Dahu: Improved Data Center Multipath Forwarding

Radhakrishnan, Sivasankar
Kapoor, Rishi
Tewari, Malveeka
et al.

2013-02-11

# Dahu: Improved Data Center Multipath Forwarding

Sivasankar Radhakrishnan*, Rishi Kapoor*, Malveeka Tewari*,
George Porter*, Amin Vahdat†*
* University of California, San Diego † Google Inc.
{sivasankar, rkapoor, malveeka, gmporter, vahdat}@cs.ucsd.edu

## ABSTRACT

Solving "Big Data" problems requires bridging massive quantities of compute, memory, and storage, which requires significant amounts of bisection bandwidth. Topologies like Fat-tree [4], VL2 [18], and HyperX [3] achieve a scale-out design by leveraging multiple paths from source to destination. However, traditional routing protocols are not able to effectively utilize these links while also harnessing spare network capacity to better statistically multiplex network resources.

In this work we present Dahu[1], a switch mechanism for efficiently load balancing traffic in multipath networks. Dahu avoids congestion hot-spots by dynamically spreading traffic uniformly across links, and forwarding traffic over non-minimal paths where possible. By performing load balancing primarily using local information, Dahu can act more quickly than centralized approaches, and responds to failure gracefully. Our evaluation shows that Dahu delivers up to 50% higher throughput relative to ECMP in an 8,192 server Fat-tree network and up to 500% improvement in throughput in large scale HyperX networks with over 130,000 servers.

## 1. INTRODUCTION

The scale and complexity of data center networks has grown with the size, scope, and ubiquity of data center hosted applications. The network fabric plays a critical role in bridging compute, storage, and memory, and is subject to several strict requirements: delivering high bisection bandwidth with low latency, ensuring predictable performance, and gracefully handling the failure of individual switches and links.

Of particular relevance is effectively utilizing the aggregate capacity of multiple network links. An ideal network topology would perhaps consist of a core with a small number of switches for fault tolerance, each switch with perhaps hundreds of links whose aggregate capacity is proportional to the product of the number of hosts in the data center and their individual NIC speed. Unfortunately, for data centers with tens of thousands of servers with 10Gbps NICs, this would require individual Tbps scale links in the core. Since such scale up bandwidth is not possible for individual links, new network topologies leverage multiple, parallel paths between sources and destinations [3, 4, 18, 19, 32] spread across thousands of switches with link speeds that roughly match the speed of the host NICs. However, current switch forwarding protocols are not able to effectively utilize massive multipath networks, resulting in an opportunity to rethink the forwarding plane in modern networks.

Today, data center switches are built for Equal-Cost Multi-Path Routing or ECMP, which performs static hashing of flows across a fixed set of equal-cost links to a destination. As a result, they restrict the network topology. Simple hierarchical topologies like Fat-trees [4] can leverage shortest path forwarding; however recently proposed direct network topologies like HyperX, BCube, and Flattened Butterfly [3, 19, 22] employ paths of different lengths to deliver large aggregate bandwidth among servers, leaving these non-shortest paths out of consideration for ECMP forwarding. Thus, during periods of localized congestion and hot-spots, traffic to a destination could potentially achieve higher throughput, if these alternate longer paths are also used for forwarding.

Further, even in hierarchical networks, ECMP makes it hard to route efficiently under failures. In the presence of failures, the network is no longer completely symmetric, and some non-shortest paths can be utilized to improve network utilization.

Commodity switches are designed this way because data center topologies were typically hierarchical and the networks were of sufficiently small scale that failures were not a big issue. These restrictions on the topology and routing also mean that higher level adaptive protocols like MPTCP [29] are unable to take advantage of the full capacity of the underlying network because all the paths are not exposed to them through routing/forwarding tables.

Two primary challenges must be addressed to enable non-shortest-path routing. First, care must be taken to avoid persistent forwarding loops. Second, longer paths result in higher network latency and consume more bandwidth hops, so they should only be leveraged when there is not sufficient bandwidth along shortest paths. Thus, we require a loop-free mechanism for utilizing non-shortest-paths that balances the need for higher throughput with the need to control end-to-end network latency.

[1] Dahu is a legendary creature well known in France with legs of differing lengths.

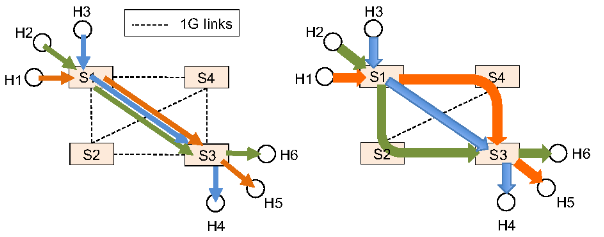
We present Dahu, a lightweight, switch mechanism that enables us to leverage non-shortest paths with loop-free forwarding, and operates locally, with small switch state requirements and minimal additional latency. It supports dynamic flow-level hashing across links, resulting in higher network utilization. For Fat-tree networks, we address the local hash imbalance that occurs with ECMP using only local information in the switches; for HyperX topologies, we show how to enable commodity silicon to dynamically leverage non-shortest paths. Dahu makes the following contributions: (1) A *virtual port* abstraction that enables dynamic multipath traffic engineering, (2) Hardware primitives to efficiently utilize non-minimal paths in different topologies with a modest increase in switch state while preventing persistent forwarding loops, (3) A decentralized load balancing algorithm and heuristic, (4) Exploration of the practicality of building a Dahu-enabled switch, and (5) Large scale simulations on networks with over 130K servers to evaluate the performance of Dahu. Our evaluation shows that Dahu delivers up to 50% higher throughput relative to ECMP in an 8,192 server Fat-tree network and up to 500% throughput improvement in large HyperX networks with over 130,000 servers.

## 2. MOTIVATION AND REQUIREMENTS

**The potential of non-shortest-path routing:** Direct connect networks like HyperX provide an interesting point in the design space of networks, since they provide good performance for most realistic traffic patterns, at lower cost than fully-provisioned Clos networks, which do well for arbitrary communication patterns. These networks typically have many varied paths between any two points in the network, however not all of the same length. Using the bandwidth offered by these longer paths requires a non-minimal routing scheme, with corresponding support from both switch hardware and software. The term *derouting* is used to refer to a non-minimal forwarding choice by a switch.

Consider a simple mesh network of four switches, as shown in Figure 1. In (a), the shortest path connecting the sources and destinations is congested, with a resulting total bandwidth of 1 Gbps. However, by sending some traffic flows over non-shortest-path links, as shown in (b), the total throughput can be increased to 3 Gbps.

Looking at larger topologies, consider an $L$-dimensional HyperX network, where each switch has a coordinate $(1..S)$ in each dimension. Switches whose coordinates differ in a single dimension are directly connected, and an *offset* dimension for a pair of switches is one in which their coordinates differ. As the network size grows, so does the number of non-minimal paths as shown in Table 1. The availability of a large number of non-shortest paths indicates potentially unused network capacity during periods of localized congestion. One goal of our work is to recapture this capacity while still leveraging shortest paths for the common case.



(a) Throughput = 1Gbps (b) Throughput = 3Gbps

**Figure 1:** (a) Shortest, and (b) Non-shortest path routing

| HyperX Topo. | Offset Dim. | Src-Dst Switch Pairs | Shortest Paths | All Paths |
|---|---|---|---|---|
| L=3, S=4 | 1 | 288 | 1 | 15 |
| | 2 | 864 | 2 | 56 |
| | 3 | 864 | 6 | 204 |
| L=3, S=14 | 1 | 53,508 | 1 | 65 |
| | 2 | 695,604 | 2 | 256 |
| | 3 | 3,014,284 | 6 | 984 |

**Table 1:** Path choices in HyperX topology for shortest path routing vs. routing with at most one hop allowed to make a non-minimal forwarding choice. L=number of dimensions, S=maximum value of coordinate in each dimension and offset dim is the number of dimensions in which the source and destination switch coordinates differ.

**The need for dynamic load balancing:** Static forwarding schemes like ECMP place flows on links based on a hash of the flow identifiers, and do not account for the current link demand or the flow's rate. This can result in imbalance, which can significantly reduce the observed throughput compared to the innate network capacity [5, 9]. One of Dahu's goals is to provide hardware abstractions that enable better dynamic traffic engineering and address the static nature of ECMP hashing.

