WCMP: Weighted Cost Multipathing for Improved Fairness in Data Centers

Junlan Zhou*, Malveeka Tewari†*, Min Zhu*, Abdul Kabbani*,
Leon Poutievski*, Arjun Singh*, Amin Vahdat†*

*Google Inc. † UC San Diego

{zjl, malveeka, mzhu, akabbani, leonp, arjun, vahdat}@google.com

Abstract

Data Center topologies employ multiple paths among servers to deliver scalable, cost-effective network capacity. The simplest and the most widely deployed approach for load balancing among these paths, Equal Cost Multipath (ECMP), hashes flows among the shortest paths toward a destination. ECMP leverages uniform hashing of balanced flow sizes to achieve fairness and good load balancing in data centers. However, we show that ECMP further assumes a balanced, regular, and fault-free topology, which are invalid assumptions in practice that can lead to substantial performance degradation and, worse, variation in flow bandwidths even for same size flows. We present a set of simple algorithms employing *Weighted Cost Multipath (WCMP)* to balance traffic based on the changing network topology. The state required for WCMP is already disseminated as part of standard routing protocols and it can be implemented in current switch silicon without any hardware modifications. We show how to deploy WCMP in a production OpenFlow network environment and present experimental and simulation results to show that variation in flow bandwidths can be reduced by as much as $25\times$ by employing WCMP relative to ECMP.

1. Introduction

There has been significant recent interest in a range of network topologies for large-scale data centers [2, 15, 20]. These topologies promise scalable, cost-effective bandwidth to emerging clusters with tens of thousands of servers by leveraging parallel paths organized across multiple switch stages. Equal Cost Multipath (ECMP) [14, 17] extension to OSPF is the most popular technique for spreading traffic among available paths. Recent efforts [3, 16] propose dynamically monitoring global fabric conditions to adjust forwarding rules, potentially on a per-flow basis. While these efforts promise improved bandwidth efficiency, their additional complexity means that most commercial deployments still use ECMP forwarding given the simplicity of using strictly local switch state for packet forwarding.

To date, however, the current routing and forwarding protocols employing ECMP-style forwarding focus on regular, symmetric, and fault-free instances of tree-based topologies. At a high-level, these protocols assume that there will be: (i) large number of equal cost paths to a given destination, and (ii) equal amount of bandwidth capacity to the destination downstream among all equal cost paths. The first assumption is violated for non-tree topologies such as HyperCubes and its descendants [1, 15] that require load balancing across non-equal cost paths. The second assumption is violated when a failure downstream reduces capacity through a particular next hop or, more simply, when a tree-based topology inherently demonstrates *imbalanced striping* (defined in Section 4) when the number of switches at a particular stage in the topology is not perfectly divisible by the number of uplinks on a switch at the previous stage.

Thus, for realistic network deployments, the bandwidth capacity available among "equal cost" next hops is typically guaranteed to be unequal even in the baseline case where all flows are of identical length. That is to say, ECMP forwarding leads to imbalanced bandwidth distribution simply from static topology imbalance rather than from dynamic communication patterns. It is our position that routing protocols should be able to correct for such imbalance as they typically already collect all the state necessary to do so. Based on our experience with large-scale data center deployments, we have found that substantial variation in per-flow bandwidth for flows that otherwise are not subject to prevailing congestion both reduces application performance and makes the network more difficult to diagnose and maintain.

This paper presents Weighted Cost Multipathing (WCMP) to deliver fair per-flow bandwidth allocation based on the routing protocol's view of the topology. We present the design and implementation of WCMP in a Software Defined Network running commodity switches, which distribute traffic among available next hops in proportion to the available link capacity (not bandwidth) to the destination according to the dynamically changing network topology. We further present topology connectivity guidelines for improving network throughput. Inserting the necessary weighted forwarding entries can consume hundreds of entries for a single destination, aggressively consuming limited on-chip SRAM/TCAM table entries. Hence, we present algorithms that trade a small bandwidth oversubscription for substantial reduction in consumed forwarding table entries. Our results indicate that even a 5% increase in oversubscription can significantly reduce the number of forwarding table entries required. Our algorithms for weight reduction make WCMP readily deployable in current commodity switches with limited forwarding table entries.

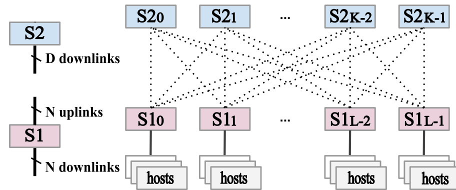


Figure 1: 2-stage Clos network

We present our evaluation of WCMP in an OpenFlow controlled 10Gb/s data center network testbed for different kinds of traffic patterns including real data center traffic traces from [7]. The WCMP weights are meant to deal with long lived failures (switch/link failures, link flappings etc.) included in the routing updates, and our ability to scale and react is only limited by the reactivity of the routing protocol. Our results show that WCMP reduces the variation in flow bandwidths by as much as 25× compared to ECMP. WCMP is complementary to traffic load balancing schemes at higher layers such as Hedera [3] and MPTCP [28].

2. Background & Motivation

The primary objective of this paper is to propose a deployable solution that addresses the ECMP weakness in handling topology asymmetry, thus improving fairness across flow bandwidths and load balancing in data center networks. While our results apply equally well to a range of topologies, including *direct connect* topologies [1, 15, 30], for concreteness we focus on multi-stage, tree-based topologies that form the basis for current data center deployments [2, 14].

We abstract the data center network fabric as a two-stage Clos topology, as depicted in Figure 1. Here, each switch in Stage 1 (S1) stripes its uplinks across all available Stage 2 (S2) switches. Hosts connected to the S1 switches can then communicate by leveraging the multiple paths through the different S2 switches. Typically, ECMP extensions to routing protocols such as OSPF [25] distribute traffic equally by hashing individual flows among the set of available paths. We show how ECMP alone is insufficient to efficiently leverage the multipaths in data center networks, particularly in presence of failures and asymmetry. While we focus on a logical two stage topology for our analysis, our results are recursively applicable to a topology of arbitrary depth. For instance, high-radix S1 switches may internally be made up of a 2-stage topology using lower-degree physical switches; our proposed routing and load balancing techniques would apply equally well at multiple levels of the topology.

2.1 Motivating Example

In a multi-stage network, *striping* refers to the distribution of uplinks from lower stage switches to the switches in the successive stage. We define striping in more detail in Section 4 but want to point out that a uniformly symmetric or balanced striping requires the number of uplinks at a lower stage switch be equal to (or an integer multiple of) the number of switches in the next stage, to ensure that the uplinks at a lower stage switch can be striped uniformly across the switches in the next stage. In large scale data centers, it is hard to maintain such balanced stripings due to heterogeneity in switch port counts and frequent switch/link failures. We discuss the impact of striping asymmetry in detail in later sections. Here, we motivate how employing ECMP over topologies with uneven striping results in poor load balancing and unfair bandwidth distribution.

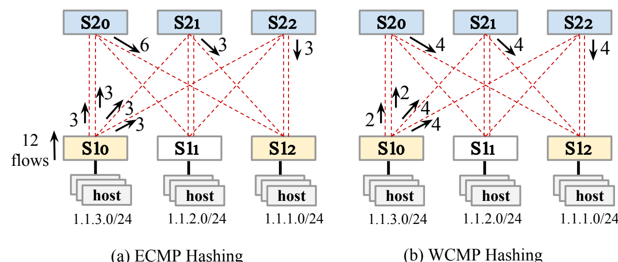


Figure 2: Multipath flow hashing in an asymmetric topology

Consider the 2-stage Clos topology depicted in Figure 2(a). Each link in the topology has the same capacity, 10 Gb/s. Since each S1 switch has four uplinks to connect to the three S2 switches, it results in asymmetric distribution of uplinks or *imbalanced striping* of uplinks across the S2 switches. Assuming ideal ECMP hashing of 12 flows from the source switch $S1_0$ to the destination switch $S1_2$, three flows each are hashed onto each of the four uplinks at $S1_0$. However, the six flows reaching $S2_0$ now have to contend for capacity on the single downlink to $S1_2$. As such, the six flows via $S2_0$ receive *one-sixth* of the link capacity, 1.66 Gb/s each, while the remaining six flows receive *one-third* of the link capacity, 3.33 Gb/s each, resulting in unfair bandwidth distribution even for identical flows.

