HULA: Scalable Load Balancing Using Programmable Data Planes

Naga Katta*, Mukesh Hira†, Changhoon Kim‡, Anirudh Sivaraman+, Jennifer Rexford*
*Princeton University, †VMware, ‡Barefoot Networks, +MIT CSAIL
{nkatta, jrex}@cs.princeton.edu, mhira@vmware.com, chang@barefootnetworks.com, anirudh@csail.mit.edu

ABSTRACT

Datacenter networks employ multi-rooted topologies (e.g., Leaf-Spine, Fat-Tree) to provide large bisection bandwidth. These topologies use a large degree of multipathing, and need a data-plane load-balancing mechanism to effectively utilize their bisection bandwidth. The canonical load-balancing mechanism is equal-cost multi-path routing (ECMP), which spreads traffic uniformly across multiple paths. Motivated by ECMP's shortcomings, congestion-aware load-balancing techniques such as CONGA have been developed. These techniques have two limitations. First, because switch memory is limited, they can only maintain a small amount of congestion-tracking state at the edge switches, and do not scale to large topologies. Second, because they are implemented in custom hardware, they cannot be modified in the field.

This paper presents HULA, a data-plane load-balancing algorithm that overcomes both limitations. First, instead of having the leaf switches track congestion on *all paths* to a destination, each HULA switch tracks congestion for the *best path* to a destination through a neighboring switch . Second, we design HULA for emerging programmable switches and program it in P4 to demonstrate that HULA could be run on such programmable chipsets, without requiring custom hardware. We evaluate HULA extensively in simulation, showing that it outperforms a scalable extension to CONGA in average flow completion time (1.6× at 50% load, 3× at 90% load).

CCS Concepts

•Networks → Programmable networks;

Keywords

In-Network Load Balancing; Programmable Switches; Network Congestion; Scalability.

SOSR'16, March 14–15, 2016, Santa Clara, CA, USA

DOI: http://dx.doi.org/10.1145/2890955.2890968

1 Introduction

Data-center networks today have multi-rooted topologies (Fat-Tree, Leaf-Spine) to provide large bisection bandwidth. These topologies are characterized by a large degree of multipathing, where there are several routes between any two endpoints. Effectively balancing traffic load across multiple paths in the data plane is critical to fully utilizing the available bisection bandwith. Load balancing also provides the abstraction of a single large output-queued switch for the entire network [1–3], which in turn simplifies bandwidth allocation across tenants [4, 5], flows [6], or groups of flows [7].

The most commonly used data-plane load-balancing technique is equal-cost multi-path routing (ECMP), which spreads traffic by assigning each flow to one of several paths at random. However, ECMP suffers from degraded performance [8–12] if two long-running flows are assigned to the same path. ECMP also doesn't react well to link failures and leaves the network underutilized or congested in asymmetric topologies. CONGA [13] is a recent data-plane load-balancing technique that overcomes ECMP's limitations by using link utilization information to balance load across paths. Unlike prior work such as Hedera [8], SWAN [14], and B4 [15], which use a central controller to balance load every few minutes, CONGA is more responsive because it operates in the data plane, permitting it to make load-balancing decisions every few microseconds.

This responsiveness, however, comes at a significant implementation cost. First, CONGA is implemented in custom silicon on a switching chip, requiring several months of hardware design and verification effort. Consequently, once implemented, the CONGA algorithm cannot be modified. Second, memory on a switching chip is at a premium, implying that CONGA's technique of maintaining per-path congestion state at the leaf switches limits its usage to topologies with a small number of paths. This hampers CONGA's scalability and as such, it is designed only for two-tier Leaf-Spine topologies.

This paper presents HULA (Hop-by-hop Utilization-aware Load balancing Architecture), a data-plane load-balancing algorithm that addresses both issues.

First, HULA is more scalable relative to CONGA in two ways. One, each HULA switch only picks the next hop, in contrast to CONGA's leaf switches that determine the entire path, obviating the need to maintain forwarding state for a large number of tunnels (one for each path). Two, because HULA switches only choose the best next hop along what is globally the instantaneous best path to a destination, HULA switches only need to maintain congestion state for the best next hop per destination, not all paths to a destination.

Second, HULA is specifically designed for a programmable switch architecture such as the RMT [16], FlexPipe [17], or XPliant [18] architectures. To illustrate this, we prototype HULA in the recently proposed P4 language [19] that explicitly targets such programmable data planes. This allows the HULA algorithm to be inspected and modified as desired by the network operator, without the rigidity of a silicon implementation.

Concretely, HULA uses special probes (separate from the data packets) to gather global link utilization information. These probes travel periodically throughout the network and cover all desired paths for load balancing. This information is summarized and stored at each switch as a table that gives the best next hop towards any destination. Subsequently, each switch updates the HULA probe with its view of the best downstream path (where the best path is the one that minimizes the maximum utilization of all links along a path) and sends it to other upstream switches. This leads to the dissemination of best path information in the entire network similar to a distance vector protocol. In order to avoid packet reordering, HULA load balances at the granularity of *flowlets* [11]— bursts of packets separated by a significant time interval.

To compare HULA with other load-balancing algorithms, we implemented HULA in the network simulator ns-2 [20]. We find that HULA is effective in reducing switch state and in obtaining better flow-completion times compared to alternative schemes on a 3-tier topology. We also introduce asymmetry by bringing down one of the core links and study how HULA adapts to these changes. Our experiments show that HULA performs better than comparative schemes in both symmetric and asymmetric topologies.

In summary, we make the following two key contributions.

- We propose HULA, a scalable data-plane load-balancing scheme. To our knowledge, HULA is the first load balancing scheme to be explicitly designed for a programmable switch data plane.
- We implement HULA in the ns-2 packet-level simulator and evaluate it on a Fat-Tree topology [21] to show that it delivers between 1.6 to 3.3 times better flow completion times than state-of-the-art congestion-aware load balancing schemes at high network load.

2 Design Challenges for HULA

Large datacenter networks [22] are designed as multi-tier Fat-Tree topologies. These topologies typically consist of 2-tier Leaf-Spine pods connected by additional tiers of spines. These additional layers connecting the pods can be arbitrarily deep depending on the datacenter bandwidth capacity needed. Load balancing in such large datacenter topologies poses scalability challenges because the explosion of the number of paths between any pair of Top of Rack switches (ToRs) causes three important challenges.

Large path utilization matrix: Table 1 shows the number of paths between any pair of ToRs as the radix of a Fat-Tree topology increases. If a sender ToR needs to track link utilization on all desired paths[1] to a destination ToR in a Fat-Tree topology with radix k, then it needs to track k^2 paths for each destination ToR. If there are m such leaf ToRs, then it needs to keep track of $m * k^2$ entries , which can be prohibitively large. For example, CONGA [13] maintains around 48K bits of memory (512 ToRs, 16 uplinks, and 3 bits for utilization) to store the path-utilization matrix. In a topology with 10K ToRs and with 10K paths between each pair, the ASIC would require 600M bits of memory, which is prohibitively expensive (by comparison the packet data buffer of a shallow-buffered switch such as the Broadcom Trident [23] is 96 Mbits). For the ASIC to be viable and scale with large topologies, it is imperative to reduce the amount of congestion-tracking state stored in any switch.

[1] A path's utilization is the maximum utilization across all its links.

Topology	# Paths between pair of ToRs	# Max forwarding entries per switch
Fat-Tree (8)	16	944
Fat-Tree (16)	64	15,808
Fat-Tree (32)	256	257,792
Fat-Tree (64)	1024	4,160,512

Table 1: Number of paths and forwarding entries in 3-tier Fat-Tree topologies [24]

Large forwarding state: In addition to maintaining per-path utilization at each ToR, existing approaches also need to maintain large forwarding tables in *each* switch to support a leaf-to-leaf tunnel for each path that it needs to route packets over. In particular, a Fat-Tree topology with radix 64 supports a total of 70K ToRs and requires 4 million entries [24] per switch as shown in Table 1. The situation is equally bad [24] in other topologies like VL2 [25] and BCube [26]. To remedy this, recent techniques like Xpath [24] have been designed to reduce the number of entries using compression techniques that exploit symmetry in the network. However, since these techniques rely on the control plane to update and compress the forwarding entries, they are slow to react to failures and topology asymmetry, which are common in large topologies.