**Alternate traffic engineering approaches:** There have been several recent proposals for better traffic management in data centers. Centralized approaches such as Hedera [5] and MicroTE [9] advocate a central fabric scheduler which periodically measures traffic demands, estimates future demand for bandwidth intensive flows in the network and computes good paths for them which would maximize bandwidth utilization. The fabric scheduler can run fairly complicated scheduling algorithms and act on global knowledge of the network state. However this control delay means that such schemes can only react to changes in traffic at timescales on the order of hundreds of milliseconds at best, i.e. they are only effective for long lived flows. Further, scaling such centralized approaches to very large numbers of flows in large scale data centers presents challenges in terms of the state requirements on switches.

MPTCP [29] proposes an entirely host based traffic engineering approach. With MPTCP, each flow is split into multiple subflows which can be transmitted over separate paths in the network, probing for congestion along the paths. MPTCP improves the achieved network bandwidth by transferring more data over less congested subflows thereby avoiding additional data transfer over already congested paths. However, MPTCP only decides how much data to send on each subflow, the paths for the subflows are chosen by the network. In topologies like HyperX with many non-shortest paths between hosts, there is still a need for better path selection in the switches to help MPTCP effectively leverage the path diversity. Without this support, MPTCP will require a large number of subflows to probe all the different paths, making it impractical for short flows. We illustrate this through simulations in Section 6.5. Exposing non-shortest paths to MPTCP would further require additional network support and changes to routing schemes for ensuring that packets do not loop forever in the network and eventually reach their destinations without significantly increasing path lengths. This suggests that a complementary *in-network* technique which can reduce congestion for short lived flows and select a set of good paths over which MPTCP transmits data for longer flows would be beneficial.

Centralized or host based traffic engineering solutions are effective in improving bandwidth utilization of large long lived flows. Decentralized approaches, on the other hand, can react faster for better managing short flows as they rely on just local information or minimal knowledge from peers. Such decentralized approaches can complement the other approaches which benefit long lived flows and help achieve global optimality over longer time scales. Dimensionally Adaptive Load balanced (DAL) routing [3] is a technique specialized for HyperX networks which performs per-packet adaptive routing based on local information. However, DAL requires HyperX specific modifications to switches and packet headers. It might also necessitate modifications to the end host transport protocol to deal with extensive packet reordering that could occur due to per-packet adaptive routing. We seek to design a more generic solution where the switch hardware is not tied to topology and one which wouldn't cause as much in-network reordering as a per-packet load balancing approach.

**Dahu requirements and design decisions:** Dahu is driven by several design goals. First, any proposed changes to switch hardware should be simple enough to be realizable with current technology. The switch state required to implement the new features should be small. Switches should still make flow level forwarding decisions–i.e. packets of a particular flow should follow the same path through the network to the extent possible. This avoids excessive packet reordering, which can have undesirable consequences for TCP. Moving flows to alternate paths periodically at coarse time scales (e.g., of several RTTs) is acceptable.

Although Dahu supports non-minimal routing, it must prefer shorter paths when possible. Using shorter paths results in fewer switch hops and likely lower end-to-end latency. We only start using non-minimal paths when shortest paths do not have sufficient capacity. We choose to load balance traffic primarily using local decisions in each switch which helps react quickly to changing traffic demands and temporary hot spots. In addition, our mechanisms must interoperate with other routing and traffic engineering schemes. We allow that some traffic in the network may not be reroutable using our local load balancing heuristics, e.g., if there are some bandwidth reservations along certain paths, or if a centralized controller like Hedera is placing elephant flows along globally optimal paths [5].

Failures need to be handled gracefully, and re-routing of flows upon failure should only affect a small subset of flows (so that the effect of failures is proportional to the region of the network that has failed). Load balancing should be supported at small time granularity of a few milliseconds. To prevent traffic herds, we should not move many flows in the network around when a single path is congested. Rather, it should be possible to make finer-grained decisions and migrate a smaller subset of flows to alternate paths. While we evaluate against two major multipath network topologies, Dahu is not dependent on network topology and the same switches should be usable in different networks.

Dahu tries to achieve these targets through a combination of switch hardware and software enhancements. In Section 3, we first illustrate what goes into the switch by describing the novel Dahu hardware primitives. Then, we describe enhancements to the switch software and control logic to take advantage of these hardware primitives to better utilize the network in Section 4.

# 3. SWITCH HARDWARE PRIMITIVES

Dahu proposes new hardware primitives which enable better ways of utilizing the path diversity in data centers and addresses some of the limitations of ECMP.

## 3.1 Port Groups With Virtual Ports

The goal of ECMP is to spread traffic over multiple equal-cost paths to a particular destination. Internally, the switch must store enough state to track which ports can be used to reach a particular destination prefix. A common mechanism is storing a list of egress ports in the routing table, represented as a bitmap. Dahu augments this approach by adding a layer of indirection: each router prefix points to a set of *virtual ports*, and each virtual port is mapped to a physical port. In fact, the number of virtual ports can be much larger than number of physical ports. We define a *port group* as a collection of virtual ports mapped to their corresponding physical ports (a many-to-one mapping). The forwarding table is modified to allow a pointer to a port group for a particular destination prefix instead of a physical egress port. When multiple egress choices are available for a particular destination prefix, the forwarding table just points to a particular port group.

When a packet is received by the switch, it looks up the port group for the destination prefix from the forwarding ta-

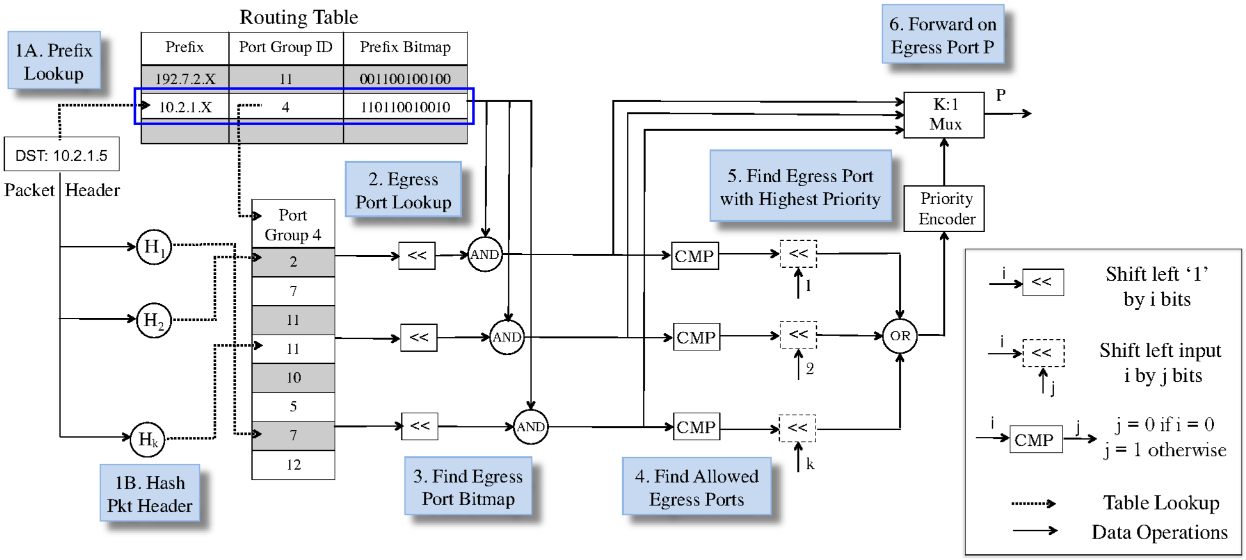


**Figure 2:** Datapath pipeline for packet forwarding in Dahu.

ble. It computes a hash based on the packet headers, similar to ECMP, and uses this hash value to index into the port group to choose a virtual port. Finally, it forwards the packet on the egress port to which the virtual port is mapped. This port group mechanism adds one additional level of indirection in the switch output port lookup pipeline, which we use to achieve better load balancing, and support for non-shortest network paths.

In hardware, a port group is simply an array of integers. The integer at index $i$ in the array is the port number of the egress port to which virtual port $i$ in that port group is mapped to. Each port group has some fixed number of virtual ports, larger than the number of egress ports in the switch. Multiple destination prefixes in the forwarding table may point to the same port group. For the rest of this paper, the term *member port* of a port group is used to refer to a physical port which has some virtual port in the port group mapped to it.