Note that the effective bandwidth capacity of the two uplinks to $S2_0$ at $S1_0$ is only 10 Gb/s, bottlenecked by the single downlink from $S2_0$ to $S1_2$. Taking this into consideration, if $S1_0$ weighs its uplinks in the ratio 1:1:2:2 for hashing (as opposed to 1:1:1:1 with ECMP) all flows would reach the destination switch with the same throughput, one-fourth of link capacity or 2.5 Gb/s in this example, as shown in Figure 2(b). This observation for weighing different paths according to their effective capacity is the premise for *Weighted Cost Multipathing (WCMP)*.

2.2 Bandwidth Fairness

Data center applications such as Web search [5], MapReduce [12], and Memcached [24] operate in a bulk synchronous manner where initiators attempt to exchange data with a large number of remote hosts. The operation can only proceed once all of the associated flows complete, with the application running at the speed of the slowest transfer. Similarly, when there are multiple "bins" of possible performance between server pairs, performance debugging

becomes even more challenging; for instance, distinguishing between network congestion or incast communication and poor performance resulting from unlucky hashing becomes problematic, motivating fair bandwidth distribution. Orchestra [8], a system for managing data transfers in large clusters, demonstrated how fair flow scheduling at the application layer resulted in a 20% speedup of the MapReduce shuffle phase, motivating the need for fairness in bandwidth distributions.

2.3 Reasons for Topology Asymmetry

While it is desirable to build regular, symmetrical topologies in the data centers, it is impractical to do so for the following reasons.

(1) **Heterogeneous Network Components:** Imbalanced striping is inherent to any topology where the number of uplinks on S1 switches is not an integer multiple of the total number of S2 switches. Building the fabric with heterogeneous switches with different port counts and numbers inherently creates imbalance in the topology. Depending on the required number of server ports, such imbalance in striping is typically guaranteed in practice without substantial network overbuild.

(2) **Network Failures:** Even in the case where the baseline topology is balanced, common case network failures will result in imbalanced striping. Consider a data center network consisting of 32 S2 switches with 96-ports each, 48 S1 switches with 128-ports each and 3072 hosts. Each S1 switch uses 64 ports to connect to the S2 switches, with two uplinks to each S2 switch. For this small data center, an average monthly link availability of 99.945 will result in more than 300 link failures each month, creating asymmetric links between S1 and S2 switches and making the topology imbalanced. If any one of the two downlinks from an S2 switch to a destination S1 switch fails, the flows hashed to that particular S2 switch will suffer a 50% reduction in bandwidth, since the destination S1 switch is now reachable by only one downlink. Since ECMP fails to account for capacity reductions due to failures while hashing flows, it leads to unfair bandwidth distribution. This is particularly harmful for the performance of many data center applications with scatter and gather traffic patterns that are bottlenecked by the slowest flow. Tables 1 and 2 show availability measurements of production data center switches and links (collected by polling SNMP counters) in support of our claim.

	Oct	Nov	Dec	Jan
Switch	99.989	99.984	99.993	99.987
Link	99.929	99.932	99.968	99.955

Table 1: Availability of Switches and Links by Month

2.4 Related Work

Load balancing and traffic engineering have been studied extensively for WAN and Clos networks. Many traffic engineering efforts focus on the wide-area Internet [4, 21, 31, 32] across two main categories: (i) traffic engineering based on measured traffic demands and (ii) oblivious routing based on the "hose" model [21]. Our work differs from the previous work as it addresses unfairness to topology asymmetry and proposes a readily deployable solution. Further more, efficient WAN load balancing relies on traffic matrices which change at the scale of several minutes or hours. However, for data centers highly dynamic traffic matrices, changing at the scale of few seconds motivate the need for load balancing solutions with low reaction times.

	01/29	02/05	02/12	02/19
Switch	99.996	99.997	99.984	99.993
Link	99.977	99.976	99.978	99.968

Table 2: Availability of Switches and Links by Week

While ours is not the first work that advocates weighted traffic distribution, we present a practical, deployable solution for implementing weighted traffic distribution in current hardware. There has been prior work that proposed setting up parallel MPLS-TE tunnels and splitting traffic across them unequally based on tunnel bandwidths [26]. The load balancing ratio between the tunnels is approximated by the number of entries for each tunnel in the hash table. However, the proposal does not address the concern that replicating entries can exhaust a large number of forwarding entries. In this work, we present algorithms that address this challenge explicitly.

Villamizar [29] proposed unequal splitting of traffic across the available paths by associating a hash boundary with each path. An incoming packet is sent out on the path whose hash boundary is less than the hash of the packet header. These hash boundaries serve as weights for each path and are adjusted based on the current demand and updated repeatedly. Implementing hash boundaries requires switch hardware modification, raising the bar for immediate deployment. Our contributions include proposing a complete, deployable solution that improves load balancing in data centers without hardware modification.

In the data center context, there have been many proposals which advocate multipath based topologies [2, 14, 15, 23, 30] to provide higher bandwidth and fault tolerance. F10, a proposal for building fault resilient data center fabric [23] advocates the use of weighted ECMP for efficient load balancing in presence of failures. Such network fabrics can leverage WCMP as a readily deployable solution for improving fairness. These topologies with high degree of multipaths have triggered other efforts [3, 10, 11] that evenly spread flows over the data center fabric for higher utilization. Hedera focuses on dynamically allocating elephant flows to paths via a centralized scheduler by repeatedly polling the switches for network utilization and/or flow sizes. Mice flows, however, would still be routed in a static manner using ECMP. Hedera's centralized measurement and software control loop limit its responsiveness to mice flows.Hence,

our WCMP algorithms provide baseline short-flow benefits for systems like Hedera, and scale better for large networks with balanced flow sizes (for applications e.g TritonSort/MapReduce).

MPTCP [28], a transport layer solution, proposes creating several sub-flows for each TCP flow. It relies on ECMP to hash the sub-flows on to different paths and thus provide better load balancing. In comparison, WCMP hashes flows across paths according to available capacity based on topology rather than available bandwidth based on communication patterns. We find that for uniform communication patterns, WCMP outperforms MPTCP whereas MPTCP outperforms WCMP in the presence of localized congestion. The two together perform better than either in isolation as they work in a complementary fashion: WCMP makes available additional capacity based on the topology while MPTCP better utilizes the capacity based on dynamic communication patterns as we show in our evaluation section.

3. Weighted Cost Multi Pathing

The need for WCMP is motivated by the asymmetric striping in data center topologies. The right striping is indeed crucial in building a data center that ensures fairness and efficient load balancing. We present our proposals for striping alternatives in Section 4. In this section, we discuss the current practices, challenges and our approach in designing a deployable solution for weighted traffic hashing.

3.1 Multipath Forwarding in Switches

Switches implement ECMP based multipath forwarding by creating multipath groups which represent an array of "equal cost" egress ports for a destination prefix. Each egress port corresponds to one of the multiple paths available to reach the destination. The switch hashes arriving packet headers to determine the egress port for each packet in a multipath group. Hashing on specific fields in the packet header ensures that all packets in the same flow follow the same network path, avoiding packet re-ordering.

To implement weighted hashing, we assign weights to each egress port in a multipath group. We refer to the array of egress ports with weights as the *WCMP group*. Each WCMP group distributes flows among a set of egress ports in proportion to the weights of each port. The weight assigned to an egress port is in turn proportional to the capacity of the path(s) associated with that egress port. Currently, many commodity switches offer an OpenFlow compatible interface with their software development kit (SDK) [6, 18]. This allows us to realize weight assignment by replicating a port entry in the multipath table in proportion to its weight.

Figure 3 shows the packet processing pipeline for multipath forwarding in a commodity switch. The switch's multipath table contains two groups. The first four entries in the table store an ECMP group for traffic destined to prefix 1.1.2.0/24. The next 12 entries in the table store a WCMP group for weighted distribution of traffic destined to prefix 1.1.1.0/24. Traffic ingressing the switch is first matched against the Longest Prefix Match (LPM) entries. Upon finding a match, the switch consults the multipath group entry to determine the egress port. For example, a packet with destination 1.1.1.1 matches the LPM table entry pointing to the WCMP group with base index of 4 in the multipath table. The switch determines the offset into the multipath table for a particular packet by hashing over header fields e.g., IP addresses, UDP/TCP ports, as inputs. The hash modulo the number of entries for the group added to the group's base index determines the table entry with the egress port for the incoming packet ((15 mod 12) + 4 = 7).