Discovering uncongested paths: If the number of paths is large, when new flows enter, it takes time for reactive load balancing schemes to discover an uncongested path especially when the network utilization is high. This increases the flow completion times of short flows because these flows finish before the load balancer can find an uncongested path. Thus, it is useful to have the utilization information conveyed to the sender in a proactive manner, before a short flow even commences.

Programmability: In addition to these challenges, implementing data-plane load-balancing schemes in hardware can be a tedious process that involves significant design and verification effort. The end product is a one-size-fits-all piece of hardware that network operators have to deploy without the ability to modify the load balancer. The operator has to wait for the next product cycle (which can be a few years) if she wants a modification or an additional feature in the load balancer. An example of such a modification is to load balance based on queue occupancy as in backpressure routing [27, 28] as opposed to link utilization.

The recent rise of programmable packet-processing pipelines [16, 17] provides an opportunity to rethink this design process. These data-plane architectures can be configured through a common programming language like P4 [19], which allow operators to program stateful data-plane packet processing at line rate. Once a load balancing scheme is written in P4, the operator can modify the program so that it fits her deployment scenario and then compile it to the underlying hardware. In the context of programmable data planes, the load-balancing scheme must be simple enough so that it can be compiled to the instruction set provided by a specific programmable switch.

3 HULA Overview: Scalable, Proactive, Adaptive, and Programmable

HULA combines distributed network routing with congestion-aware load balancing thus making it tunnel-free, scalable, and adaptive.

Similar to how traditional distance-vector routing uses periodic messages between routers to update their routing tables, HULA uses periodic probes that proactively update the network switches with the best path to any given leaf ToR. However, these probes are processed at line rate entirely in the data plane unlike how routers process control packets. This is done frequently enough to reflect the instantaneous global congestion in the network so that the switches make timely and effective forwarding decisions for volatile datacenter traffic. Also, unlike traditional routing, to achieve fine-grained load balancing, switches split flows into *flowlets* [11] whenever an inter-packet gap of an RTT (network round trip time) is seen within a flow. This minimizes receive-side packet-reordering when a HULA switch sends different flowlets on different paths that were deemed best at the time of their arrival respectively. HULA's basic mechanism of probe-informed forwarding and flowlet switching enables several desirable features, which we list below.

Maintaining compact path utilization: Instead of maintaining path utilization for all paths to a destination ToR, a HULA switch only maintains a table that maps the destination ToR to the best next hop as measured by path utilization. Upon receiving multiple probes coming from different paths to a destination ToR, a switch picks the hop that saw the probe with the minimum path utilization. Subsequently it sends its view of the best path to a ToR to its neighbors. Thus, even if there are multiple paths to a ToR, HULA does not need to maintain per-path utilization information for each ToR. This reduces the utilization state on any switch to the order of the number of ToRs (as opposed to the number of ToRs times the number of paths to these ToRs from the switch), effectively removing the pressure of path explosion on switch memory. Thus, HULA distributes the necessary *global* congestion information to enable scalable *local* routing.

Scalable and adaptive routing: HULA's best hop table eliminates the need for separate source routing in order to exploit multiple network paths. This is because in HULA, unlike other source-routing schemes such as CONGA [13] and XPath [24], the sender ToR isn't responsible for selecting optimal paths for data packets. Each switch independently chooses the best next hop to the destination. This has the additional advantage that switches do not need separate forwarding-table entries to track tunnels that are necessary for source-routing schemes [24]. This switch memory could be instead be used for supporting more ToRs in the HULA best hop table. Since the best hop table is updated by probes frequently at data-plane speeds, the packet forwarding in HULA quickly adapts to datacenter dynamics, such as flow arrivals and departures.

Automatic discovery of failures: HULA relies on the periodic arrival of probes as a keep-alive heartbeat from its neighboring switches. If a switch does not receive a probe from a neighboring switch for more than a certain threshold of time, then it ages the network utilization for that hop, making sure that hop is not chosen as the best hop for any destination ToR. Since the switch will pass this information to the upstream switches, the information about the broken path will reach all the relevant switches within an RTT. Similarly, if the failed link recovers, the next time a probe is received on the link, the hop will become a best hop candidate for the reachable destinations. This makes for a very fast adaptive forwarding technique that is robust to network topology changes and an attractive alternative to slow routing schemes orchestrated by the control plane.

Proactive path discovery: In HULA, probes are sent separately from data packets instead of piggybacking on them. This lets congestion information be propagated on paths independent of the flow of data packets, unlike alternatives such as CONGA. HULA leverages this to send periodic probes on paths that are not currently used by any switch. This way, switches can instanteously pick an uncongested path on the arrival of a new flowlet without having to first explore congested paths. In HULA, the switches on the path connected to the bottleneck link are bound to divert the flowlet onto a less-congested link and hence a less-congested path. This ensures short flows quickly get diverted to uncongested paths without spending too much time on path exploration.

Programmability: Processing a packet in a HULA switch involves switch state updates at line rate in the packet processing pipeline. In particular, processing a probe involves updating the best hop table and replicating the probe to neighboring switches. Processing a data packet involves reading the best hop table and updating a flowlet table if necessary. We demonstrate in section 5 that these operations can be naturally expressed in terms of reads and writes to match-action tables and register arrays in programmable data planes [29].

Topology and transport oblivious: HULA is not designed for a specific topology. It does not restrict the number of tiers in the network topology nor does it restrict the number of hops or the number of paths between any given pair of ToRs. However, as the topology becomes larger, the probe overhead can also be high and we discuss ways to minimize this overhead in section 4. Unlike load-balancing schemes that work best with symmetric topologies, HULA handles topology asymmetry very effectively as we demonstrate in section 6. This also makes incremental deployment plausible because HULA can be applied to either a subset of switches or a subset of the network traffic. HULA is also oblivious to the end-host application transport layer and hence does not require any changes to the host TCP stack.

4 HULA Design: Probes and Flowlets

The probes in HULA help proactively disseminate network utilization information to all switches. Probes originate at the leaf ToRs and switches replicate them as they travel through the network. This replication mechanism is governed by multicast groups set up once by the control plane. When a probe arrives on an incoming port, switches update the best path for flowlets traveling in the *opposite* direction. The probes also help discover and adapt to topology changes. HULA does all this while making sure the probe overhead is minimal.

In this section, we explain the probe replication mechanism (§4.1), the logic behind processing probe feedback (§4.2), how the feedback is used for flowlet routing (§4.3), how HULA adapts to topology changes (§4.4), and finally an estimate of the probe overhead on the network traffic and ways to minimize it (§4.5).

We assume that the network topology has the notion of upstream and downstream switches. Most datacenter network topologies have this notion built in them (with switches laid out in multiple tiers) and hence the notion can be exploited naturally. If a switch is in tier i, then the switches directly connected to it in tiers less than i are its *downstream* switches and the switches directly connected to it in tiers greater than i are its upstream switches. For example, in Figure 1, $T1$, $T2$ are the downstream switches for $A1$ and $S1$, $S2$ are its upstream switches.