The virtual port to egress port mapping provides an abstraction for dynamically migrating traffic from one egress port to another within any port group. Each virtual port is mapped to exactly one egress port at any time, but this mapping can be changed dynamically. A key advantage of the indirection approach is that when a virtual port is remapped, only those flows which hash to that virtual port are migrated to other egress ports. Other flows remain on their existing paths and their packets don't get re-ordered. All flows that map to a virtual port can only be remapped as a group to another egress port. Thus, virtual ports dictate the granularity of traffic engineering. The larger the number of virtual ports per port group, the finer the granularity at which traffic can be moved around. We propose that each port group have a relatively large number of virtual ports – on the order of 1,000 for high-radix switches with 64-128 physical ports. That means each virtual port is responsible for an average of 0.1% or less of the total traffic to the port group. If required, the forwarding table can be augmented with more fine grained forwarding entries.

Each port group keeps a set of counters corresponding to each member port indicating how much traffic the port group has forwarded to that port. While having a traffic counter for each virtual port provides fine grained state, it comes at a higher cost for two reasons: (1) The amount of memory required to store individual counters is fairly large. For example, a switch with 64 port groups, 1,024 virtual ports per port group, and 64 bit traffic counters would need 512KB of on-chip memory just to store these port group counters. (2) Reading all 65,536 counters from hardware to switch software would take a long time, increasing the reaction time of traffic engineering schemes that use all these counters for computations. Dahu uses the port group mechanism and associated counters to implement a novel load balancing scheme which we describe in Section 4.

## 3.2 Allowed Port Bitmaps

Port groups enable the use of multiple egress port choices for a single destination prefix. However, it is sometimes useful to have many egress ports in a particular port group, but use only a subset of them for forwarding traffic to a particular prefix. One reason to do this is to avoid forwarding on failed egress ports. Consider two prefixes in the routing table $F_1$ and $F_2$ both of which point to port group $G_1$ which has member egress ports $P_1, P_2, P_3, P_4$. Suppose a link fails and egress port $P_4$ cannot be used to reach prefix $F_1$, whereas all member ports can still be used to reach $F_2$. Now, one option is to create another port group $G_2$ with member ports $P_1, P_2$ and $P_3$ only, for prefix $F_1$. This can quickly result in the creation of many port groups for a large network which might experience many failures at once. We propose a more scalable approach where we continue to use the existing port group, but restrict the subset of member ports which are used for forwarding traffic to a particular destination.

For each destination prefix in the routing table, we store an *allowed port bitmap*, which indicates the set of egress ports which are allowed to be used to reach the prefix. The allowed port bitmap is as wide as the number of egress ports in the switch, and only the bits corresponding to the allowed egress ports for the destination prefix are set. A potential solution to restrict forwarding to the allowed egress ports is to compute a hash for the packet and check if the corresponding port group virtual port maps to a allowed egress port. If the egress port is not allowed, then we compute another hash function for the packet and repeat this process until we find an allowed egress port.

To pick an allowed port more efficiently, we propose a parallel version of this scheme where the switch computes 16 different hash functions for the packet in parallel. The egress ports corresponding to these virtual ports are looked up from the port group and those egress port numbers in bitmap form are bitwise AND'd with the allowed port bitmap for the destination prefix from the routing table. The result is fed into a priority encoder to pick the first valid allowed egress port among the hashed choices. In case none of the 16 hash functions picked an allowed egress port, we generate another set of 16 hash values for the packet and retry. This may be repeated some fixed number of times (say 2) to bound the latency for output port lookup. If an allowed egress port is still not found, we just fall back to randomly picking one of the allowed egress ports, i.e. we ignore the port group mechanism for this packet and just hash it on to one of the allowed egress ports directly. We explore other uses of the allowed port bitmap in Section 4.1.2.

Figure 2 illustrates the switch egress port lookup pipeline incorporating both port groups and allowed egress port mechanisms. We propose that the switch routing table have support for some fixed number of such allowed port bitmaps for each prefix and have a selector field (say a 2 bit field) to indicate which bitmap should be used. The selector 00 refers to an *allow all* bitmap and indicates a hardwired default bitmap $B_{all}$ where all bits are set, i.e. all member ports of the port group are allowed. Selector 01 refers to a *shortest path* bitmap $B_{short}$ which is an always available bitmap that corresponds to the set of shortest path egress ports to reach the particular destination prefix. Unlike the $B_{all}$ bitmap, $B_{short}$ is not in-built and has to be updated by the switch control logic if port group forwarding is used for the prefix. Its use is described in Sections 3.3 and 4.1.2. Any available higher numbered bitmap selectors may be used for restricting the set of egress ports based on other considerations for the destination prefix.

### 3.3 Eliminating Forwarding Loops

Dahu uses non-shortest path forwarding to avoid congestion hot-spots when possible. The number of times a particular packet has been derouted is referred to as the *derouting count*. An immediate concern with derouting is that it can result in forwarding loops. Dahu prevents persistent forwarding loops in the following manner. Network packets are augmented with an additional 4-bit field in the IP header which stores the derouting count. Switches increment this field only if they choose a non-minimal route for the packet. Servers set this field to zero when they transmit traffic. In practice, the derouting count need not be a new header field, e.g., part of the TTL field or an IP option may be used instead.

When a switch receives a packet, it checks the header and if the derouting count has reached a maximum derouting count threshold for the network, then it only uses shortest path forwarding for the packet. This is enforced using the allowed port bitmap mechanism described earlier. The $B_{short}$ allowed port bitmap for the destination prefix is used whenever the derouting count has reached the maximum threshold. This ensures that the packet will eventually be forced along the shortest paths if the maximum derouting count is reached. The derouting count is also used while computing the packet hash. If a packet loops through the network and revisits a switch, its derouting count will have changed. The resulting change to the hash value will likely forward the packet along a different path to the destination. Each switch also ensures that a packet is not forwarded back on the ingress port that the packet arrived on. Further, in practice, only a few deroutings are required to achieve benefits from non-minimal routing and the derouting count threshold for the network can be configured by the administrator as appropriate. These factors ensure that any loops that occur due to non-minimal routing are infrequent and don't hinder performance.

As with current distributed routing protocols, transient loops may occur during routing convergence in certain fault scenarios. Dahu uses standard IP TTL defense mechanisms to ensure that packets eventually get dropped if there are loops during routing convergence.

## 4. SWITCH SOFTWARE

In this section, we look at how Dahu's hardware primitives can more efficiently utilize the network's available capacity. We first describe how to leverage non-minimal paths, and then look at dynamic traffic engineering to address local hash imbalances in switches. These techniques rely on Dahu's hardware primitives, but are independent of each other and may be deployed separately.

### 4.1 Non-Minimal Routing

As described earlier, ECMP constrains traffic routes to the set of shortest paths between any pair of switches. While this keeps path lengths low, it can also impose artificial constraints on available bandwidth. For example, in direct connect networks like HyperX [3], there are many paths from a source to a destination providing large aggregate bandwidth, but not all paths are of the same length. The non-shortest paths may not have any other traffic on them and will unfortunately remain unused if the forwarding scheme only uses shortest paths. In a HyperX switch $S_s$, there are three classes of egress port choices to reach any destination switch $S_d$.

1. Set of shortest path egress ports to reach $S_d$. The size of the set is equal to the number of offset dimensions, i.e. dimensions in which the switch coordinates of $S_s$ and $S_d$ differ.
2. Set of egress ports connected to neighbors along offset dimensions excluding the shortest path egress ports. Each of these neighbors is at the same distance from $S_d$, equal to the shortest path distance from $S_s$ to $S_d$.
3. All the remaining ports that are connected to other switches. The egress ports which connect to neighbors along dimensions already aligned with $S_d$ are members of this class. Each of these neighbors is one additional hop away from $S_d$ as compared to the shortest path distance from $S_s$ to $S_d$.

Our port group mechanism and allowed port bitmaps enable switches to efficiently route along non-minimal paths. The number of shortest and non-minimal path egress ports for a single destination prefix is not limited artificially, unlike n-way ECMP. The virtual port to physical port mapping can be used to control how traffic gets forwarded onto shortest paths and non-minimal paths. We now look in more detail at how to utilize port groups for non-minimal routing in HyperX networks.

### 4.1.1 *Strawman solution*

A straightforward approach for non-minimal routing is to create one port group for each destination prefix. For each prefix's port group, we make all appropriate physical ports (both along shortest paths and non-minimal paths to the destination) members of the port group. When a switch receives a packet, it looks up the port group for the destination prefix, and hashes the packet onto one of the virtual ports in the port group. Use of some of the corresponding egress ports results in non-minimal forwarding.

