - w_{a_i})$$

Initially, $e^p = 0$. Based on the value of e^p, the heuristics decide what bound can be chosen without exceeding the tolerable error. Specifically, if $e^p \geq 0$, meaning pool is over-estimated, then it can pick upper-bounds; if $e^p \leq 0$, meaning pool is under-estimated, then it can pick lower-bounds. Then, it tries on every weight, which are not approximated in previous iterations, with the allowed bounds. Finally, It picks the weight and bound that minimize the current number of rules. The iterations are repeated until all non-pool weights are approximated.

Iteration	L	U	$\mathbf{w_{v1} - L}$	$\mathbf{U - w_{v1}}$
0	0	1	0.1667	0.8333
1	0.4^2	0.6^2	0.0077	0.0493
2	–	$0.4^2 + 0.6^2 * 0.4^4$	–	0.00205

Table 10: Steps to compute the upper and lower bounds to approximate $w_{v1} = \frac{1}{6}$ with $e = 0.02$. Note that the second iteration U is obtained by adding $0.6^2 * 0.4^4$ to first iteration L.

Pattern	Action
$*111100$	fwd to 1
$*110100$	fwd to 3
$*11$	fwd to 1
$*00$	fwd to 2
$*$	fwd to 3

Table 11: Final rules for $\frac{1}{6}, \frac{1}{3}$ and $\frac{1}{2}$ with unbalanced request distribution.

C. MINIMIZE OVERALL STRETCH.

In the paper, we focus on the computation of hardware rules to minimize imbalance – the fraction of traffic that should be deflected. Besides the volume of traffic to deflect, we are also interested in the total stretch the misdirected traffic experiences. For example, there are four SW switches – A, B, C and D, where AB, BC and CD are neighbors. A and D are most distant. Based on the hardware rules, A and C are overloaded with 1% requests each, while B and D are underloaded and each can take another 1% requests. The solution with optimal stretch is to have A forward excessive requests to B and C forward to D. However, a solution that does not consider the stretch may ask A to forward request to D and B to C. The overall stretch is determined by the rules installed on SW switches. In this section, we

[5] We let the infrequent change of request distribution trigger rule update, thus keeping the loads on backends accurate.

[6] For any $req(q) \leq b^{len(q)}$, where $b = \max\{\frac{req(p0)}{req(p)}, \frac{req(p1)}{req(p)} \forall p\}$

```
PICK-BOUNDS(bounds, pool)
e^p ← 0
approx ← []
while bounds is not empty
    pick_upper ← e^p ≥ 0
    pick_lower ← e^p ≤ 0
    for (L_i, U_i, w_i) ∈ bounds
        if pick_lower and min_r > COMPUTE-RULE(approx + [L_i])
            min_r ← COMPUTE-RULE(approx + [L_i])
            a, ← i, L_i
        if pick_upper and min_r > COMPUTE-RULE(approx + [U_i])
            min_r ← COMPUTE-RULE(approx + [U_i])
            a, w' ← i, U_i
    approx ← approx + [w']
    e^p ← e^p − w' + w_a
    Remove (L_a, U_a, w_a) from bounds
return approx
```

Figure 16: Greedy heuristics for picking approximation bounds.

briefly discuss how to compute rules for SW switches so as to minimize the stretch.

We take two steps to compute the software rules. The first step is to decide how much fraction of traffic each overloaded SW switch should forward to other software switch, so as to minimize stretch. This problem can be directly reduced to a *Min-Cost Max-Flow* problem. Specifically, in the Max-Flow graph, the overloaded SW switches serve as suppliers and the underloaded SW switches are the consumers. The amount of supply and consumption is the imbalance of the corresponding SW switches. Then, we create weighted edges between suppliers and consumers, where the weight of edge is the distance between SW switches. The Min-Cost Max-Flow solution gives the supply between SW switches, i.e., the transferred traffic.

Once we have the amount of traffic to transfer between SW switches, the second step is to generate rules. As the software rule-table is presumably infinite, we do not need consider the number of software rules. Hence, we generate rules for each pair of SW switches with non-zero traffic transfer. For every pair, we use the algorithm in Section 4.2.1 by regarding the existing rules as an initial approximation and adding extra rules to deflect the traffic.