4.1 Origin and Replication of HULA Probes

Every ToR sends HULA probes on all the uplinks that connect it to the datacenter network. The probes can be generated by either the ToR CPU, the switch data plane (if the hardware supports a packet

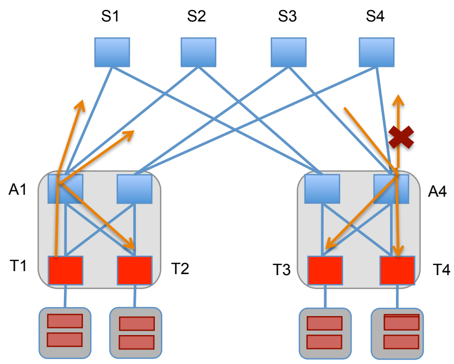


Figure 1: HULA probe replication logic

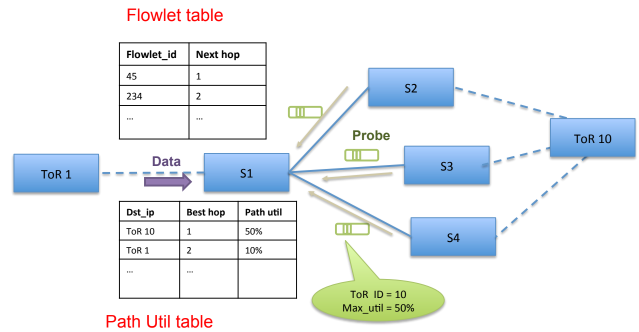


Figure 2: HULA probe processing logic

generator), or a server attached to the ToR. These probes are sent once every T_p seconds, which is referred to as the probe frequency hereafter in this paper. For example, in Figure 1, probes are sent by ToR $T1$, one on each of the uplinks connecting it to the aggregate switch $A1$.

Once the probes reach $A1$, it will forward the probe to all the other downstream ToRs ($T2$) and all the upstream spines ($S1$, $S2$). The spine $S1$ replicates the received probe onto all the other downstream aggregate switches. However, when the switch $A4$ receives a probe from $S3$, it replicates it to all its downstream ToRs (but not to other upstream spines — $S4$). This makes sure that all paths in the network are covered by the probes. This also makes sure that no probe loops forever.[2] Once a probe reaches another ToR, it ends its journey.

The control plane sets up multicast group tables in the data plane to enable the replication of probes. This is a one-time operation and does not have to deal with link failures and recoveries. This makes it easy to *incrementally* add switches to an existing set of multicast groups for replication. When a new switch is connected to the network, the control plane only needs to add the switch port to multicast groups on the adjacent upstream and downstream switches, in addition to setting up the multicast mechanism on the new switch itself.

4.2 Processing Probes to Update Best Path

A HULA probe packet is a minimum-sized packet of 64 bytes that contains a HULA header in addition to the normal Ethernet and IP headers. The HULA header has two fields:

- **torID (24 bits):** The leaf ToR at which the probe originated. This is the destination ToR for which the probe is carrying downstream path utilization in the opposite direction.
- **minUtil (8 bits):** The utilization of the best path if the packet were to travel in the opposite direction of the probe.

Link Utilization: Every switch maintains a link utilization estimator per switch port. This is based on an exponential moving average generator (EWMA) of the form $U = D + U * (1 - \frac{\Delta t}{\tau})$ where U is the link utilization estimator and D is the size of the outgoing packet that triggered the update for the estimator. Δt is the amount of time passed since the last update to the estimator and τ is a time constant that is at least twice the HULA probe frequency. In steady state, this estimator is equal to $C \times \tau$ where C is the outgoing link bandwidth. As discussed in section 5, this is a low pass filter similar to the DRE estimator used in CONGA [13]. We assume that a probe can access the TX (packets sent) utilization of the port that it enters.

[2]Where the notion of upstream/downstream switches is ambiguous [30], mechanisms like TTL expiry can also be leveraged to make sure HULA probes do not loop forever.

A switch uses the information on the probe header and the local link utilization to update switch state in the data plane before replicating the probe to other switches. Every switch maintains a best path utilization table ($pathUtil$) and a best hop table $bestHop$ as shown in Figure 2. Both the tables are indexed by a ToR ID. An entry in the $pathUtil$ table gives the utilization of the best path from the switch to a destination ToR. An entry in the $bestHop$ table is the next hop that has the minimum path utilization for the ToR in the $pathUtil$ table. When a probe with the tuple ($torID$, $probeUtil$) enters a switch on interface i, the switch calculates the min-max path utilization as follows:

- The switch calculates the maximum of $probeUtil$ and the TX link utilization of port i and assigns it to $maxUtil$.
- The switch then calculates the minimum of this $maxUtil$ and the $pathUtil$ table entry indexed by $torID$.
- If $maxUtil$ is the minimum, then it updates the $pathUtil$ entry with the newly determined best path utilization value $maxUtil$ and also updates the $bestHop$ entry for $torID$ to i.
- The probe header is updated with the latest $pathUtil$ entry for $torID$.
- The updated probe is then sent to the multicast table that replicates the probe to the appropriate neighboring switches as described earlier.

The above procedure carries out a distance-vector-like propagation of best path utilization information along all the paths destined to a particular ToR (from which the probes originate). The procedure involves each switch updating its local state and then propagating a summary of the update to the neighboring switches. This way any switch only knows the utilization of the best path that can be reached via a best next hop and does not need to keep track of the utilization of all the paths. The probe propagation procedure ensures that if the best path changes downstream, then that information will be propagated to all the relevant upstream switches on that path.

Maintaining best hop at line rate: Ideally, we would want to maintain a path utilization matrix that is indexed by both the ToR ID *and* a next hop. This way, the best next hop for a destination

ToR can be calculated by taking the minimum of all the next hop utilizations from this matrix. However, programmable data planes cannot calculate the minimum or maximum over an array of entries at line rate [31]. For this reason, instead of calculating the minimum over all hops, we maintain a current best hop and replace it in place when a better probe update is received.

This could lead to transient sub-optimal choices for the best hop entries – since HULA only tracks the current best path utilization, which could potentially go up in the future until a utilization update for the current best hop is received, HULA has no way of tracking other next hop alternatives with lower utilization that were also received within this window of time. However, we observe that this suboptimal choice can only be transient and will eventually converge to the best choice within a few windows of probe circulation. This approximation also reduces the amount of state maintained per destination from the order of number of neighboring hops to just one hop entry.

4.3 Flowlet Forwarding on Best Paths

HULA load balances at the granularity of flowlets in order to avoid packet reordering in TCP. As discussed earlier, a flowlet is detected by a switch whenever the inter-packet gap (time interval between the arrival of two consecutive packets) in a flow is greater than a flowlet threshold T_f. All subsequent packets, until a similar inter-packet gap is detected, are considered part of a new flowlet. The idea here is that the time gap between consecutive flowlets will absorb any delays caused by congested paths when the flowlets are sent on different paths. This will ensure that the flowlets will still arrive in order at the receiver and thereby not cause packet reordering. Typically, T_f is of the order of the network round trip time (RTT). In datacenter networks, T_f is typically of the order of a few hundreds of microseconds but could be larger in topologies with many hops.

HULA uses a flowlet hash table to record two pieces of information:the last time a packet was seen for the flowlet, and the best hop assigned to that flowlet. When the first packet for a flow arrives at a switch, it computes the hash of the flow's 5-tuple and creates an entry in the flowlet table indexed by the hash. In order to choose the best next hop for this flowlet, the switch looks up the $bestHop$ table for the destination ToR of the packet. This best hop is stored in the flowlet table and will be used for all subsequent packets of the flowlet. For example, when the second packet of a flowlet arrives, the switch looks up the flowlet entry for the flow and checks that the inter-packet gap is below T_f. If that is the case, it will use the best hop recorded in the flowlet table. Otherwise, a new flowlet is detected and it replaces the old flowlet entry with the current best hop, which will be used for forwarding the new flowlet.

Flowlet detection and path selection happens at every hop in the network. Every switch selects only the best next hop for a flowlet. This way, HULA avoids an explicit source routing mechanism for forwarding of packets. The only forwarding state required is already part of the $bestHop$ table, which itself is periodically updated to reflect congestion in the entire network.