### 4.1.2 *Space saving techniques*

The strawman solution uses a separate port group for each destination prefix. As noted earlier, it is desirable to restrict the switch to just use shortest path forwarding if sufficient capacity is available. A fundamental characteristic of the HyperX topology is that in any source switch, the set of shortest path egress ports is different for each destination switch. So, these ports must be stored separately for each destination switch, thereby requiring a separate destination prefix and a separate port group (strawman) as well. For a large HyperX network, the corresponding switch memory overhead would be impractical. For example, a network with 2,048 128-port switches, and 1,024 virtual ports per port group would need 2MB of on-chip SRAM just to store the port group mapping (excluding counters). Thus, we seek to aggregate more prefixes to share port groups. We now describe some techniques to use just a small number of port groups to enable the use of non-minimal paths, while still using only shortest paths whenever possible.

In a HyperX switch, any egress port that is connected to another switch can be used to reach any destination. However not all egress ports would result in paths of equal length. Let us assume that we only use a single port group $PG_{all}$, say with 1,024 virtual ports. All physical ports in the switch connected to other switches are members of the port group $PG_{all}$. If we simply used this port group for all prefixes in the routing table, that would enable non-minimal forwarding for all destinations.

Dahu uses the allowed port bitmap hardware primitive to restrict forwarding to shorter paths when possible. If Dahu determines that only shortest paths need to be used for a particular destination prefix, the $B_{short}$ allowed port bitmap for the prefix is used for forwarding, even when the derouting count has not reached the maximum threshold. Otherwise, the allowed port bitmap is expanded to also include egress ports that would result in one extra hop being used and so on for longer paths. For HyperX, there are only three classes of egress port choices by distance to destination; in our basic non-minimal routing scheme, we either restrict forwarding to the shortest path ports or allow all paths to the destination (all member ports of $PG_{all}$).

We now look at the question of how Dahu determines when additional longer paths have to be enabled for a destination prefix to meed traffic demands. Switches already have port counters for the total traffic transmitted by each physical port. Periodically (e.g., every 10ms), the switch software reads all egress port counters, iterates over each destination prefix, and performs the steps shown in Figure 3 to enable non-minimal paths based on current utilization. It is straightforward to extend this to progressively enable paths of increasing lengths instead of all non-minimal paths at once.

```
1. Compute aggregate utilization (Agg) and capacity
   (Cap) for all egress ports in the $B_{short}$ bitmap
2. If Agg / Cap < Threshold,
       Set allowed ports to $B_{short}$ (there is sufficient
       capacity along shortest paths for this prefix)
   Else, set allowed ports to $B_{all}$.
```

**Figure 3:** Restricting non-minimal forwarding

In summary, we have a complete mechanism for forwarding traffic along shorter paths whenever possible, using just a single port group and enabling non-minimal routing whenever required for capacity reasons.

### 4.1.3 *Constrained non-minimal routing*

As described earlier, in a HyperX network, each switch has three classes of egress port choices to reach any destination. Based on this, Dahu defines a constrained routing scheme where we restrict routing as follows — any switch can forward a packet only to neighbors along offset dimensions. If a packet is allowed to use a non-minimal route at a switch, it can only be derouted along already offset dimensions. Once a particular dimension is aligned, we do not further deroute the packet along that dimension. We call this scheme *Dahu constrained routing*. For this technique, we create one port group for each possible set of dimensions in which the switch is offset from the destination switch. This uses $2^L$ port groups where $L$, the number of dimensions in a HyperX is usually small, e.g., 3–5. This allows migrating groups of flows between physical ports at an even smaller granularity than with a single port group.

This technique is largely inspired by Dimensionally Adaptive, Load balanced (DAL) routing [3]. However, there are some key differences. DAL uses per-packet load balancing, whereas we use flow level hashing to reduce TCP reordering effects. DAL allows at most one derouting in each offset dimension, but we allow any number of deroutings along offset dimensions as long as the derouting count threshold is not reached. This means that a switch $S_1$ can deroute traffic to a neighbor $S_2$ along an offset dimension when it does not have sufficient capacity to reach the destination along the shortest paths. In case of DAL, $S_2$ cannot further deroute along this dimension even if it does not have sufficient capacity along shortest paths.

## 4.2 Traffic Load Balancing

Per-packet uniform distribution of traffic across available paths from a source to destination can theoretically lead to very good network utilization in some symmetric topologies such as fat trees. But this is not used in practice due to the effects of packet reordering and faults on the transport protocol. ECMP tries to spread traffic uniformly across shortest length paths at the flow level instead. But due to its static nature, there can be local hash imbalances. Dahu presents a simple load balancing scheme using local information at each switch to spread traffic more uniformly.

Each Dahu switch performs load balancing with the objective of *balancing* or *equalizing* the aggregate load on each physical egress port. This also balances the bandwidth headroom on each switch port, so TCP flows can increase their rates. This simplifies our design, and enables us to avoid more complex approaches for demand estimation. When multiple egress ports are available for forwarding, we can remap virtual ports between physical ports, thus getting fine grained control over traffic across ports. Intuitively, in any port group, the number of virtual ports that map to any member port is a measure of the fraction of traffic from the port group that gets forwarded through that member port. We now describe the constraints and assumptions under which we load balance traffic at each switch in the network.

### *4.2.1 Design Considerations*

Periodically, each switch uses local information to rebalance traffic. This allows the switch to react quickly to changes in traffic demand and rebalance port groups more frequently than a centralized approach or one that requires information from peers. Note that this design decision is not fundamental–certainly virtual port mappings can be updated through other approaches. For different topologies, more advanced schemes may be required to achieve global optimality such as through centralized schemes.

We assume that each physical port might also have some traffic that is not *re-routable*. So Dahu's local load balancing scheme is limited to moving the remainder of traffic within port groups. Dahu's techniques can inter-operate with other traffic engineering approaches. For example, a centralized controller can make globally optimal decisions for placing elephant flows on globally efficient paths in the network [5], or higher layer adaptive schemes like MPTCP can direct more traffic onto uncongested paths. Dahu's heuristic corrects local hashing inefficiencies and can make quick local decisions within a few milliseconds to avoid temporary congestion. This can be complemented by a centralized or alternate approach that achieves global optimality over longer time scales of few hundreds of milliseconds.

### *4.2.2 Control Loop Overview*

Every Dahu switch periodically rebalances the aggregate traffic on its port groups once each epoch (e.g., every 10ms). At the end of each rebalancing epoch, the switch performs the following 3 step process:

*Step 1*: **Measure current load**: The switch collects the following local information from hardware counters: $(1a)$ for each port group, the amount of traffic that the port group sends to each of the member ports, and $(1b)$ for each egress port, the aggregate bandwidth used on the port.

*Step 2*: **Compute balanced allocation**: The switch computes a balanced traffic allocation for port groups, i.e. the amount of traffic each port group should send in a balanced setup to each of its member ports. We describe two ways of computing this in Sections 4.3 and 4.4.

*Step 3*: **Remap port groups**: The switch then determines which virtual ports in each port group must be remapped to other member ports in order to achieve a balanced traffic allocation, and changes the mapping accordingly. We have the current port group traffic matrix (measured) and the computed balanced traffic allocation matrix for each port group to it member egress ports.

As described in Section 3.1, a switch only maintains counters for the total amount of traffic from a port group to each of its member ports. We assume that all virtual ports that map to a particular member port are responsible for an equal share of traffic to the member port through that port group. We use the port group counters to compute the average traffic that each virtual port is responsible for. Then, we remap an appropriate number of virtual ports to other member ports depending on the intended traffic allocation matrix using a first-fit heuristic. In general, this remapping problem is similar to bin packing.

| PG \ Port | 0 | 1 | 2 | 3 |
|---|---|---|---|---|
| 0 | 4 | 1 | 2 | -- |
| 1 | -- | 1 | -- | 2 |
| BG | 2 | 2 | 2 | 0 |
| Agg | 6 | 4 | 4 | 2 |

Initial Port Group (PG)
Utilizations

| PG \ Port | 0 | 1 | 2 | 3 |
|---|---|---|---|---|
| 0 | 2 ⅔ | 1 ⅔ | 2 ⅔ | -- |
| 1 | -- | 1 | -- | 2 |
| BG | 2 | 2 | 2 | 0 |
| Agg | 4 ⅔ | 4 ⅔ | 4 ⅔ | 2 |

Step 1:
Balancing Port Group 0

| PG \ Port | 0 | 1 | 2 | 3 |
|---|---|---|---|---|
| 0 | 2 ⅔ | 1 ⅔ | 2 ⅔ | -- |
| 1 | -- | 0 | -- | 3 |
| BG | 2 | 2 | 2 | 0 |
| Agg | 4 ⅔ | 3 ⅔ | 4 ⅔ | 3 |

Step 2:
Balancing Port Group 1

| PG \ Port | 0 | 1 | 2 | 3 |
|---|---|---|---|---|
| 0 | 2 ⅓ | 2 ⅓ | 2 ⅓ | -- |
| 1 | -- | 0 | -- | 3 |
| BG | 2 | 2 | 2 | 0 |
| Agg | 4 ⅓ | 4 ⅓ | 4 ⅓ | 3 |

Step 3:
Balancing Port Group 0 (again)

**Figure 4:** Port group rebalancing algorithm. Egress ports that are not a member of the port group are indicated by '—'. The last row of the matrix represents the aggregate traffic (Agg) on the member ports (from port counters). The row indicating background traffic (BG) is added for clarity and is not directly measured by Dahu.