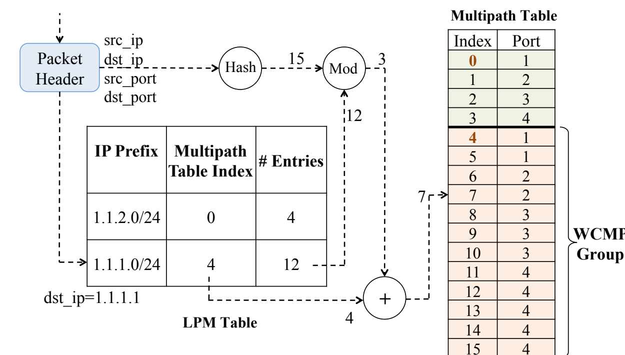


Figure 3: Flow hashing in hardware among the set of available egress ports

Replicating entries for assigning weights can easily exceed the number of table entries in commodity silicon, typically numbering in the small thousands. To overcome this hardware limitation on table entries, we map the "ideal" WCMP port weights onto a smaller set of integer weights, with the optimization goal of balancing consumed multipath table resources against the impact on flow fairness. In our switch pipeline example, the egress port numbers 1, 2, 3, 4 in the WCMP group have weights 2, 2, 3, 5 respectively (weight ratio 1:1:1.5:2.5) and use 12 entries in the multipath table to provide ideal fairness. If we change these weights to 1, 1, 2, 3 respectively, we reduce the number of table entries required from 12 to 7 with small changes to the relative ratios between the weights. This reduction is extremely useful in implementing weighted hashing as this helps in significantly lowering down the requirement for TCAM entries. This is important because hardware TCAM entries are used for managing access control rules, maintaining flow counters for different kinds of traffic and are always a scarce resource in commodity switches. In the next section, we present algorithms that aim at reducing the WCMP weights with limited impact on fairness.

3.2 Weight Reduction Algorithms

We begin by formalizing how WCMP egress port weights affect network performance. Consider a WCMP group G

with P member egress ports.We denote the weights for each member port in G by the tuple $(X_1, X_2, ..X_P)$, and use the following definitions:

λ_G: maximum traffic demand served by WCMP group G

$B_i(G)$: maximum traffic demand served by the ith member port in G

$G[i].weight$: weight of ith member port in G (also denoted by X_i)

$G.size$: number of table entries used by G (sum of all X_i)

$$B_i(G) = \lambda_G \cdot \frac{G[i].weight}{\sum_{k=1}^{P} G[k].weight} = \lambda_G \cdot \frac{X_i}{\sum_{k=1}^{P} X_k} \quad (1)$$

Let $G\prime$ denote a new WCMP group reduced from G. It has the same set of member egress ports as G but with smaller weights denoted by $(Y_1, Y_2, ..Y_P)$. Given a traffic demand of λ_G, the fraction of traffic demand to the ith member port, we compute $B_i(G\prime)$ by replacing G with $G\prime$ in eq. 1. When $B_i(G\prime) > B_i(G)$, the ith port of G receives more traffic than it can serve, becoming oversubscribed. We seek to reduce the weights in G to obtain $G\prime$ while observing a maximum oversubscription of its member ports, denoted as $\Delta(G, G\prime)$:

$$\begin{aligned}\Delta(G, G\prime) &= max_i(B_i(G\prime)/B_i(G)) \\ &= max_i(\frac{G\prime[i].weight \cdot \sum_{k=1}^{P} G[k].weight}{G[i].weight \cdot \sum_{k=1}^{P} G\prime[k].weight}) \\ &= max_i(\frac{Y_i \cdot \sum_{k=1}^{P} X_k}{X_i \cdot \sum_{k=1}^{P} Y_k})\end{aligned} \quad (2)$$

While reducing the weights for a WCMP group, we can optimize for one of two different objectives: (i) maximum possible reduction in the group size, given a maximum oversubscription limit as the constraint, or (ii) minimizing the maximum oversubscription with a constraint on the total size of the group.

3.2.1 Weight Reduction with an Oversubscription Limit

Given a WCMP group G with P member ports and a maximum oversubscription limit, denoted by parameter θ_{max}, we want to find a WCMP group $G\prime$ with P member ports where the member weights $(Y_1, Y_2, ..Y_P)$ for $G\prime$ are obtained by solving the following optimization problem.

$$\begin{aligned} \text{minimize} \quad & \sum_{i=1}^{P} Y_i \\ \text{subject to} \quad & \Delta(G, G\prime) \le \theta_{max} \\ & X_i, Y_i \text{ are +ve integers } (1 \le i \le P) \end{aligned} \quad (3)$$

Algorithm 1 *ReduceWcmpGroup(G, θ_{max}).* Returns a smaller group $G\prime$ such that $\Delta(G, G\prime) \le \theta_{max}$

```
1: for i = 1 to P do
2:     G′[i].weight = 1
3: end for
4: while Δ(G, G′) > θ_max do
5:     index = ChoosePortToUpdate(G, G′)
6:     G′[index].weight = G′[index].weight + 1
7:     if G′.size ≥ G.size then
8:         return G
9:     end if
10: end while
11: return G′
```

Algorithm 2 *ChoosePortToUpdate(G, G$\prime$).* Returns the index of member port whose weight should be incremented to result in least maximum oversubscription.

```
1: min_oversub = INF
2: index = −1
3: for i = 1 to P do
4:     oversub = ((G′[i].weight+1)·G.size) / ((G′.size+1)·G[i].weight)
5:     if min_oversub > oversub then
6:         min_oversub = oversub
7:         index = i
8:     end if
9: end for
10: return index
```

This is an *Integer Linear Programming (ILP)* problem in variables Y_is, known to be NP-complete [9]. Though the optimal solution can be obtained by using well known linear programming solutions, finding that optimal solution can be time-intensive, particularly for large topologies. Hence, we propose an efficient greedy algorithm that gives an approximate solution to the optimization problem in Algorithm 1.

We start by initializing each Y_i in $G\prime$ to 1. $G\prime$ is the smallest possible WCMP group with the same member ports as G (Lines 1-3). We then increase the weight of one of the member ports by 1 by invoking Algorithm 2 and repeat this process until either: (i) we find weights with maximum oversubscription less than θ_{max} or, (ii) the size of $G\prime$ is equal to the size of the original group G. In the latter case, we return the original group G, indicating the algorithm failed to find a $G\prime$ with the specified θ_{max}.

Starting from the WCMP group with the smallest size, this algorithm greedily searches for the next smaller WCMP group with the least oversubscription ratio. For a WCMP group G with P ports and original size W, runtime complexity of the greedy algorithm is $O(P \cdot (W - P))$. To compare the results of the greedy algorithm with the optimal solution , we ran Algorithm 1 on more than 30,000 randomly generated WCMP groups and compared the results with the optimal solution obtained by running the GLPK LP solver [13]. The greedy solution was sub-optimal for only 34 cases (0.1%) and within 1% of optimal in all cases.

3.2.2 Weight Reduction with a Group Size Limit

Forwarding table entries are a scarce resource and we want to create a WCMP group that meets the constraint on the table size. Formally, given a WCMP group G with P member ports and multipath group size T, the weights for WCMP group $G\prime$ can be obtained by solving the following optimization problem:

$$\begin{aligned} \text{minimize} \quad & \Delta(G, G\prime) \\ \text{subject to} \quad & \sum_{i=1}^{P} Y_i \leq T \\ & X_i, Y_i \text{ are +ve integers } (1 \leq i \leq P) \end{aligned} \tag{4}$$

Programming forwarding table entries is a bottleneck for route updates [11]. Reducing the number of forwarding table entries has the added benefit of improving routing convergence as it reduces the number of entries to be programmed. We refer to the above optimization problem as *Table Fitting*. This is a *Non-linear Integer Optimization* problem because the objective function $\Delta(G, G\prime)$ is a non-linear function of the variables Y_is, which makes it harder to find an optimal solution to this problem.We present a greedy algorithm for the Table Fitting optimization in Algorithm 3.