Bootstrapping forwarding: To begin with, we assume that the path utilization is infinity (a large number in practice) on all paths to all ToRs . This gets corrected once the initial set of probes are processed by the switch. This means that if there is no probe from a certain ToR on a certain hop, then HULA will always choose a hop on which it actually received a probe. Thereafter, once the probes begin circulating in the network before sending any data packets, valid routes are automatically discovered.

4.4 Data-Plane Adaptation to Failures

In addition to learning the best forwarding routes from the probes, HULA also learns about link failures from the *absence* of probes. In particular, the data plane implements an aging mechanism for the entries in the $bestHop$ table. HULA tracks the last time $bestHop$ was updated using an $updateTime$ table. If a $bestHop$ entry for a destination ToR is not refreshed within the last T_{fail} (a threshold for detecting failures), then any other probe that carries information about this ToR (from a different hop) will simply replace the $bestHop$ and $pathUtil$ entries for the ToR. When this information about the change in the best path utilization is propagated further up the path, the switches may decide to choose a completely disjoint path if necessary to avoid the bottleneck link.

This way, HULA does not need to rely on the control plane to detect and adapt to failures. Instead HULA's failure-recovery mechanism is much faster than control-plane-orchestrated recovery, and happens at network RTT timescales. Also, note that this mechanism is better than having pre-coded backup routes because the flowlets immediately get forwarded on the next best alternative path as opposed to congestion-oblivious pre-installed backup paths. This in turn helps avoid sending flowlets on failed network paths and results in better network utilization and flow-completion times.

4.5 Probe Overhead and Optimization

The ToRs in the network need to send HULA probes frequently enough so that the network receives fine-grained information about global congestion state. However, the frequency should be low enough so that the network is not overwhelmed by probe traffic alone.

Setting probe frequency: We observe that even though network feedback is received on every packet, CONGA [13] makes flowlet routing decisions with probe feedback that is stale by an RTT because it takes a round trip time for the (receiver-reflected) feedback to reach the sender. In addition to this, the network switches only use the congestion information to make load balancing decisions when a new flowlet arrives at the switch. For a flow scheduled between any pair of ToRs, the best path information between these ToRs is used only when a new flowlet is seen in the flow, which happens at most once every T_f seconds. While it is true that flowlets for different flows arrive at different times, any flowlet routing decision is still made with probe feedback that is stale by at least an RTT. Thus, a reasonable sweet spot is to set the probe frequency to the order of the network RTT. In this case, the HULA probe information will be stale by at most a few RTTs and will still be useful for making quick decisions.

Optimization for probe replication: HULA also optimizes the number of probes sent from any switch A to an adjacent switch B. In the naive probe replication model, A sends a probe to neighbor B whenever it receives a probe on another incoming interface. So in a time window of length T_p (probe frequency), there can be multiple probes from A to B carrying the best path utilization information for a given ToR T, if there are multiple paths from T to A. HULA suppresses this redundancy to make sure that for any given ToR T, only one probe is sent by A to B within a time window of T_p. HULA maintains a $lastSent$ table indexed by ToR IDs. A replicates a probe update for a ToR T to B only if the last probe for T was sent more than T_p seconds ago. Note that this operation is similar to the calculation of a flowlet gap and can be done in constant time

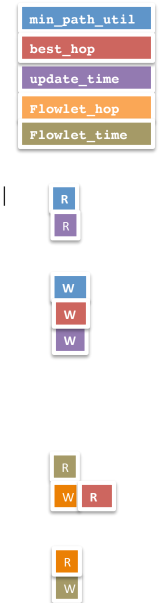
```
header_type hula_header {
      fields{
            dst_tor : 24;
            path_util : 8;
      }
}

header_type metadata{
      fields{
            nxt_hop : 8;
            self_id : 32;
            dst_tor : 32;
      }
}

control ingress {
      apply(get_dst_tor)
      apply(hula_logic)
      if(ipv4.protocol == PROTO_HULA){
            apply(hula_mcast);
      }
      else if(metadata.dst_tor
                  === metadata.self_id) {
            apply(send_to_host);
      }
}
```

(a) HULA header format and control flow

```
action hula_logic{
  if(ipv4_header.protocol == IP_PROTOCOLS_HULA){
         /*HULA Probe Processing
     if(hula_hdr.path_util < tx_util)
         hula_hdr.path_util = tx_util;
     if(hula_hdr.path_util < min_path_util[hula_hdr.dst_tor] ||
         curr_time - update_time[dst_tor] > KEEP_ALIVE_THRESH)
     {
         min_path_util[dst_tor] = hula_hdr.path_util;
         best_hop[dst_tor] = metadata.in_port;
         update_time[dst_tor] = curr_time;
     }
     hula_header.path_util = min_path_util[hula_hdr.dst_tor];
  }
 else { /*Flowlet routing of data */
     if(curr_time — flowlet_time[flow_hash]> FLOWLET_TOUT) {
         flowlet_hop[flow_hash] = best_hop[metadata.dst_tor];
     }
     metadata.nxt_hop = flowlet_hop[flow_hash];
     flowlet_time[flow_hash] = curr_time;
  }
}
```



(b) HULA stateful packet process in P4

Figure 3: Various components of the P4 program for HULA

in the data plane.[3] Thus, by making sure that on any link, only one probe is sent per destination ToR within this time window, the total number of probes that are sent on any link is proportional to the number of ToRs in the network alone and is not dependent on the number of possible paths the probes may take.

Overhead: Given the above parameter setting for the probe frequency and the optimization for probe replication, the probe overhead on any given network link is $\frac{probeSize*numToRs*100}{probeFreq*linkBandwidth}$ where *probeSize* is 64 bytes, *numTors* is the total number of leaf ToRs supported in the network and *probeFreq* is the HULA probe frequency. Therefore, in a network with 40G links supporting a total of 1000 ToRs, with probe frequency of $1ms$, the overhead comes to be 1.28%.

5 Programming HULA in P4

5.1 Introduction to P4

P4 is a packet-processing language designed for programmable data-plane architectures like RMT [16], Intel Flexpipe [17], and Cavium Xpliant [18]. The language is based on an abstract forwarding model called protocol-independent switch archtecture (PISA) [32]. In this model, the switch consists of a programmable parser that parses packets from bits on the wire. Then the packets enter an ingress pipeline containing a series of match-action tables that modify packets if they match on specific packet header fields. The packets are then switched to the output ports. Subsequently, the packets are processed by another sequence of match-action tables in the egress pipeline before they are serialized into bytes and transmitted.

A P4 program specifies the the protocol *header format*, a *parse graph* for the various headers, the definitions of tables with their match and action formats and finally the *control flow* that defines the order in which these tables process packets. This program defines the configuration of the hardware at compile time. At runtime, the tables are populated with entries by the control plane and network packets are processed using these rules. The programmer writes P4 programs in the syntax described by the P4 specification [29].

Programming HULA in P4 allows a network operator to compile HULA to any P4 supported hardware target. Additionally, network operators have the flexibility to modify and recompile their HULA P4 program as desired (changing parameters and the core HULA logic) without having to invest in new hardware. The wide industry interest in P4 [33] suggests that many switch vendors will soon have P4 compilers from P4 to their switch hardware, permitting operators to program HULA on such switches in the future.

5.2 HULA in P4

We describe the HULA packet processing pipeline using version 1.1 of P4 [29]. We make two minor modifications to the specification for the purpose of programming HULA.

1. We assume that the link utilization for any output port is available in the ingress pipeline. This link utilization can be computed using a low-pass filter applied to packets leaving a particular output port, similar to the Discounting Rate Estimator (DRE) used by CONGA [13]. At the language level, a link utilization object is syntactically similar to counter/meter objects in P4.

[3] If a probe arrives with the latest best path (after this bit is set), we are still assured that this best path information will be replicated (and propagated) in the next window assuming it still remains the best path.

2. Based on recent proposals [34] to modify P4, we assume support for the conditional operator within P4 actions.[4]

We now describe various components of the HULA P4 program in Figure 3. The P4 program has two main components: one, the HULA probe header format and parser specification, and two, packet control flow, which describes the main HULA logic.