## 4.3 Load Balancing Algorithm

We now describe an algorithm for computing a balanced traffic allocation on egress ports. Based on the measured traffic, the switch generates a *port group traffic matrix* where the rows represent port groups and columns represent egress ports in the switch (see Figure 4). The elements in a row represent egress ports and the amount of traffic (bandwidth) that the port group is currently forwarding to those egress ports. If an egress port is not a member of the port group corresponding to the matrix row, then the respective matrix element is zeroed. Additionally, the *Aggregate utilization* row of elements stores the total bandwidth utilization on each egress port. This is the bandwidth based on the egress port counter, and accounts for traffic forwarded by any of the port groups, as well as background traffic on the port that is not re-routable using port groups, such as elephant flows pinned to the path by Hedera.

We first pick a port group in the matrix and try to balance the aggregate traffic for each of the member ports by moving traffic from this port group to different member ports where possible. To do this, Dahu computes the average aggregate utilization of all member ports of the port group. Then, it reassigns the traffic for that port group to equalize the aggregate traffic for the member ports to the extent possible. If a member port's aggregate traffic exceeds the average across all members, and the member port receives some traffic from this port group then we reassign the traffic to other member ports as appropriate. Dahu performs this operation for all port groups and repeats until convergence. To ensure convergence, we terminate the algorithm when subsequent iterations offload less than a small constant threshold $\delta$. Figure 4 shows the steps in the algorithm. Host facing ports in the switch can be ignored when executing this algorithm.

## 4.4 Load Balancing Heuristic

The load balancing algorithm considers all physical ports and port groups in the switch and aims to balance the aggregate load on all of them to the extent possible. However, the algorithm may take many steps to converge for a large switch with many ports and port groups. We now describe a quick and practical heuristic to compute the balanced traffic allocation. By running quickly, the switch can balance the port groups at time scales on the order of a few milliseconds.

1. Sort the physical ports by their aggregate utilization
2. Offload traffic from the highest loaded port H1 to the least loaded port with which it shares membership in any port group
3. Continue offloading traffic from H1 to the least loaded ports in order until they are completely balanced or H1 runs out of lesser loaded ports to offload to.

**Figure 5:** Load Balancing Heuristic

The heuristic performs the simple steps shown in Figure 5. This process is repeated for some fixed number $R$ (say 16) of the highest loaded ports in the switch. The heuristic has a low runtime of around 1ms[2]. In the rest of this paper, we employ this heuristic for load balancing.

## 4.5 Fault Tolerance

Dahu relies on link-level techniques for fault detection, and uses existing protocols to propagate fault updates through the network. If a particular egress link or physical port $P_f$ on the switch is down, the virtual ports in each port group which map to $P_f$ are remapped to other member ports of the respective port groups. The remapping is performed by switch software and the actual policy could be as simple as redistributing the virtual ports uniformly to other egress ports or something more complicated.

On the other hand, when the switch receives fault notifications from the rest of the network, a specific egress port $P_f$ may have to be disabled for only some destination prefixes because of downstream faults. We use the allowed port bitmaps technique described in Section 3.2 to just disable $P_f$ for specific prefixes. In both scenarios, the only flows migrated to other egress ports are ones that were earlier mapped

[2] The runtime depends on the number of physical ports and port groups in the switch and is independent of the number of flows in the system. Our research grade implementation of the heuristic for our simulator running on a general purpose x86 CPU showed average runtimes of few 10's of microseconds to 0.5 milliseconds, even for large networks with over 130,000 servers. We believe an optimized version targeted at a switch ARM or PPC processor can run within 1ms with a small DRAM requirement of under 10MB.

to the failed egress port $P_f$. When a physical port comes up, some virtual ports automatically get mapped to it the next time port groups are balanced.

## 5. DEPLOYABILITY

Deployability has been an important goal during the design of Dahu. In this section, we look at two primary requirements for adding Dahu support to switches: the logic to implement the functionality, and the memory requirements of the data structures.

To our knowledge, existing switch chips do not provide Dahu-like explicit hardware support for non-minimal routing in conjunction with dynamic traffic engineering. However, there are some similar efforts including Broadcom's resilient hashing feature [11] in their modern switch chips which is targeted at handling link failure and live topology updates, and the Group Table feature in the recent OpenFlow 1.1 Specification [28] which uses a layer of indirection in the switch datapath for multipath support. The increasing popularity of OpenFlow [27], software defined networks [24], and custom computing in the control plane (via embedded ARM style processors in modern switch silicon) indicates a new trend that we can leverage where large data centers operators are adopting the idea of a programmable control plane for the switches. The need for switch hardware modification to support customizable control plane for switches is no longer a barrier to innovation, as indicated by the deployment of switches with custom hardware by companies like Google [20].

To implement the hardware logic, we also need sufficient memory in the chip to support the state requirements for Dahu functionality. We now briefly estimate this overhead. Consider a large Dahu switch with 128 physical ports, 64 port groups with 1,024 virtual ports each, 16,384 prefixes in the routing table, and support for up to two different allowed port bitmaps for each prefix. The extra state required for all of Dahu's features is a modest 640KB. Of this, 64KB each are required for storing the virtual to physical port mappings for all the port groups, and the port group counters per egress port. 512KB is required for storing two bitmaps for each destination prefix. A smaller 64 port switch would only need a total of 352KB for a similar number of port groups and virtual ports. This memory may come at the expense of additional packet buffers (typically around 10MB); however, recent trends in data center congestion management [7, 6] indicate that trading a small amount of buffer memory for more adaptive routing may be worthwhile.

## 6. EVALUATION

We evaluated Dahu through flow-level simulations on both HyperX and Fat-tree topologies. Overall, our results show:

1. 10-50% throughput improvement in Fat-tree networks, and 250-500% improvement in HyperX networks compared to ECMP.
2. With an increase of only a single network hop, Dahu achieves significant improvements in throughput.
3. Dahu scales to large networks of over 130,000 nodes.
4. Dahu can complement MPTCP by achieving higher throughput with fewer subflows.

The evaluation seeks to provide an understanding of Dahu's effect on throughput and hop count on different network topologies under different traffic patterns. We start by looking at HyperX networks, large and small, and measure throughput as well as expected hop count for different workloads. We them move on to evaluate Dahu on an 8,192 host Fat-tree network using two communication patterns. We conclude the evaluation section by presenting the methodology we use to validate the accuracy of our simulator.

### 6.1 Simulator

We evaluated Dahu using a flow level network simulator that models the performance of TCP flows. We used the flow level simulator from Hedera [5], and added support for decentralized routing in each switch, port groups, allowed port bitmaps, and the load balancing heuristic. The Dahu-augmented Hedera simulator evaluates the AIMD behavior of TCP flow bandwidths to calculate the total throughput achieved by flows in the network.

In addition, we also built a workload generator, that generates open-loop input traffic profiles for the simulator. It creates traffic profiles with different distributions of flow inter-arrival times and flow sizes. This allows us to evaluate Dahu's performance over a wide range of traffic patterns, including those based on existing literature [8, 18]. Modeling the AIMD behavior of TCP flow bandwidth instead of per-packet behavior means that the simulator does not model TCP timeouts, retransmits, switch buffer occupancies and queuing delay in the network. We chose to make these trade-offs in the simulator to evaluate at a large scale–over 130K servers, which would not have been possible otherwise. We simulated five seconds of traffic in each experiment, and each switch rebalanced port groups (16 highest loaded ports) and recomputed prefix bitmaps every 10ms.

### 6.2 HyperX Networks

We first evaluate Dahu with HyperX networks which typically have many paths of differing lengths between a source and a destination. This exercises Dahu's non-shortest path selection algorithm as well as its load balancing capability. We simulate a (L=3, S=14, T=48) HyperX network with 1Gbps links, taken from [3]; where L = number of dimensions, S = number of switches in each dimension, T = number of servers connected to each switch. This models a large data center with 131,712 servers, 2,744 switches and an oversubscription of 1:8.