We begin by initializing $G\prime$ to G. Since the final weights in $G\prime$ must be positive integers, ports with unit weight are counted towards the $non_reducible_size$ as their weight cannot be reduced further. If fractional weights were allowed, reducing weights in the same ratio as that between the size limit T and original size will not result in any oversubscription. However, since weights can only be positive integers, we round the weights as shown in Lines 14-16 of Algorithm 3. It is possible that after rounding, the size of the group exceeds T, since some weights by may be rounded up to 1. Hence, we repeatedly reduce weights until the size of the group is less than T. We do not allow zero weights for egress ports because that may lower the maximum throughput capacity of the original WCMP group. In some cases it is possible that the size of the reduced group is strictly less than T, because of rounding down (Line 14). In that case, we increase the weights of the WCMP group up to the limit T, in the same manner as in Algorithm 1 to find the set of weights that offer the minimum $\Delta(G, G\prime)$ (Lines 20-30).

For a WCMP group with size W and P member ports, the runtime complexity of Algorithm 3 is $O(P \cdot (W - T) + P^2)$. If high total bandwidth is not a requirement and zero weights are allowed, we can further optimize this algorithm by rounding fractional weights to zero and eliminate the outer while loop, with final runtime complexity as $O(P^2)$.

3.2.3 Weight Reduction for Multiple WCMP Groups

For reducing weights given a maximum oversubscription limit for a group for WCMP groups H, we can simply run Algorithm 1 independently for each WCMP group in H.

Algorithm 3 *TableFitting(G, T)*. WCMP Weight Reduction for Table Fitting a single WCMP group G into size T.

1: $G\prime = G$
2: **while** $G\prime.size > T$ **do**
3: $\quad non_reducible_size = 0$
4: $\quad$ **for** $i = 1$ to P **do**
5: $\quad\quad$ **if** $G\prime[i].weight = 1$ **then**
6: $\quad\quad\quad non_reducible_size += G\prime[i].weight$
7: $\quad\quad$ **end if**
8: $\quad$ **end for**
9: $\quad$ **if** $non_reducible_size = P$ **then**
10: $\quad\quad break$
11: $\quad$ **end if**
12: $\quad reduction_ratio = \frac{(T - non_reducible_size)}{G.size}$
13: $\quad$ **for** $i = 1$ to P **do**
14: $\quad\quad G\prime[i].weight = \lfloor (G[i].weight \cdot reduction_ratio) \rfloor$
15: $\quad\quad$ **if** $G\prime[i].weight = 0$ **then**
16: $\quad\quad\quad G\prime[i].weight = 1$
17: $\quad\quad$ **end if**
18: $\quad$ **end for**
19: **end while**
20: $remaining_size = T - G\prime.size$
21: $min_oversub = \Delta(G, G\prime)$
22: $G\prime\prime = G\prime$
23: **for** $k = 1$ to $remaining_size$ **do**
24: $\quad index = ChoosePortToUpdate(G, G\prime)$
25: $\quad G\prime[index].weight = G\prime[index].weight + 1$
26: $\quad$ **if** $min_oversub > \Delta(G, G\prime)$ **then**
27: $\quad\quad G\prime\prime = G\prime$
28: $\quad\quad min_oversub = \Delta(G, G\prime)$
29: $\quad$ **end if**
30: **end for**
31: **return** $G\prime\prime$

However, for reducing weights given a limit on the total table size for H, the exact size limit for individual WCMP groups is not known. One alternative is to set the size limit for each group in H in proportion to the ratio between the total size limit for H and the original size of H and then run Algorithm 3 independently on each group in H. While this approach ensures that groups in H fit the table size S, it does not strive to find the minimum $\Delta(G, G\prime)$ of member groups that could be achieved by changing the individual size constraints on each WCMP group in H. For that, we present Algorithm 4 that achieves weight reduction for a set of WCMP groups by linearly searching for the lowest maximum oversubscription limit.

Algorithm 4 shows the pseudo code for weight reduction across groups in H. We begin with a small threshold for the maximum oversubscription θ_c, and use it with Algorithm 1 to find smaller WCMP groups for each of the member groups in H. We sort the groups in H in descending order of their size and begin by reducing weights for the largest group in H. We repeatedly increase the oversubscription threshold (by a step size ν) for further reduction

of WCMP groups until their aggregate size in the multipath table drops below S. Since the algorithm progresses in step size of ν, there is a trade-off between the accuracy of the oversubscription limit and the efficiency of the algorithm, which can be adjusted as desired by changing the value of ν.

Algorithm 4 *TableFitting(H, S).* WCMP Weight Reduction for Table Fitting a set of WCMP groups H into size S.

```
1:  θ_c = 1.002 // θ_c: enforced oversubscription
2:  ν = 0.001 // ν: step size for increasing the θ_c
3:  // Sort the groups in H in descending order of size.
4:  H′ = H
5:  while TotalSize({H′}) > S do
6:      for i = 1 ... NumGroups({H}) do
7:          H′[i] = ReduceWcmpGroup(H[i], θ_c)
8:          if TotalSize({H′}) ≤ S then
9:              return
10:         end if
11:     end for
12:     θ_c += ν
13: end while
14: return H′
```

4. Striping

Striping refers to the distribution of uplinks from lower stage switches to the switches in the successive stage in a multi-stage topology. The striping is crucial in determining the effective capacity for different paths and hence the relative weights for the different ports in a WCMP group. While weight reduction algorithms reduce weights for a WCMP group, the striping determines the original weights in a WCMP group. Balanced stripings are desirable in data center networks so as to deliver uniform and maximum possible throughput among all communicating switch pairs. However, maintaining balanced striping is impractical due to its strict requirement that the number of upper links in a switch must be an integer multiple of the number of upper layer switches. It is practically unsustainable due to frequent node and link failures. In this section, we attempt to identify the design goals for topology striping and propose striping alternatives that illustrate trade-offs between different goals.

4.1 Striping Alternatives

We use the 2-stage network topology depicted in Figure 1 in describing striping alternatives. The class of 2-stage Clos networks is characterized by the following parameters, assuming all physical links have unit capacity:

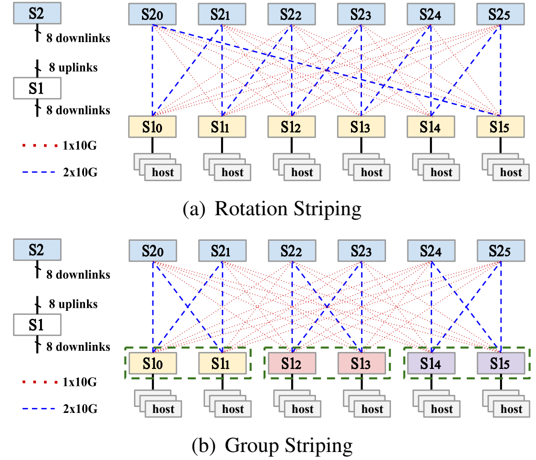
K: Number of stage-2 (S2) switches.

D: Number of downlinks per S2 switch.

L: Number of stage-1 (S1) switches.

N: Number of uplinks per S1 switch.

The striping is imbalanced when N cannot be divided by K. In particular, one S1 switch may be connected to an S2 switch with more uplinks than another S1 switch.While a naïve approach to build fully balanced topologies is to remove the additional uplinks at a switch, this will lower the maximum throughput between pairs of switches that can be connected symmetrically to the upper stage switches. For example in Figure 4(b), even though $S1_0$ and $S1_2$ are asymmetrically connected to the S2 switches, switches $S1_0$ and $S1_1$ are connected symmetrically and can achieve higher throughput as compared to the case where the extra links were removed to build a fully symmetrical topology.