Header format and parsing: We define the P4 header format for the probe packet and the parser state machine as shown in Figure 3(a). The header consists of two fields and is of size 4 bytes. The parser parses the HULA header immediately after the IPv4 header based on the special HULA protocol number in the IPv4 protocol field. Thereafter, the header fields are accessible in the pipeline through the header instance. The metadata header is used to access packet fields that have special meaning to a switch pipeline (e.g., the next hop) and local variables to be carried across multiple tables (e.g, a data packet's destination ToR or the current switch ID).

The control flow in Figure 3(a) shows that the processing pipeline first finds the ToR that the incoming packet is destined to. This is done by the `get_dst_tor` table that matches on the destination IP address and retrieves the destination ToR ID. Then the packet is processed by the `hula_logic` table whose actions are defined in Figure 3(b). Subsequently, the probe is sent to the `hula_mcast` table that matches on the in_port the probe came in and then assigns the appropriate set of multicast ports for replication.

HULA pipeline logic: Figure 3(b) shows the main HULA table where a series of actions perform two important pieces of HULA logic — (i) Processing HULA *probes* and (ii) Flowlet forwarding for *data* packets. We briefly describe how these two are expressed in P4. At a high level, the `hula_logic` table reads and writes to five register data structures shown in Figure 3(b) — `path_util`, `best_hop`, `update_time`, `flowlet_hop` and `flowlet_time`. The reads and writes performed by each action are color coded in the figure. For example, the red colored write tagging line 9 indicates that the action makes a write access to the `best_hop` register array.

1. Processing HULA probes: In step 1, the path utilization being carried by the HULA probe is updated (lines 4-5) with the the maximum of the local link utilization (`tx_util`) and the probe utilization. This gives the path utilization across all the hops including the link connecting the switch to its next hop. Subsequently, the current best path utilization value for the ToR is read from the `min_path_util` register into a temporary metadata variable (line 5).

In the next step, if either the probe utilization is less than the current best path utilization (line 6) or if the best hop was not refreshed in the last failure detection window (line 7), then three updates take place - (i) The best path utilization is updated with the probe utilization (line 9), (ii) the best hop value is updated with the incoming interface of the probe (line 10), and (iii) the best hop refresh time is updated with the current timestamp (line 11). Finally, the probe utilization itself is updated with the final best hop utilization (line 13). Subsequently the probe is processed by the `hula_mcast` match-action table that matches on the probe's input port and then assigns the appropriate multicast group for replication.

2. Flowlet Forwarding: If the incoming packet is a data packet (line 15), first we detect new flowlets by checking if the inter-packet gap for that flow is above the flowlet threshold (line 16). If that is the case, then we use the current best hop to reach the destination ToR (line 17). Subsequently, we populate the next hop metadata with the final flowlet hop (line 19). Finally, the arrival time of the packet is noted as the last seen time for the flowlet (line 20).

The benefits of programmability: Writing a HULA program in P4 gives multiple advantages to a network operator compared to a dedicated ASIC implementation. The operator could modify the sizes of various registers according to her workload demands. For example, she could change the sizes of the `best_hop` and `flowlet` register arrays based on her requirements. More importantly, she could change the way the algorithm works by modifying the HULA header to carry and process queue occupancy instead of link utilization to implement backpressure routing [27, 28].

5.3 Feasibility of P4 Primitives at Line Rate

In the P4 program shown in Figure 3, we require both stateless (i.e., operations that only read or write packet fields) and stateful (i.e., operations that may also maniupulate switch state in addition to packet fields) operations to program HULA's logic. We briefly comment on the hardware feasibility of each kind of operation below.

The stateless operations used in the program (like the assignment operation in line 4) are relatively easy to implement and have been discussed before [16]. In particular, Table 1 of the RMT paper [16] lists many stateless operations that are feasible on a programmable switch architecture with forwarding performance competitive with the highest-end fixed-function switches.

For determining the feasibility of stateful operations, we use techniques developed in Domino [31], a recent system that allows stateful data-plane algorithms such as HULA to be compiled to line-rate switches. The Domino compiler takes as inputs a data-plane algorithm and a set of *atoms*, which represent a programmable switch's instruction set. Based on the atoms supported by a programmable switch, Domino determines if a data-plane algorithm can be run on a line-rate switch. The same paper also proposes atoms that are expressive enough for a variety of data-plane algorithms, while incurring $< 15\%$ estimated chip area overhead. Table 3 of the Domino paper [31] lists these atoms.

We now discuss how the stateful operations required by each of HULA's five state variables `min_path_util`, `best_hop`, `update_time`, `flowlet_hop`, and `flowlet_time`, can be supported by Domino's atoms (the atom names used here are from Table 3 of the Domino paper [31]).

1. Both `flowlet_time` and `update_time` track the last time at which some event happened, and require only a simply read/write capability to a state variable (the Read/Write atom).

2. The `flowlet_hop` variable is conditionally updated whenever the flowlet threshold is exceeded. This requires the ability to predicate a write to a state variable based on some condition (the PRAW atom).

3. The variables `min_path_util` and `best_hop` are mutually dependent on one another: `min_path_util` (the utilization on the best hop) needs to be updated if a new probe is received for the current `best_hop` (the variable tracking the best next hop) ; conversely, the `best_hop` variable needs to be updated if a probe for another path indicates a utilization lesser than the current `min_path_util`. This mutually dependence requires hardware support for updating

[4]For ease of exposition, we replace conditional operators with equivalent if-else statements in Figure 3.

a *pair* of state variables depending on the previous values of the pair (the Pairs atom).

The most complex of these three atoms (Read/Write, PRAW, and Pairs) is the Pairs atom. However, even the Pairs atom only incurs modest estimated chip area overhead based on synthesis results from a 32 nm standard-cell library. Further, this atom is useful for other algorithms besides HULA as well (Table 4 of the Domino paper describes several more examples). We conclude based on these results that it is feasible to implement the instructions required by HULA without sacrificing the performance of a line-rate switch.

6 Evaluation

In this section, we illustrate the effectiveness of the HULA load balancer by implementing it in the ns-2 discrete event simulator and comparing it with the following alternative load balancing schemes:

1. **ECMP:** Each flow's next hop is determined by taking a hash of the flow's five tuple (src IP, dest IP, src port, dest port, protocol).

2. **CONGA':** CONGA [13] is the closest alternative to HULA for congestion-aware data-plane load balancing. However, CONGA is designed specifically for 2-tier Leaf-Spine topologies. However, according to the authors [35], if CONGA is to be extended to larger topologies, CONGA should be applied within each pod and for cross-pod traffic, ECMP should be applied at the flowlet level. This method involves taking a hash of the six tuple that includes the flow's five tuple and the flowlet ID (which is incremented every time a new flowlet is detected at a switch). This hash is subsequently used by all the switches in the network to find the next hop for each flowlet. We refer to this load balancing scheme as CONGA' in our evaluation results.

We use our experiments to answer the following questions:

- How does HULA perform in the baseline topology compared to other schemes?
- How does HULA perform when there is asymmetry in the network?
- How quickly does HULA adapt to changes in the network like link failures?
- How robust is HULA to various parameters settings?

Topology: We simulated a 3-tier Fat-Tree topology as shown in Figure 4, with two spines (S1 and S2) connecting two *pods*. Each pod contains two aggregate switches connected to two leaf ToRs with 40G links. Each ToR is connected to 8 servers with 10G links. This ensures that the network is not oversubscribed: the 16 servers in one pod can together use the 160G bandwidth available for traffic across the two pods. In this topology, even though there are only two uplinks from any given ToR, there are a total of 8 different paths available between a pair of ToRs sitting in different pods. To simulate asymmetry in the baseline symmetric topology, we disable the 40G link connecting the spine *S*2 with the aggregate switch *A*4.