We seek to measure how the use of non-minimal paths and dynamic load balancing affects performance as we vary the allowed derouting threshold and non-minimal routing scheme (constrained or not) and different traffic patterns. For evaluation, we run simulations for Clique and Mixed traffic patterns (described next), and compare the throughput, average hop count and edge utilizations for Dahu and ECMP. In the graphs, Dahu-n refers to Dahu routing with at most $n$ deroutings. C-Dahu-n refers to the similar Constrained Dahu variant.

### 6.2.1 Clique Traffic Pattern

A *Clique* is a subset of switches and associated hosts that communicate among themselves, where each host communicates with every other host in its clique over time. This represents distributed jobs in a data center which are usually run on a subset of the server pool. Often a job runs on a few racks of servers. There could be multiple such cliques (or jobs) running in different parts of the network. We parameterize this traffic pattern by i) clique size, the number of switches in the clique and ii) the total number of cliques in the network. In this experiment, we vary the total number of cliques from 64 to 768, keeping the clique size fixed at 2 switches (96 servers). Each source switch in a clique generates 18Gbps of traffic with 1.5 MB average flow size.

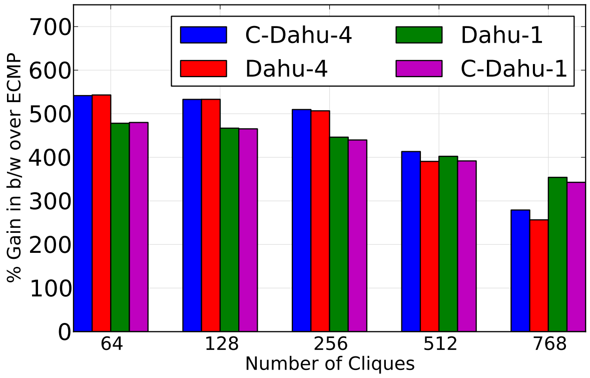
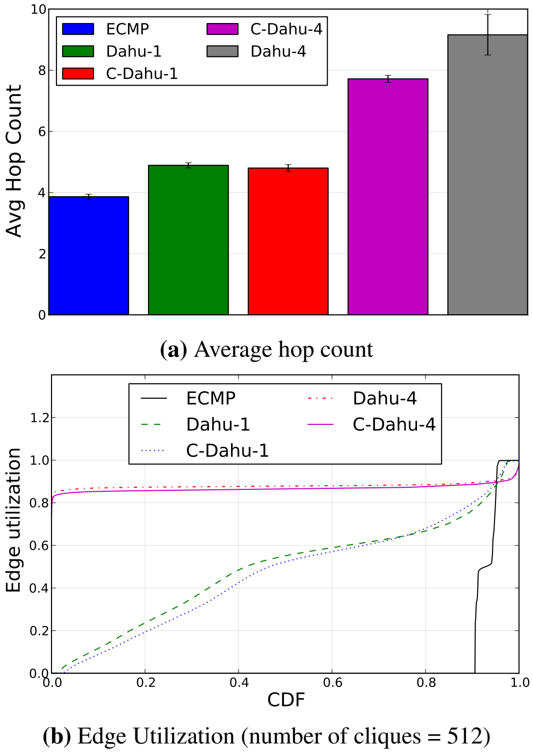


**Figure 6:** Throughput gain with Clique Traffic Pattern.

Figure 6 shows the bandwidth gains with Dahu relative to ECMP as we vary the number of communicating cliques. Dahu offers substantial gains of 400-500% over ECMP. The performance gain is highest with a smaller number of cliques, showing that indeed derouting and non-shortest path selection can effectively take advantage of excess bandwidth in HyperX networks. This validates a major goal of this work, which is improving the statistical multiplexing of bandwidth in complex network topologies. As the number of cliques increases, the bandwidth slack in the network decreases, and the relative benefit of non-minimal routing comes down to around 250%. Dahu and constrained Dahu have similar performance for the same derouting threshold.

We further find that a large derouting threshold provides larger benefit with less load, since there are many unused links in the network. As load increases, links are more utilized on average, bandwidth slack reduces, and a derouting threshold of one starts performing better.



**(a)** Average hop count

**(b)** Edge Utilization (number of cliques = 512)

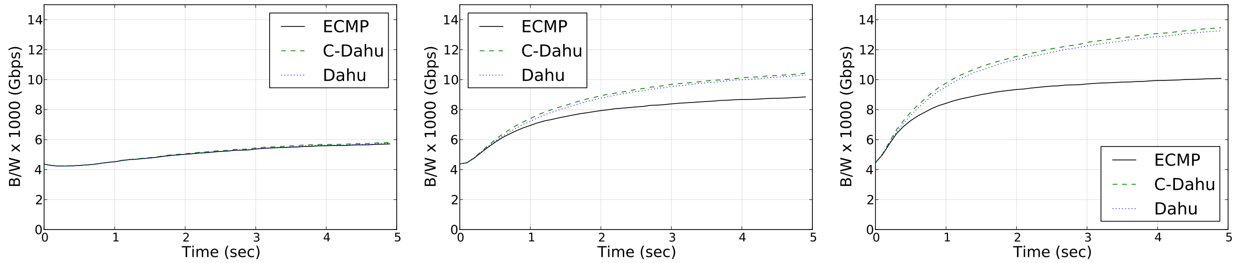
**Figure 7:** Edge utilization & Average hop count for Clique traffic pattern

Beyond raw throughput, latency in an important performance metric that is related to network hop count. Figure 7a shows the average hop count observed for each routing scheme. Dahu is able to deliver significantly higher bandwidth with a small increase in average hop count. The relationship between increased throughput and increased hop count can be tuned using the derouting threshold mechanism. For large derouting thresholds, the average hop count likewise increases. However, for smaller derouting threshold, the hop count is similar to that of ECMP while still achieving most of the bandwidth improvements of non-minimal routing. Note that the small error bars indicate that the average hop count is similar while varying the number of cliques.

Figure 7b shows the CDF of inter-switch edge utilizations for ECMP and Dahu. With shortest path routing, 90% of the edges have zero utilization whereas, Dahu achieves its bandwidth gains by utilizing available capacity on additional edges in the network. Also, we see that a single derouting can achieve most of the overall bandwidth gains while consuming bandwidth on significantly fewer links in the network thereby sparing network capacity for more traffic.

### 6.2.2 Mixed Traffic Pattern

The "mixed traffic pattern" represents an environment with a few *hot racks* that send lot of traffic, representing jobs like data backup. For this traffic pattern, we simulate 50 cliques with 10 switches in each. Every switch acts as a network hot-spot and has flows to other members of the clique with



**(a)** Load = 4.5Gbps/switch **(b)** Load = 17.5Gbps/switch **(c)** Load = 33.5Gbps/switch

**Figure 8:** Dahu with Mixed Traffic Pattern

average size = 100MB. The load from each source switch is varied from 3Gbps to 32Gbps. In addition to these hotspots, we also generate random all-to-all traffic between all the hosts in the network. This background traffic creates an additional load of 1.5Gbps per source switch with average flow size of 200KB. Figure 8 shows that for low load levels (4.5Gbps total load per switch) ECMP paths are sufficient to fulfill demand. As expected, total bisection bandwidth achieved is same for both ECMP and Dahu. However, at high load (17.5Gbps and 33.5Gbps per switch) Dahu performs significantly better than ECMP by utilizing available slack bandwidth.

## 6.3 Fat-Tree Networks

We next evaluate Dahu in the context of a Fat-tree topology. In Fat-trees, unlike HyperX, there are a large number of shortest paths between a source and destination, so the following evaluation focuses on Dahu's load balancing behavior, rather than its use of non-shortest paths. We compare Dahu with ECMP, with hosts communicating over long lived flows in a k = 32 Fat-tree (8,192 hosts). We consider these traffic patterns: (1) *Stride*: With *n* total hosts and stride length *x*, each host *i* sends traffic to host *(i + x) mod n*. (2) *Random*: Each host communicates with another randomly chosen destination host in the network. To study the effect of varying overall network load, we pick a subset of edge switches that send traffic to others and vary the number of hosts on each of these edge switches that originate traffic.