(a) Rotation Striping

(b) Group Striping

Figure 4: Striping Alternatives

In order to be as close to balanced striping, we distribute the set of N uplinks from an S1 switch among S2 switches as evenly as possible. We first assign $\lfloor \frac{N}{K} \rfloor$ uplinks to each of the K S2 switches and the remaining N - K.($\lfloor \frac{N}{K} \rfloor$) uplinks at the S1 switch are distributed across a set of N - K.($\lfloor \frac{N}{K} \rfloor$) S2 switches. This strategy ensures that each S1 switch is connected to an S2 switch with at most 1 extra uplink as compared to other S1 switches connected to the same S2 switch. Formally we can represent this striping strategy using a connectivity matrix R, where each element R_{jk} is the number of uplinks connecting $S1_j$ to $S2_k$. Let p = $\lfloor \frac{N}{K} \rfloor$, then

$$R_{jk} = p \text{ or } p + 1, \quad (5)$$

The maximum achievable throughput may vary among different pairs of S1 switches based on the striping. The highest throughput between a pair of switches is achieved when each S1 switch has an equal number of uplinks to all S2 switches. The maximum achievable throughput between a pair of switches is lower when the pair have asymmetric striping to the S2 switches. Based on this observation, we derived two alternative striping options: (i) *Rotation Striping* and (ii) *Group Striping*, which illustrate the trade-off between improving the mean throughput versus improving the maximum achievable throughput across the different S1 switch pairs.

Figure 4(a) depicts a 2-stage Clos topology with 6 S1 switches and 6 S2 switches using *rotation striping*. For any pair of S1 switches, there is at least one oversubscribed S2 switch connected asymmetrically to the two S1 switches. Thus the maximum possible throughput for traffic between these switches is less than their total uplink bandwidth, even without competing traffic. The rotation striping can be generated by connecting a switch $S1_i$ with a contiguous set of π_p S2 switches with p uplinks each starting from $S2_{i(mod)K}$, and the remaining π_{p+1} S2 switches with $p+1$ uplinks.The derivations of π_p and π_{p+1} are shown below.

Algorithm 5 Generating group striping - Phase I

1: $\Omega_{p+1} = D - L \cdot p$
2: $\Omega_p = L - \Omega_{p+1}$
3: $\Omega = min(\Omega_{p+1}, \Omega_p)$
4: $\pi = min(\pi_p, \pi_{p+1})$
5: **if** $\Omega = 0$ **then**
6: $\quad Q = 1$
7: **else**
8: $\quad Q = \lfloor \frac{L}{\Omega} \rfloor - \lceil \frac{L\%\Omega}{\Omega} \rceil$
9: **end if**
10: **if** $\Omega = \Omega_{p+1}$ **then**
11: $\quad R_{lk} = p$ for $\forall\, l < L, k < K$
12: **else**
13: $\quad R_{lk} = p + 1$ for $\forall\, l < L, k < K$
14: **end if**
15: **for** $i = 0$ to $Q - 1$ **do**
16: $\quad$ **for** $l = i \cdot \Omega$ to $i \cdot \Omega + \Omega - 1$ **do**
17: $\quad\quad$ **for** $k = i \cdot \pi$ to $i \cdot \pi + \pi - 1$ **do**
18: $\quad\quad\quad$ **if** $\Omega = \Omega_{p+1}$ **then**
19: $\quad\quad\quad\quad R_{lk} = p + 1$
20: $\quad\quad\quad$ **else**
21: $\quad\quad\quad\quad R_{lk} = p$
22: $\quad\quad\quad$ **end if**
23: $\quad\quad$ **end for**
24: $\quad$ **end for**
25: **end for**

Algorithm 6 Generating group striping - Phase II

1: $shift = 0$
2: **for** $l = Q \cdot \Omega$ to $L - 1$ **do**
3: $\quad$ **for** $offset = 0$ to $\pi - 1$ **do**
4: $\quad\quad k = \pi \cdot Q + ((offset + shift)\%(K - Q \cdot \pi));$
5: $\quad\quad$ **if** $\Omega = \Omega_{p+1}$ **then**
6: $\quad\quad\quad R_{lk} = p + 1$
7: $\quad\quad$ **else**
8: $\quad\quad\quad R_{lk} = p$
9: $\quad\quad$ **end if**
10: $\quad$ **end for**
11: $\quad shift += N/D;$
12: **end for**

π_p: the number of S2 switches connected to an S1 switch with p downlinks.

π_{p+1}: the number of S2 switches connected to an S1 switch with $p+1$ downlinks.

$$\begin{aligned} \pi_{p+1} &= N - K \cdot p \\ \pi_p &= K - \pi_{p+1} = K \cdot (p+1) - N \end{aligned} \quad (6)$$

Figure 4(b) depicts the *group striping* for the same topology. Three pairs of S1 switches have identical uplink distributions and can achieve maximum possible throughput, 80 Gb/s. However, compared to the rotation striping, there are also more S1 switch pairs with reduced maximum achievable throughput, 60 Gb/s. Rotation striping reduces the variance of network capacity while group striping improves the probability of achieving ideal capacity among S1 switches.

The algorithm for generating the group striping runs in two phases as shown in Algorithm 5 and 6. We first create Q (Q = 3 in Figure 4(b)) sets of S1 switches such that S1 switches within each set can achieve maximum throughput for a destination S1 switch in the same set. In the second phase, the algorithm generates the striping for the remainder of S1 switches. Each of these S1 switches have a unique connection pattern to the S2 switches.

4.2 Link Weight Assignment

WCMP is a generic solution that makes no assumption about the underlying topology for assigning and reducing weights for links. For arbitrary topologies, the link weights can be computed by running max-flow min-cut algorithms of polynomial complexity and creating WCMP groups per source-destination pair. The weight reduction algorithm proposed in the section 3 would similarly reduce the number of table entries in such settings. With its efficient algorithms for weight assignment and weight reduction, WCMP can react quickly to changes in the topology due to inherent heterogeneity or failures and reprogram updated WCMP groups in just a few milliseconds.

In the topology shown in Figure 1, for any switch $S1_s$, its traffic flows destined to a remote switch $S1_d$ can be distributed among its N uplinks. Given: i) the varying effective capacities and ii) delivering uniform throughput among flows as the driving objective, more flows should be scheduled to uplinks with higher capacity than those with lower capacities. Therefore, when installing a WCMP group on $S1_s$ to balance its traffic to $S1_d$ among its N uplinks, we set the weight of each uplink proportional to its effective capacity which is either 1 or $\frac{p}{p+1}$.

5. System Architecture

We base our architecture and implementation for supporting WCMP around the Onix OpenFlow Controller[22] as shown in Figure 5. The Network Controller is the central entity that computes the routes between switches (hosts) and the

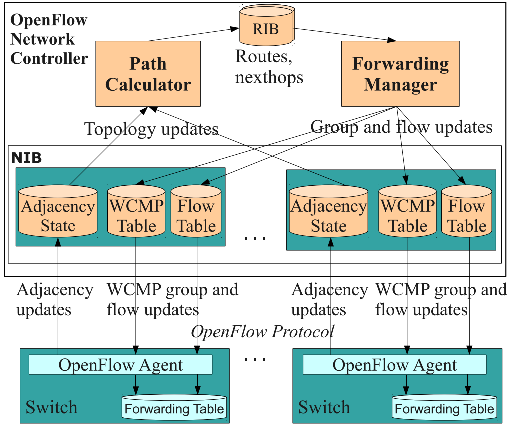


Figure 5: WCMP Software Architecture

weights for distributing the traffic among these routes based on the latest view of network topology. It also manages the forwarding table entries in the switches. We assume a reliable control channel between the network controller and the switches, e.g., a dedicated physical or logical control network. We implement the Network Controller functionality through three components: Network Information Base (NIB), Path Calculator and Forwarding Manager.

NIB: This component discovers the network topology by subscribing to adjacency updates from switches. Switch updates include discovered neighbors and the list of healthy links connected to each neighbor. The NIB component interacts with the switches to determine the current network topology graph and provides this information to the path calculator. The NIB component also caches switch's flows and WCMP group tables and is responsible for propagating changes to these tables to switches in case of topology changes due to failures or planned changes. In our implementation, all communication between NIB and switches is done using the OpenFlow protocol[27]. An OpenFlow agent running on each switch receives forwarding updates and accordingly programs the hardware forwarding tables.

Path Calculator: This component uses the network topology graph provided by the NIB and computes the available paths and their corresponding weights for traffic distribution between any pair of S1 switches. The component is also responsible for all other routes, e.g., direct routes to hosts connected to an S1 switch.