Empirical Workload: We use two realistic workloads to generate traffic for our experiments - (i) A Web-search workload [36] and (ii) a data-mining workload [25]. Both of these workloads are

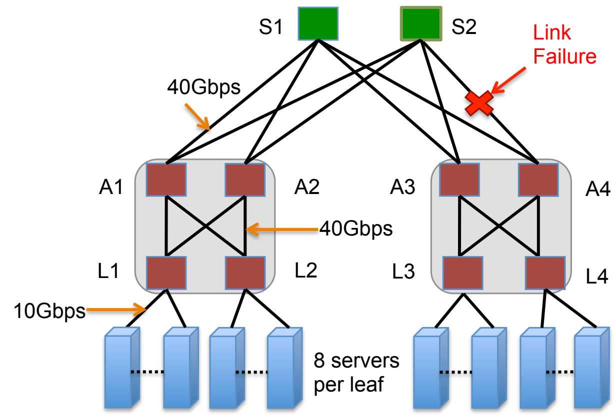


Figure 4: Topology used in evaluation

obtained from production datacenters. Figure 5a shows the cumulative distribution of flow sizes seen in these two workloads. Note that flow sizes in the CDF are in log scale. Both the workloads are heavy tailed: most flows are small, while a small number of large flows contribute to a substantial portion of the traffic. For example, in the data mining workload, 80% of the flows are of size less than 10KB.

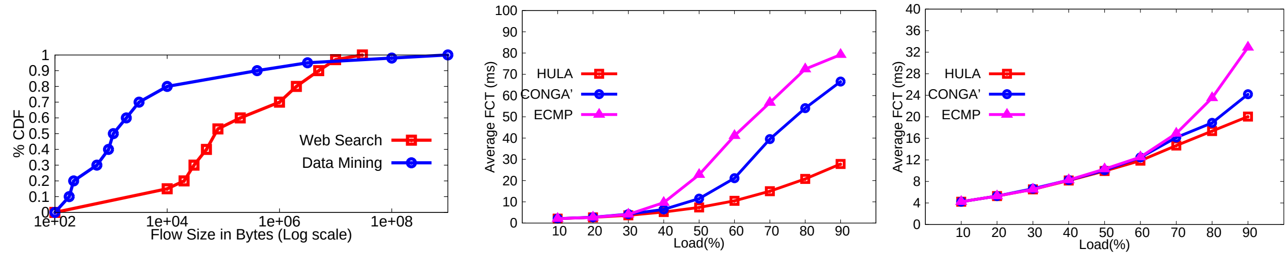
We simulate a simple client-server communication model where each client chooses a server at random and initiates three persistent TCP connections to the server. The client sends a flow with size drawn from the empirical CDF of one of the two workloads. The inter-arrival rate of the flows on a connection is also taken from an exponential distribution whose mean is tuned to achieve a desired load on the network. Similar to previous work [6, 13], we look at the average flow completion time (FCT) as the overall performance metric so that all flows including the majority of small flows are given equal consideration. We run each experiment with three random seeds and then measure the average of the three runs.

Parameters: In our experimental setting, there are two important parameters that determine the system behavior. First, the flowlet inter-packet gap, as is recommended in previous work [11, 13], is set to be of the order of the network RTT so that packet reordering at the receiver is minimized. In our experiments, we used a flowlet gap of 100 µs . The second parameter is the probe frequency, which (as mentioned in §4.5) is set to few times the RTT so that it is frequent enough to quickly react to congestion but does not overwhelm the network. In our experiments, unless stated explicitly, the probe frequency was set to 200 µs.

6.1 Symmetric 3-tier Fat-Tree Topology

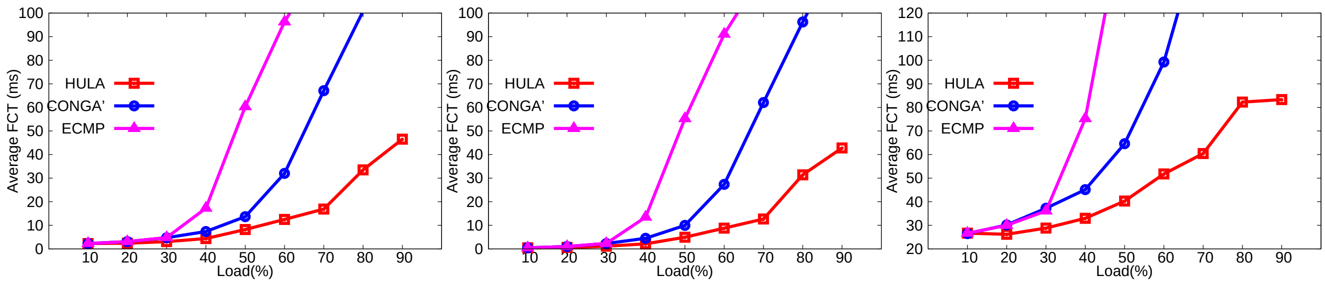
Figure 5 shows the average completion time for all flows as the load on the network is varied. HULA performs better than ECMP and CONGA' for both the workloads at higher loads. At lower loads, the performance of all three load balancing schemes is nearly the same because when there is enough bandwidth available in the network, there is a greater tolerance for congestion-oblivious path forwarding. However, as the network load becomes higher, the flows have to be carefully assigned to paths such that collisions do not occur. Given that flow characteristics change frequently, at high network load, the load balancing scheme has to adapt quickly to changes in link utilizations throughout the network.

ECMP performs the worst because it performs congestion-oblivious load balancing at a very coarse granularity. CONGA' does slightly



(a) Empirical traffic distribution used in evaluation (b) Web-search overall avg FCT (c) Data-mining overall avg FCT

Figure 5: Average FCT for the Web-search and data-mining workload on the *symmetric* topology.



(a) Overall Average FCT (b) Small Flows (<100KB) (c) Large Flows (>10MB)

Figure 6: Average FCT for the Web-search workload on the *asymmetric* topology.

better because it still does congestion-oblivious ECMP (across pods) but at the granularity of flowlets. In particular, flows sent on congested paths see more inter-flowlet gaps being created due to the delay caused by queue growth. Hence, compared to ECMP, CONGA' has additional opportunities to find an uncongested path when new flowlets are hashed. HULA performs the best because of its fine-grained congestion-aware load balancing. For the Web-search workload, HULA achieves 3.7x lower FCT (better performance) compared to ECMP and 2.7x better compared to CONGA' at 70% network load. The performance of HULA is slightly less apparent in the data mining workload because a vast portion of the flows in the workload are really small (50% are just 1 packet flows) and HULA does not often get a chance to better load balance large flows with multiple flowlets. Nevertheless, HULA achieves 1.35x better performance than ECMP at 80% network load.

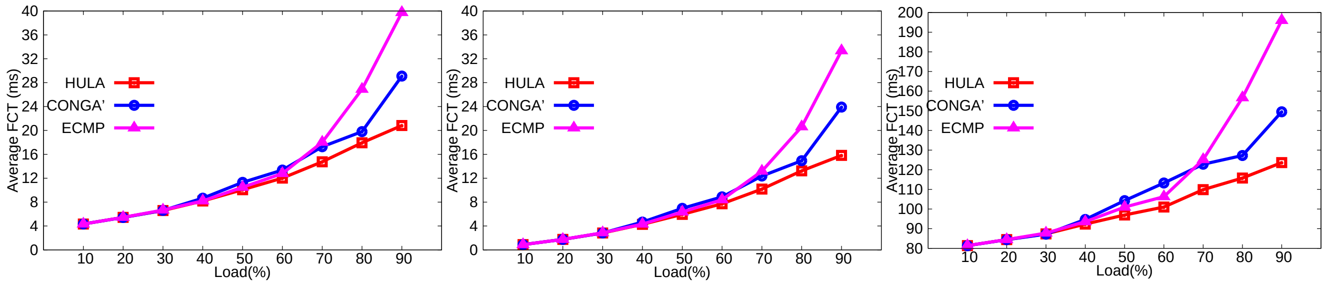
6.2 Handling Topology Asymmetry

When the link between the spine switch $S2$ and switch $A4$ is removed, the effective bandwidth of the network drops by 25% for traffic going across the pods. This means that the load balancing schemes have to carefully balance paths at even lower network loads compared to the baseline topology scenario. In particular, the load balancing scheme has to make sure that the bottleneck link connecting $S2$ to $A3$ is not overwhelmed with a disproportionate amount of traffic.