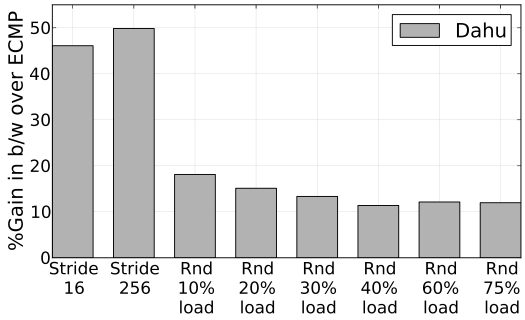


**Figure 9:** Throughput gain for k = 32 Fat-tree with Stride and Random (Rnd) traffic patterns.

Figure 9 shows that Dahu achieves close to 50% improvement with stride traffic patterns. The load balancing heuristic rebalances virtual port mappings at each switch minimizing local hash imbalances and improves total throughput. For random traffic patterns, Dahu outperforms ECMP by 10-20%. Overall, Dahu is better able to utilize network links in Fat-tree networks than ECMP, even when only shortest-path links are used.

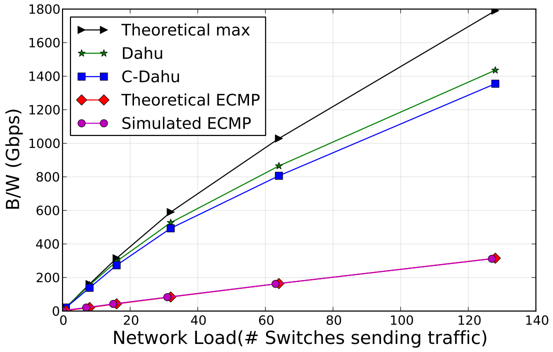


**Figure 10:** Simulator throughput vs. theoretical maximum

## 6.4 Simulator Validation

To validate the throughput reported by the simulator, we generated a traffic profile with a large number of long-lived flows between random hosts for a (L=3, S=8, T=48) HyperX network. We computed the theoretical maximum bandwidth achievable for the traffic pattern by formulating a maximum multi-commodity network flow problem and solving it with the CPLEX [12] linear program solver. We also ran our simulator on the same traffic profile. As shown in Figure 10 the aggregate throughput reported by the simulator was within the theoretical maximum for all the traffic patterns that we validated. In case of shortest path forwarding, the theoretical and simulator numbers matched almost perfectly indicating that the ECMP implementation was valid. With non-minimal forwarding, the simulator's performance is reasonably close to the theoretical limit. Note that the multi-commodity flow problem simply optimizes for the total network utilization whereas the simulator and TCP in general, also take fairness into account.

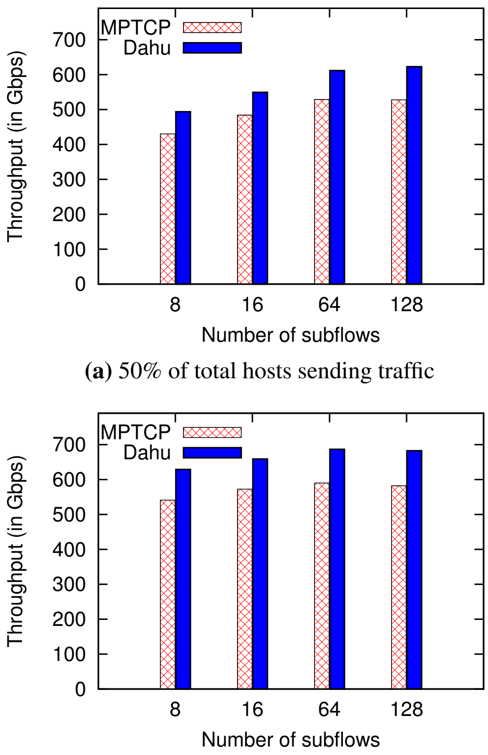
In addition, for these traffic profiles we also explicitly computed the flow bandwidth distribution that ensures max-min fairness using the water-filling algorithm [10]. We compared the resulting aggregate throughput to those reported by the simulator. For all evaluated traffic patterns, the simulator throughput was within 10% of those reported by the max-min validator. This small difference is because TCP's AIMD congestion control mechanism only yields approximate max-min fairness in flow bandwidths whereas the validator computes a perfectly max-min fair distribution.

### 6.5 MPTCP in HyperX Networks

We conclude our evaluation by studying how the host based MPTCP approach would perform in HyperX networks if shortest as well as non-shortest paths were exposed to MPTCP for forwarding subflows. In particular, we wanted to evaluate how well MPTCP could probe the network paths and utilize the capacity with long lived flows. We used *htsim* [13], a packet-level simulator that models TCP and MPTCP performance and extended it to support HyperX networks. We simulated a (L=3, S=10, T=20) HyperX network with 100Mbps links. This network has 1000 switches, 20,000 hosts and an oversubscription of 1:4. We chose this smaller topology and lower link speed due to the higher computational overhead of packet level simulations. We enabled routing of subflows over shortest paths and non-shortest paths with one allowed deroute. We evaluated MPTCP's performance by varying the load and the number of MPTCP subflows per long lived flow as shown in Figure 11. To generate traffic, we first created a random permutation matrix (without replacement), and then selected a subset of source-destination pairs from the matrix that send data over long lived flows.

We also used the Dahu simulator on the same network topology and traffic pattern to evaluate Dahu's performance by treating each subflow as an independent TCP flow. Since we were using packet level simulations for MPTCP and flow level simulations for Dahu, we wanted to validate that their results would be comparable for identical scenarios. So in order to calibrate the two simulators, we ran an experiment with varying number of long lived flows between a pair of hosts on two different switches in the topology with htsim (with 1 subflow each), and also with the Dahu simulator. We allowed only ECMP routing on the same set of paths in both simulators. We obtained similar throughput results from both, thus validating that the packet level htsim, and flow level Dahu simulators generate comparable numbers for the same scenarios.

From Figure 11a we make two observations. First, when there are fewer hosts sending traffic, the network throughput for both Dahu and MPTCP increases with more subflows. This is because more flows can be routed over the large number of paths present in the HyperX topology. As such, in order for MPTCP to effectively utilize the bandwidth on all the paths, it needs to create a large number of subflows that can probe these paths for capacity (e.g., in Figure 11a achieved throughput is maximum with 64 subflows). Second, while Dahu's performance also improves with more flows, it is able to achieve the same throughput with 16 subflows that MPTCP achieves with 64 or 128 subflows.



**(a)** 50% of total hosts sending traffic

**(b)** 90% of total hosts sending traffic

**Figure 11:** MPTCP and Dahu performance for L=3, S=10, T=20 HyperX topology. Results obtained from packet level simulations for MPTCP and flow level simulations for Dahu.

When more hosts send traffic (Figure 11b), throughput does not improve much with more subflows because there is less unused capacity in the network that could have been utilized by additional subflows. Again, Dahu is able to achieve the same throughput as MPTCP with fewer subflows.

These experiments indicate that MPTCP needs many subflows to fully leverage the capacity of such a network with high path diversity. This makes it mainly useful for long lived flows. On the other hand Dahu can address short lived flows by rebalancing aggregates of short flows through hashing. As a transport layer protocol, MPTCP also has no way of distinguishing shortest paths from non-shortest paths for forwarding its subflows. Dahu can complement MPTCP by choosing the right set of paths for routing, allowing it to achieve higher throughput for long lived flows with fewer subflows, while shorter flows are handled by Dahu.

## 7. DISCUSSION

As seen in Section 6, Dahu exploits non-minimal routing to derive large benefits over ECMP for different topologies and varying communication patterns. Yet, there is a scenario where using non-minimal routing can be detrimental. This occurs when the network as a whole is pretty highly saturated; ECMP style shortest path forwarding itself does well as most links have sufficient traffic and there is no “unused” capacity or slack in the network. With Dahu, a derouted flow consumes bandwidth on more links than if had used just shortest paths thereby contributing to congestion on more links. In large data centers, this network saturation scenario is uncommon. Networks are usually run at much lower average utilizations although there may be hot-spots or small cliques of racks with lot of communication between them. Usually, there is network slack or capacity in the rest of the network which Dahu can leverage. The network saturation case can be dealt with in many ways. For example, a centralized monitoring infrastructure can check if a large fraction of the network is in its saturation regime and notify switches to stop using non-minimal paths.