Forwarding Manager: This component manages the flow and WCMP tables. It converts the routes computed by the path calculator into flow rules, and next hops into WCMP groups to be installed at the switches. Since there are only limited hardware entries at each switch, this component also computes reduced weights for links such that WCMP performance with reduced weights is within tolerable bounds of the performance for the weights computed by the path calculator component. This optimization could also be implemented on the switch side.

Upon start-up, the Path Calculator queries the NIB for the initial topology and computes the routes and next hops. The Forwarding Manager converts the computed routes and next hops to flow rules and WCMP groups. Once the weights are computed, it invokes weight reduction algorithm for different switches in parallel and installs the weights into the switches. As illustrated in Figure 5, the Path Calculator further receives topology change events from the NIB (link up /down events), and recomputes next hops upon such events as link/switch failure or topology expansion. The updated next hops are then converted to updated WCMP groups to be installed on individual switches. The Path Calculator and the Forwarding Manager components together consist of 3K lines of C++. We use previously developed Onix and OpenFlow extensions for multipath support (OpenFlow 1.1 specification supports multipath groups).

6. Failure Handling

WCMP relies on the underlying routing protocol to be notified of switch/link failures and change the weights based on the updates. As such, WCMP is limited by the reaction time of the underlying protocol to react to changes in the topology.When a link failure is reported, the only additional overhead for WCMP is in recomputing the weights and programming the TCAM entries with the updated weights in the forwarding table. Since our weight reduction algorithms scale quadratically with number of ports in a group and are very fast for switches with port counts upto 96 ports, this overhead is extremely small.

Our simulations indicate that the weight computation and reduction algorithms can recompute weights in under 10 milliseconds for topologies with 100,000 servers when notified of a topology update. However, failures like link flapping may indeed result in unnecessary computations. In that case, WCMP will disable the port from the multipath group in order to avoid unnecessary weight computations. We also note that while a series of failures may result in recomputing (and reducing) weights for WCMP groups at a large number of switches, in general most failures will only require updating weights for only a small number of WCMP groups at few switches. The cost of updating weights is incurred only rarely but once the right weights are installed it results in significantly improving the fairness for both short and large flows in the network.

7. Evaluation

We evaluate WCMP using a prototype implementation and simulations. The prototype cluster network consists of six non-blocking 10Gb/s S1 switches interconnected by six non-blocking 10Gb/s S2 switches and 30 hosts with 10G NICs. Each host is running Linux kernel version 2.6.34 with ECN enabled in the TCP/IP stack. Our switches support OpenFlow and ECN, marking packets at 160KB for port buffer occupancy.

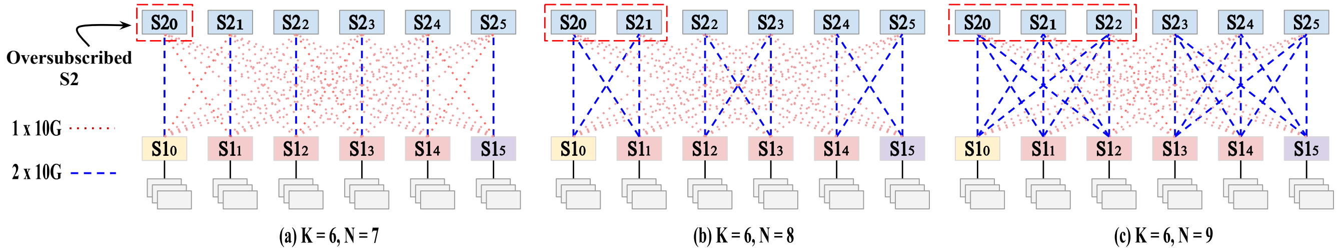
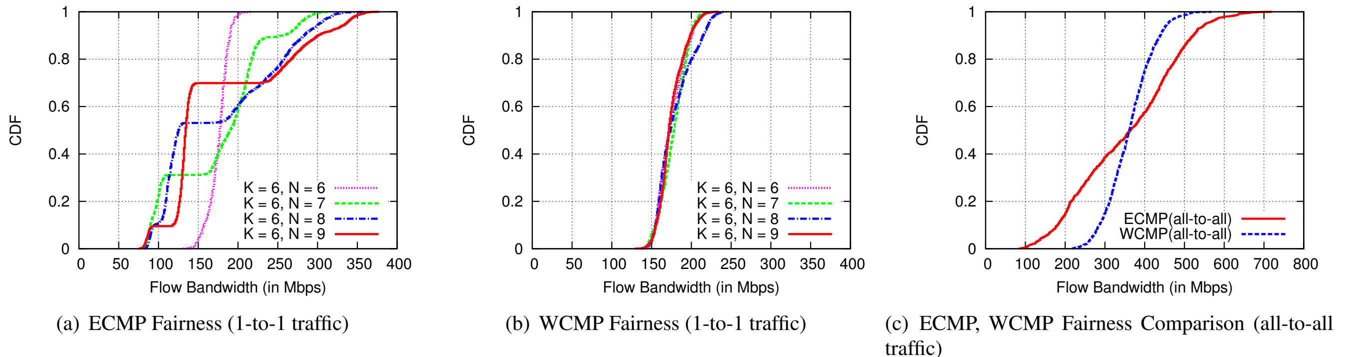


Figure 6: Testbed topologies (with Group striping) with increasing imbalance for one-to-one traffic pattern



(a) ECMP Fairness (1-to-1 traffic)

(b) WCMP Fairness (1-to-1 traffic)

(c) ECMP, WCMP Fairness Comparison (all-to-all traffic)

Figure 7: Comparing ECMP, WCMP performance on Clos topologies with different imbalance, different traffic patterns

We evaluate TCP performance with ECMP and WCMP hashing for topologies with different degrees of imbalance and for different traffic patterns including real data center traffic patterns from [7]. We extended the *htsim* simulator [19] for MPTCP to support WCMP hashing and evaluate WCMP benefits relative to MPTCP. We also measure the effectiveness of our weight reduction algorithms and the impact of weight reduction on flow bandwidth fairness. Overall, our results show:

- WCMP always outperforms ECMP, reducing the variation in the flow bandwidths by as much as $25\times$.
- WCMP complements MPTCP performance and reduces variation in flow bandwidth by $3\times$ relative to the baseline case of TCP with ECMP.
- The weight reduction algorithm can reduce the required WCMP table size by more than half with negligible oversubscription overhead.

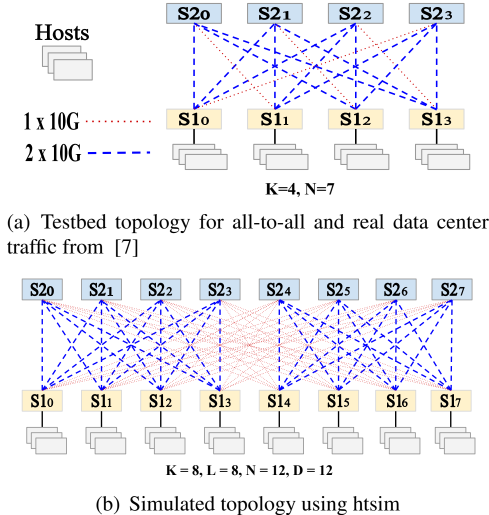
7.1 TCP Performance

We begin by evaluating TCP performance with ECMP and WCMP for different topologies and for different traffic patterns: one-to-one, all-to-all and real data center traffic.

One-to-one Traffic: We first compare the impact of striping imbalance on TCP flow bandwidth distribution for ECMP and WCMP hashing. To vary the striping imbalance, we increase the number of oversubscribed S2 switches, to which a pair of S1 switches is asymmetrically connected. We manually rewire the topology using our group striping algorithm into the three topologies shown in Figure 6 and generate traffic between asymmetrically connected switches $S1_0$ and $S1_5$. Each of the nine hosts connected to source switch $S1_0$ transmit data over four parallel TCP connections to a unique host on destination switch $S1_5$ over long flows. We plot the CDF for the flow bandwidths in Figure 7.