Figure 6 shows how various schemes perform with the Web-search workload as the network load is varied. The overall FCT for ECMP rises quickly and goes off the charts beyond a 60% network load. Once the network load reaches 50%, the bottleneck link incurs pressure from the flows hashed to go through $S2$. This is why ECMP and CONGA' have bad performance at high network loads. CONGA' does slightly better than ECMP here because the network sees more flowlets being created on congested paths (due to the delays caused by queue growth) and hence has a slightly higher chance of finding the uncongested paths for new flowlets. Because of this, CONGA' is 3x better than ECMP at 60% load. However, HULA performs the best because of its proactive utilization-aware path selection, which avoids pressure on the bottleneck link. This helps HULA achieve 8x better performance at 60% network load.

Figure 6(b) shows the average FCTs for small flows of size less than 100KB and Figure 6(c) shows the average FCTs for large flows of size greater than 10MB. HULA's gains are most pronounced on the large number of small flows where it does 10x better than ECMP at 60% load. Even for large flows, HULA is 4x better than ECMP at 60% load.

HULA prevents queue growth: The superior performance of HULA can be understood by looking at the growth of switch queues. As described earlier, in the link failure scenario, all the traffic that crosses the pod through the spine $S2$ has to go through the link connecting it to $A3$, which becomes the bottleneck link at high network load. Figure 8c shows the CDF of queue depth at the bottleneck link. The queue was monitored every 100 microseconds and the instantaneous queue depth was plotted. ECMP has high depth most of the time and frequently sees packet drops as well. HULA on the other hand maintains zero queue depth 90% of the time and sees no packet drops. In addition, the 95th percentile queue depth for HULA is 8x smaller compared to CONGA' and 19x smaller compared to ECMP.

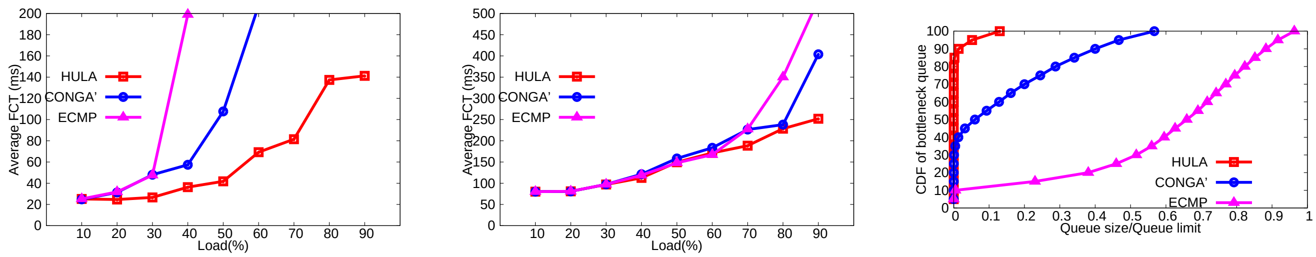


(a) Overall Average FCT

(b) Small Flows (<100KB)

(c) Large Flows (>10MB)

Figure 7: Average FCT for the data mining workload on the *asymmetric* topology.



(a) 99th percentile FCT for Web-search workload

(b) 99th percentile FCT for datamining

(c) Queue length at bottleneck link (S2->A3) in the link failure scenario

Figure 8: 99th percentile FCTs and queue growth on the asymmetric topology

Figure 7 shows that HULA's gains are less pronounced with the data mining workload similar to what was seen with the baseline topology. Due to the extremely large number of small flows, the effect of congestion-aware load balancing is less pronounced. Nevertheless, HULA does the best with small flows having 1.53x better performance than ECMP at 80% load. With large flows, it does 1.35x better than ECMP. Overall, HULA does 1.52x better than ECMP and 1.17x better than CONGA'.

HULA achieves better tail latency: In addition to performing better on average FCT, HULA also achieves good tail latency for both workloads. Figure 8 shows the 99th percentile FCT for all the flows. For the Web-search workload, HULA achieves 10x better 99th percentile FCT compared to ECMP and 3x better compared to CONGA' at 60% load. For the data mining workload, HULA achieves 1.53x better tail latency compared to ECMP.

6.3 Stability

In order to study HULA's stability in response to topology changes, we monitored the link utilization of the links that connect the spine to the aggregate switches in the asymmetric topology while the Web-search workload is running. We then brought down the bottleneck link at 0.2 milliseconds from the beginning of the experiment. As Figure 9(a) shows, HULA quickly adapts to the failure and redistributes the load onto the two links going through $S1$ within a millisecond. Then when the failed link comes up later, HULA quickly goes back to the original utilization values on all the links. This demonstrates that HULA is robust to changes in the network topology and also shows that the load is distributed almost equally on all the available paths at any given time regardless of the topology.

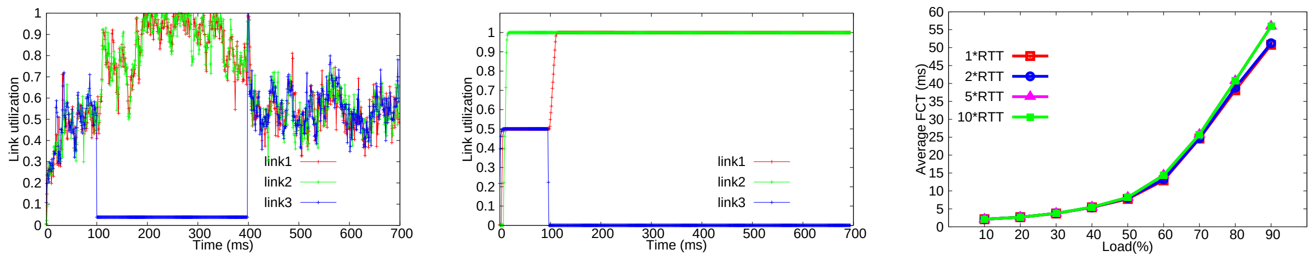
Figure 9(b) shows a similar experiment but run with long-running flows as opposed to the empirical workload. Long-running flows allow us to study HULA's stability better than empirical workloads, because in an emprical workload the link utilizations may fluctuate depending on flow arrivals and departures. As the figure shows, when the link connecting a spine to an aggregate switch fails, HULA quickly deflects the affected flows onto another available path within half a millisecond. Further, while doing this, it does not disturb the bottleneck link and cause instability in the network.

6.4 Robustness of probe frequency

As discussed earlier, carrying probes too frequently can reduce the effective network bandwidth available for data traffic. While we argued that the ideal frequency is of the order of the network RTT, we found that HULA is robust to change in probe frequency. Figure 9(c) shows the average FCT with the Web-search workload running on the asymmetric topology. When the network load is below 70%, increasing the probe frequency to 10 times its ideal has no effect on the performance. Even at 90% load, the average FCT for 10x frequency is only 1.15x higher. In addition, compared with ECMP and CONGA', these numbers are much better. Therefore, we believe HULA probes can be circulated with moderately low frequency so that the effective bandwidth is not affected while still achieving utilization-aware load balancing.