For Dahu, we propose a simple refinement to the localized load balancing scheme that uses congestion feedback from neighboring switches to fall back to shortest path forwarding in such high load scenarios especially for smaller networks. We describe this for the HyperX topology but a similar feedback approach can be used for avoiding downstream hash collisions in case of Fat-trees. We propose overriding the IP TOS field of packets to denote a 1 bit route type. This is updated by each switch along the path of the packet to indicate whether it used a shortest path egress port or derouted the packet. When a switch $S_1$ receives a packet it checks if the previous hop switch $S_0$ derouted the packet. In that case, if $S_1$ also does not have enough capacity to the destination using shortest paths alone, then it creates a control packet to notify $S_0$ to stop sending derouted traffic for the particular destination prefix to $S_1$ for a certain duration of time (say 5ms). When $S_0$ receives such a congestion feedback packet, it modifies its allowed port bitmap for the prefix to disallow the corresponding port for some time.

This mechanism is in many ways similar to Ethernet pause frames [1], however it only disables paths for specific prefixes. The technique can be easily extended to incorporate more classes of traffic or routes by using different allowed port bitmaps for different QoS classes. This kind of an extension would be comparable to priority flow control (PFC) [2].

# 8. RELATED WORK

There have been several recent proposals for scale-out multipath data center topologies such as Clos networks [4, 18, 25, 26], direct connect networks like HyperX [3], Flattened Butterfly [22], DragonFly [23], and even randomly connected network fabrics proposed in Jellyfish [30] in order to deliver high bandwidth and fault tolerance. Such topologies have motivated the need for generic mechanisms that can leverage the high path diversity. Many current proposals use ECMP-based techniques which are inadequate to utilize all paths or to dynamically load balance the traffic. Moreover, the routing proposals for these topologies are limited to shortest path routing (or K-shortest path routing with Jellyfish) and end up under-utilizing the network, more so in the presence of failures. While DAL routing allows derouting, it is limited to HyperX topologies. In contrast, Dahu proposes a topology-independent solution for non-minimal routing that eliminates routing loops, routes around failures and achieves high network utilization.

As discussed in Section 2, Hedera [5] and MicroTE [9] propose a centralized controller to schedule long lived flows on globally optimal paths. However they operate on longer time scales and scaling them to large networks with lots of flows is challenging. While DevoFlow [14] improves Hedera’s scalability by proposing switch hardware modifications, forwarding rule cloning and triggered counters, it still does not support dynamic hashing or non-shortest path routing. Dahu can co-exist with such techniques to better handle congestion at finer time scales and gracefully offload shorter flows on to less loaded ports.

Recently, DeTail [33] proposed switch modifications to reduce flow completion time tail. Unlike Dahu, DeTail only leverages shortest paths and performs per-packet adaptive load balancing which requires end-host modifications to handle TCP reordering. There have been proposals that employ variants of switch-local per-packet traffic splitting [15], but again require end-host modifications for TCP. With Dahu, instead of per-packet splitting, we locally rebalance flow aggregates across different paths thereby largely reducing in-network packet reordering.

MPTCP [29] proposes a host based approach for dividing a single TCP flow into subflows. However, as a transport protocol, it does not have control over the network paths taken by subflows and does not distinguish between shortest and non-shortest paths. Dahu can complement MPTCP by selecting a good subset of paths over which the subflows can be routed.

Traffic engineering has been well studied in the context of wide area networks. TeXCP [21] and MATE [16] repeatedly probe the network and split flows on different paths based on load. However, low latency and faster response time requirements along with short flow sizes make these solutions inapplicable in a data center context. FLARE [31] exploits the inherent burstiness in TCP flows to schedule “flowlets” (bursts of packets) on different paths to reduce extensive packet reordering. REPLEX [17] relies on game-theoretic analysis along with periodic exchanges between neighbouring switches about path utilization to make switch-local decisions for splitting traffic on different paths. However, the convergence time for the game theoretic analysis is too long for it to benefit the short flows in data centers.

Finally, a key distinction between Dahu and the related traffic engineering and load balancing approaches is that Dahu actively routes over non-shortest paths in order to satisfy traffic demand. Dahu decouples non-minimal routing and its mechanism for more balanced hashing and offers a more flexible architecture for better network utilization.

## 9. CONCLUSION

We present a new switch mechanism, Dahu, that enables dynamic hashing of traffic onto different network paths. Dahu proposes switch hardware primitives and control software to efficiently exploit non-shortest paths in the network and reduce congestion while preventing persistent forwarding loops. We present a decentralized load balancing heuristic that makes quick, local decisions to mitigate congestion, and show the feasibility of proposed switch hardware modifications. We evaluate Dahu using a simulator for different topologies and different traffic patterns and show that it significantly outperforms shortest path routing. Finally, we evaluate MPTCP in HyperX networks and show that Dahu can complement MPTCP by selecting good paths for subflows, in addition to efficiently routing short lived flows.

## 10. REFERENCES

[1] Ethernet Flow Control, IEEE 802.3x.

[2] Priority-based Flow Control, IEEE 802.1Qbb.

[3] 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*, 2009.

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

[5] 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*, 2010.

[6] M. Alizadeh, A. Greenberg, D. A. Maltz, J. Padhye, P. Patel, B. Prabhakar, S. Sengupta, and M. Sridharan. Data center TCP (DCTCP). In *Proc. of ACM SIGCOMM*, 2010.

[7] M. Alizadeh, A. Kabbani, T. Edsall, B. Prabhakar, A. Vahdat, and M. Yasuda. Less Is More: Trading a Little Bandwidth for Ultra-Low Latency in the Data Center. In *Proc. of Usenix NSDI*, 2012.

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

[9] T. Benson, A. Anand, A. Akella, and M. Zhang. MicroTE: Fine Grained Traffic Engineering for Data Centers. In *Proc. of ACM CoNEXT*, 2011.

[10] D. Bertsekas and R. Gallager. *Data Networks*. Prentice-Hall, 1987.

[11] Broadcom Smart-Hash Technology. `http://www.broadcom.com/collateral/wp/StrataXGS_SmartSwitch-WP200-R.pdf`.

[12] CPLEX Linear Program Solver. `http://www-01.ibm.com/software/integration/optimization/cplex-optimizer/`.

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

[14] 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*, 2011.

[15] A. A. Dixit, P. Prakash, R. R. Kompella, and C. Hu. On the Efficacy of Fine-Grained Traffic Splitting Protocols in Data Center Networks. Technical Report Purdue/CSD-TR 11-011, 2011.

[16] A. Elwalid, C. Jin, S. Low, and I. Widjaja. MATE: MPLS Adaptive Traffic Engineering. In *Proc. of IEEE INFOCOM*, 2001.

[17] S. Fischer, N. Kammenhuber, and A. Feldmann. REPLEX: Dynamic Traffic Engineering Based on Wardrop Routing Policies. In *Proc. of ACM CoNEXT*, 2006.

[18] 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*, 2009.

[19] 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*, 2010.

[20] U. Hölzle. OpenFlow @ Google. Talk at Open Networking Summit, 2012.

[21] S. Kandula, D. Katabi, B. Davie, and A. Charny. Walking the Tightrope: Responsive Yet Stable Traffic Engineering. In *Proc. of ACM SIGCOMM*, 2005.

[22] J. Kim, W. J. Dally, and D. Abts. Flattened butterfly: A Cost-efficient Topology for High-radix networks. In *Proc. of ISCA*, 2007.

[23] J. Kim, W. J. Dally, S. Scott, and D. Abts. Technology-Driven, Highly-Scalable Dragonfly Topology. In *Proc. of ISCA*, 2008.

[24] 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*, 2010.

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

[26] R. N. Mysore, A. Pamboris, N. Farrington, N. Huang, P. Miri, S. Radhakrishnan, V. Subramanya, and A. Vahdat. PortLand: A Scalable Fault-Tolerant Layer 2 Data Center Network Fabric. In *Proc. of ACM SIGCOMM*, 2009.

[27] OpenFlow Consortium. `http://www.openflow.org`.

[28] OpenFlow Switch Specification - Version 1.1. `http://www.openflow.org/documents/openflow-spec-v1.1.0.pdf`.

[29] 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*, 2011.

[30] A. Singla, C.-Y. Hong, L. Popa, and P. B. Godfrey. Jellyfish: Networking Data Centers Randomly. In *NSDI*, 2012.

[31] S. Sinha, S. Kandula, and D. Katabi. Harnessing TCPs Burstiness using Flowlet Switching. In *HotNets*, 2004.

[32] A. Vahdat, M. Al-Fares, N. Farrington, R. N. Mysore, G. Porter, and S. Radhakrishnan. Scale-Out Networking in the Data Center. *IEEE Micro*, 2010.

[33] D. Zats, T. Das, P. Mohan, D. Borthakur, and R. Katz. DeTail: Reducing the Flow Completion Time Tail in Datacenter Networks. In *Proc. of ACM SIGCOMM*, 2012.