Figure 7(b) shows that WCMP effectively load balances the traffic such that all flows receive almost the same bandwidth despite the striping imbalance. Bandwidth variance for ECMP on the other hand increases with the striping imbalance as shown in Figure 7(a). We make the following observations from these CDFs: (1) For ECMP, the number of *slower* flows in the network increases with the striping imbalance in the topology. More importantly, for imbalanced topologies, the minimum ECMP throughput is significantly smaller than the minimum WCMP throughput, which can lead to poor performance for applications bottlenecked by the slowest flow. (2) the variation in the flow bandwidth increases with the imbalance in the topology. WCMP reduces the variation in flow bandwidth by $25\times$ for the topology where K=6, N=9. High variations in flow bandwidth make it harder to identify the right bottlenecks and also limit application performance by introducing unnecessary skew.

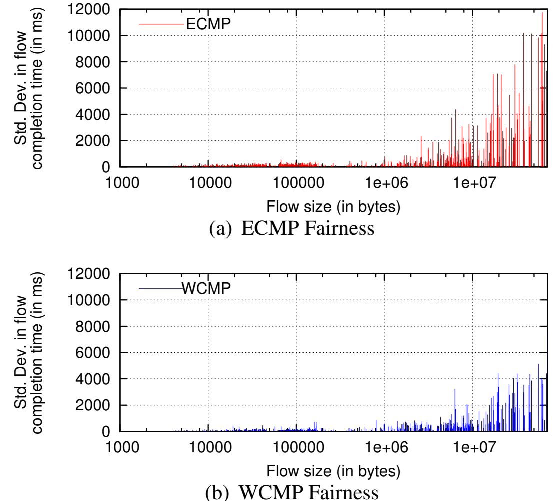
All-to-all Traffic: We next compare ECMP and WCMP hashing for all-to-all communication for the topology shown in Figure 8(a) and present the results in Figure 7(c). Each host communicates with hosts on all the remote switches



(a) Testbed topology for all-to-all and real data center traffic from [7]

(b) Simulated topology using htsim

Figure 8: Evaluation topologies



(a) ECMP Fairness

(b) WCMP Fairness

Figure 9: Comparing ECMP and WCMP performance for data center traffic measured by Benson et. al. [7]

over long flows. Again, the load balancing is more effective for WCMP than for ECMP. This graph provides empirical evidence that the weighted hashing of flows provides fairer bandwidth distribution relative to ECMP even when the traffic is spread across the entire topology. In this case, WCMP lowered variation in flow bandwidth by $4\times$ and improved minimum bandwidth by $2\times$.

Real Data Center Traffic: We also compare ECMP and WCMP hashing for mapreduce style real data center traffic as measured by Benson et. al. [7]. We generate traffic between randomly selected inter-switch source-destination pairs for topology shown in Figure 8(a). The flow sizes and flow inter-arrival times have a lognormal distribution as described in [7].

Figure 9 shows the standard deviation (std. dev.) in the completion time of flows as a function of the flow size. Though both ECMP and WCMP are quite effective for small flows, for flow sizes greater than 1MB, the variation in the flow completion times is much more for ECMP compared to WCMP, even for flows of the same size. Moreover, this variation increases as the flow size increases.In summary, for real data center traffic, WCMP reduced the std. dev. in bandwidth by $5\times$ on average and, more importantly, $13\times$ at 95%-ile relative to ECMP while average bandwidth improved by 20%.

7.2 MPTCP Performance

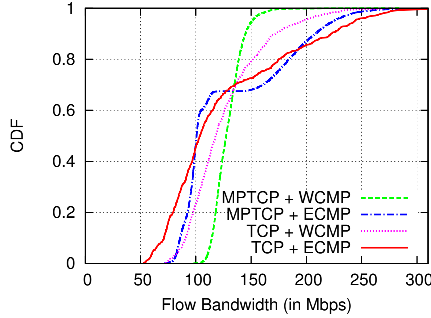
We extended the packet level MPTCP simulator, htsim to support WCMP hashing and evaluate its impact on MPTCP performance. The simulated topology consists of eight 24-port S1 switches (12 uplinks and 12 downlinks) and eight 12-port S2 switches (K=8, L=8, N=12, D=12) as shown in Figure 8(b). Each S1 switch is also connected to 12 hosts. We use 1000 byte packets, 1 Gb/s links, 100KB buffers and $100\mu s$ as per-hop delay. We use a permutation traffic matrix, where each host sends data to a randomly chosen host on a remote switch. We consider two scenarios (i) all flows start/stop at the same time, (on-off data center traffic pattern Figure 10(a)), (ii) flows start at different times, subjected to varying level of congestion in the network (Figure 10(b)). We evaluate all 4 possible combinations of TCP, MPTCP (with 8-subflows per TCP flow) with ECMP and WCMP.

The results in Figure 10(a) and 10(b) show that MPTCP with WCMP clearly outperforms all other combinations. It improves the minimum flow bandwidth by more than 25% and reduces the variance in flow bandwidth by up to $3\times$ over MPTCP with ECMP. While WCMP with TCP outperforms ECMP with TCP for the on-off communication pattern, it has to leverage MPTCP for significant improvement for the skewed traffic patterns. This is because MPTCP can dynamically adjusts the traffic rates of subflows to avoid hot spots while WCMP is useful for addressing the bandwidth variation due to structural imbalance in the topology.

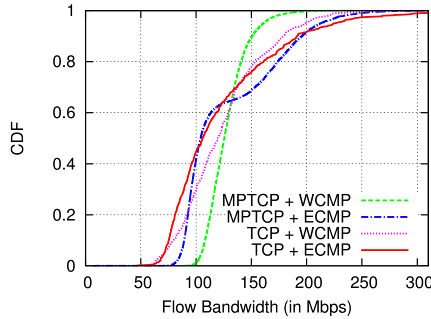
7.3 Weight Reduction Effectiveness

Next, we evaluate the effectiveness of our weight reduction algorithms. We analyze two topologies, with the number of S1 uplinks (N) equal to 96 and number of S2 downlinks (D) as 32 and other where N = 192 and D = 64. We vary the number of S1 switches from 5 to 19 and compute the maximum number of table entries required at an S1 switch.

We run Algorithm 1 to reduce this number with different oversubscription limits (1.05 and 1.1) and show the results in Figure 11. Without any weight reduction, the maximum number of multipath table entries at the switch is $>$10k for the case where S1 switches have 192 uplinks, while the size of TCAM/SRAM on commodity switches is usually only 4K. The reduction algorithm reduces the required multipath table entries by more than 25% while incurring only 1.05 oversubscription (Figure 11(b)). It can further fit the entire set of WCMP groups to the table of size 4K at the maximum

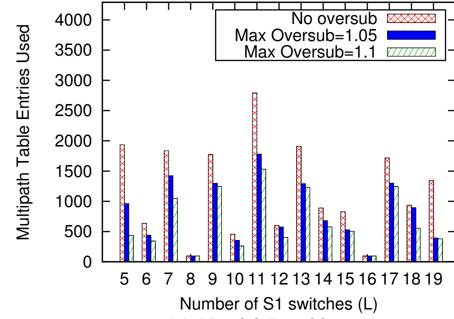


(a) htsim ON-OFF traffic

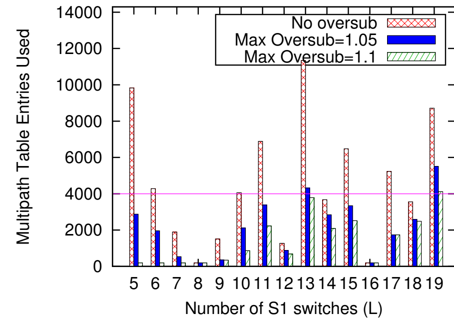


(b) htsim skewed traffic

Figure 10: Evaluating MPTCP performance with WCMP



(a) N = 96, D = 32



(b) N = 192, D = 64

Figure 11: Table entries before and after running Algorithm 1 for reducing WCMP groups at the switch requiring maximum table entries

oversubscription of only 1.1. We also ran the LP solver for reducing the size of the WCMP groups at the S1 switch requiring maximum table entries. In all cases, the results from our weight reduction algorithm were same as the optimal result. Figure 11 further shows that without the reduction algorithm, the maximum number of table entries grows by almost $3\times$ when the switch port counts were doubled. The reduction algorithm significantly slows such growth with limited impact on the fairness.

7.4 Weight Reduction Impact on Fairness

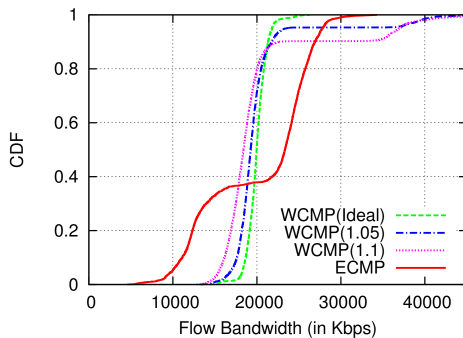


Figure 12: Impact of weight reduction on fairness

Our weight reduction algorithms trade-off achieving ideal fairness in order to create WCMP groups of smaller size. Since our evaluation testbed was small and did not require weight reduction, we simulate a large topology for evaluating weight reduction impact on fairness. We instantiate a topology with 19 S1 switches with 96 uplinks each, 57 S2 switches with 32 downlinks each and 1824 hosts using htsim (K=57, L=19, N=96, D=32). With group striping for this topology, we have two groups of six S1 switches, and each S1 switch in a group has identical striping to the S2 switches. The remaining seven S1 switches are asymmetrically connected to all the other S1 switches. We run Algorithm 1 with different oversubscription limits to create WCMP groups with reduced weights. We generate traffic using a random permutation traffic matrix, where each host sends data to another host on a remote switch with long flows. Figure 12 shows the results of this experiment. With a oversubscription limit of 1.05, we achieve a 70% reduction in the maximum number of multipath table entries required (Figure 11(a)) with very little impact on the fairness. As we increase the oversubscription limit to 1.1, the fairness reduces further, but WCMP still outperforms ECMP.

8. Conclusion

Existing techniques for statically distributing traffic across data center networks evenly hash flows across all paths to a particular destination. We show how this approach can lead to significant bandwidth unfairness when the topology is inherently imbalanced or when intermittent failures exacerbate imbalance. We present WCMP for weighted flow hashing across the available next-hops to the destination and show

how to deploy our techniques on existing commodity silicon. Our performance evaluation on an OpenFlow-controlled network testbed shows that WCMP can substantially reduce the performance difference among flows compared to ECMP, with the potential to improve application performance and network diagnosis; and complements dynamic solutions like MPTCP for better load balancing.

Acknowledgments

We thank our anonymous reviewers and our shepherd, Hitesh Ballani, for their feedback and helpful comments. We also thank the authors of [28] for sharing their htsim implementation with us.

References

[1] J. H. Ahn, N. Binkert, A. Davis, M. McLaren, and R. S. Schreiber. HyperX: Topology, Routing, and Packaging of Efficient Large-Scale Networks. In *Proc. of SC*, November 2009.

[2] M. Al-Fares, A. Loukissas, and A. Vahdat. A Scalable, Commodity Data Center Network Architecture. In *Proc. of ACM SIGCOMM*, August 2008.

[3] M. Al-Fares, S. Radhakrishnan, B. Raghavan, N. Huang, and A. Vahdat. Hedera: Dynamic Flow Scheduling for Data Center Networks. In *Proc. of Usenix NSDI*, April 2010.

[4] D. Applegate, L. Breslau, and E. Cohen. Coping with Network Failures: Routing Strategies for Optimal Demand Oblivious Restoration. In *Proc. of SIGMETRICS*, June 2004.

[5] L. A. Barroso, J. Dean, and U. Hölzle. Web Search for a Planet: The Google Cluster Architecture. *IEEE Micro*, Volume 23, March-April 2003.

[6] Broadcom's OpenFlow Data Plane Abstraction. `http://www.broadcom.com/products/Switching/Software-Defined-Networking-Solutions/OF-DPA-Software`.

[7] T. Benson, A. Akella, and D. A. Maltz. Network Traffic Characteristics of Data Centers in the Wild. In *Proc. of ACM IMC*, November 2010.

[8] M. Chowdhury, M. Zaharia, J. Ma, M. I. Jordan, and I. Stoica. Managing Data Transfers in Computer Clusters with Orchestra. In *Proc. of ACM SIGCOMM*, August 2011.

[9] T. H. Cormen, C. Stein, R. L. Rivest, and C. E. Leiserson. *Introduction to Algorithms*. McGraw-Hill Higher Education, 2nd edition, 2001.

[10] A. R. Curtis, W. Kim, and P. Yalagandula. Mahout: Low-Overhead Datacenter Traffic Management using End-Host-Based Elephant Detection. In *Proc. of IEEE INFOCOM*, April 2011.

[11] A. R. Curtis, J. C. Mogul, J. Tourrilhes, P. Yalagandula, P. Sharma, and S. Banerjee. DevoFlow: Scaling Flow Management For High-Performance Networks. In *Proc. of ACM SIGCOMM*, August 2011.

[12] J. Dean and S. Ghemawat. MapReduce: Simplified Data Processing on Large Clusters. In *Proc. of Usenix OSDI*, December 2004.

[13] Gnu Linear Programming Kit. `http://www.gnu.org/software/glpk`.

[14] 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. In *Proc. of ACM SIGCOMM*, August 2009.

[15] C. Guo, G. Lu, D. Li, H. Wu, X. Zhang, Y. Shi, C. Tian, Y. Zhang, and S. Lu. BCube: A High Performance, Server-centric Network Architecture for Modular Data Centers. In *Proc. of ACM SIGCOMM*, August 2009.

[16] B. Heller, S. Seetharaman, P. Mahadevan, Y. Yiakoumis, P. Sharma, S. Banerjee, and N. McKeown. ElasticTree: Saving Energy In Data Center Networks. In *Proc. of Usenix NSDI*, April 2010.

[17] C. Hopps. Analysis of an Equal-Cost Multi-Path Algorithm. RFC 2992, IETF, November 2000.

[18] HP SDN Ecosystem. `http://h17007.www1.hp.com/us/en/networking/solutions/technology/sdn/`.

[19] MPTCP htsim implementation. `http://nrg.cs.ucl.ac.uk/mptcp/implementation.html`.

[20] J. Kim and W. J. Dally. Flattened Butterfly: A Cost-Efficient Topology for High-Radix Networks. In *ISCA*, June 2007.

[21] M. Kodialam, T. V. Lakshman, and S. Sengupta. Efficient and Robust Routing of Highly Variable Traffic. In *Proc. of ACM HotNets*, November 2004.

[22] T. Koponen, M. Casado, N. Gude, J. Stribling, L. Poutievski, M. Zhu, R. Ramanathan, Y. Iwata, H. Inoue, T. Hama, and S. Shenker. Onix: A Distributed Control Platform for Large-scale Production Networks. In *Proc. of Usenix OSDI*, October 2010.

[23] V. Liu, D. Halperin, A. Krishnamurhty, and T. Anderson. F10: A Fault-Tolerant Engineered Network. In *Proc. of Usenix NSDI*, April 2013.

[24] Memcached. `http://www.memcached.org`.

[25] J. Moy. OSPF Version 2. RFC 2328, Internet Engineering Task Force, April 1998.

[26] MPLS-TE. `http://blog.ioshints.info/2007/02/unequal-cost-load-sharing.html`.

[27] Openflow. `www.openflow.org`.

[28] C. Raiciu, S. Barre, C. Pluntke, A. Greenhalgh, D. Wischik, and M. Handley. Improving Datacenter Performance and Robustness with Multipath TCP. In *Proc. of ACM SIGCOMM*, August 2011.

[29] C. Villamizar. OSPF Optimized Multipath (OSPF-OMP), draft-ietf-ospf-omp-02 , February 1999.

[30] H. Wu, G. Lu, D. Li, C. Guo, and Y. Zhang. MDCube: A High Performance Network Structure for Modular Data Center Interconnection. In *Proc. of ACM CoNEXT*, December 2009.

[31] X. Xiao, A. Hannan, and B. Bailey. Traffic Engineering with MPLS in the Internet. *IEEE Network Magazine*, 2000.

[32] Y. Zhang and Z. Ge. Finding Critical Traffic Matrices. In *Proc. of DSN*, June 2005.