7 Related Work

Stateless or local load balancing: Equal-Cost Multi-Path routing (ECMP) is a simple hash-based load-balancing scheme that is im-



(a) Link utilization on failures with Web-search workload

(b) Link utilization on failures with long running flows

(c) Effect of decreasing probe frequency

Figure 9: HULA resilience to link failures and probe frequency settings

plemented widely in switch ASICs today. However, it is congestion-agnostic and only splits traffic at the flow level, which causes collisions at high network load. Further, ECMP is shown to have degraded performance during link failures that cause asymmetric topologies [13]. DRB [10] is a per-packet load balancing scheme that sprays packets effectively in a round robin fashion. More recently, PRESTO [37] proposed splitting flows into TSO (TCP Segment Offload) segments of size 64KB and sending them on multiple paths. On the receive side GRO (General Receive Offload), the packets are buffered temporarily to prevent reordering. Neither DRB nor Presto is congestion aware, which causes degraded performance during link failures. Flare [11] and Localflow [12] discuss switch-local solutions that balance the load on all switch ports but do not take global congestion information into account.

Centralized load balancing: B4 [15] and SWAN [14] propose centralized load balancing for wide-area networks connecting their data centers. They collect statistics from network switches at a central controller and push forwarding rules to balance network load. The control plane operates at the timescale of minutes because of relatively predictable traffic patterns. Hedera [8] and MicroTE [9] propose similar solutions for datacenter networks but still suffer from high control-loop latency in the critical path and cannot handle highly volatile datacenter traffic in time.

Modified transport layer: MPTCP [38] is a modified version of TCP that uses multiple subflows to split traffic over different paths. However, the multiple subflows cause burstiness and perform poorly under Incast-like conditions [13]. In addition, it is difficult to deploy MPTCP in datacenters because it requires change to all the tenant VMs, each of which might be running a different operating system. DCTCP [36], pFabric [6] and PIAS [39] reduce the tail flow completion times using modified end-host transport stacks but do not focus on load balancing. DeTail [40] proposes a per-packet adaptive load balancing scheme that adapts to topology asymmetry but requires a complex cross-layer network stack including end-host modifications.

Global utilization-aware load balancing TeXCP [41] and MATE [42] are adaptive traffic-engineering proposals that load balance across multiple ingress-egress paths in a wide-area network based on per-path congestion metrics. TeXCP also does load balancing at the granularity of flowlets but uses router software to collect utilization information and uses a modified transport layer to react to this information. HALO [43], inspired by a long line of work beginning with Minimum Delay Routing [44], studies load-sensitive adaptive routing as an optimization problem and implements it in the router software. Relative to these systems, HULA is a routing mechanism that balances load at finer granularity and is simple enough to be implemented entirely in the data plane.

As discussed earlier, CONGA [13] is the closest alternative to HULA for global congestion-aware fine-grained load balancing. However, it is designed for specific 2-tier Leaf-Spine topologies in a custom ASIC. HULA, on the other hand, scales better than CONGA by distributing the relevant utilization information across all switches. In addition, unlike CONGA, HULA reacts to topology changes like link failures almost instantaneously using data-plane mechanisms. Lastly, HULA's design is tailored towards programmable switches—a first for data-plane load balancing schemes.

8 Conclusion

In this paper, we design HULA (Hop-by-hop Utilization-aware Load balancing Architecture), a scalable load-balancing scheme designed for programmable data planes. HULA uses periodic probes to perform a distance-vector style distribution of network utilization information to switches in the network. Switches track the next hop for the best path and its corresponding utilization for a given destination, instead of maintaining per-path utilization congestion information for each destination. Further, because HULA performs forwarding locally by determining the next hop and not an entire path, it eliminates the need for a separate source routing mechanism (and the associated forwarding table state required to maintain source routing tunnels). When failures occur, utilization information is automatically updated so that broken paths are avoided.

We evaluate HULA against existing load balancing schemes and find that it is more effective and scalable. While HULA is effective enough to quickly adapt to the volatility of datacenter workloads, it is also simple enough to be implemented at line rate in the data plane on emerging programmable switch architectures. While the performance and stability of HULA is studied empirically in this paper, an analytical study of its optimality and stability will provide further insights into its dynamic behavior.

Acknowledgments: We thank the SOSR reviewers for their valuable feedback, Mohammad Alizadeh for helpful discussions about extending CONGA to larger topologies, and Mina Tahmasbi Arashloo for helpful comments on the writing. This work was supported in part by the NSF under the grant CNS-1162112 and the ONR under award N00014-12-1-0757.

9 References

[1] N. Kang, Z. Liu, J. Rexford, and D. Walker, "Optimizing the "one big switch" abstraction in software-defined networks," CoNEXT '13, (New York, NY, USA), ACM.

[2] M. Alizadeh and T. Edsall, "On the data path performance of leaf-spine datacenter fabrics," in *HotInterconnects 2013*, pp. 71–74.

[3] J. Perry, A. Ousterhout, H. Balakrishnan, D. Shah, and H. Fugal, "Fastpass: A centralized "zero-queue" datacenter network," SIGCOMM, 2014, (New York, NY, USA), pp. 307–318, ACM.

[4] V. Jeyakumar, M. Alizadeh, D. Mazières, B. Prabhakar, C. Kim, and A. Greenberg, "Eyeq: Practical network performance isolation at the edge," NSDI 2013, (Berkeley, CA, USA), pp. 297–312, USENIX Association.

[5] L. Popa, A. Krishnamurthy, S. Ratnasamy, and I. Stoica, "Faircloud: Sharing the network in cloud computing," HotNets-X, (New York, NY, USA), pp. 22:1–22:6, ACM, 2011.

[6] M. Alizadeh, S. Yang, M. Sharif, S. Katti, N. McKeown, B. Prabhakar, and S. Shenker, "pfabric: Minimal near-optimal datacenter transport," SIGCOMM 2013, (New York, NY, USA), pp. 435–446, ACM.

[7] M. Chowdhury, Y. Zhong, and I. Stoica, "Efficient coflow scheduling with varys," in *Proceedings of the 2014 ACM Conference on SIGCOMM*, SIGCOMM '14, (New York, NY, USA), pp. 443–454, ACM, 2014.

[8] M. Al-Fares, S. Radhakrishnan, B. Raghavan, N. Huang, and A. Vahdat, "Hedera: Dynamic flow scheduling for data center networks," NSDI 2010, (Berkeley, CA, USA), pp. 19–19, USENIX Association.

[9] T. Benson, A. Anand, A. Akella, and M. Zhang, "Microte: Fine grained traffic engineering for data centers," CoNEXT 2011, pp. 8:1–8:12, ACM.

[10] J. Cao, R. Xia, P. Yang, C. Guo, G. Lu, L. Yuan, Y. Zheng, H. Wu, Y. Xiong, and D. Maltz, "Per-packet load-balanced, low-latency routing for clos-based data center networks," CoNEXT 2013, pp. 49–60, ACM.

[11] S. Kandula, D. Katabi, S. Sinha, and A. Berger, "Dynamic load balancing without packet reordering," *SIGCOMM Comput. Commun. Rev.*, vol. 37, pp. 51–62, Mar. 2007.

[12] S. Sen, D. Shue, S. Ihm, and M. J. Freedman, "Scalable, optimal flow routing in datacenters via local link balancing," CoNEXT 2013, pp. 151–162, ACM.

[13] M. Alizadeh, T. Edsall, S. Dharmapurikar, R. Vaidyanathan, K. Chu, A. Fingerhut, V. T. Lam, F. Matus, R. Pan, N. Yadav, and G. Varghese, "Conga: Distributed congestion-aware load balancing for datacenters," *SIGCOMM Comput. Commun. Rev.*, vol. 44, pp. 503–514, Aug. 2014.

[14] C.-Y. Hong, S. Kandula, R. Mahajan, M. Zhang, V. Gill, M. Nanduri, and R. Wattenhofer, "Achieving high utilization with software-driven wan," SIGCOMM 2013, pp. 15–26, ACM.

[15] S. Jain, A. Kumar, S. Mandal, J. Ong, L. Poutievski, A. Singh, S. Venkata, J. Wanderer, J. Zhou, M. Zhu, J. Zolla, U. Hölzle, S. Stuart, and A. Vahdat, "B4: Experience with a globally-deployed software defined wan," SIGCOMM 2013, pp. 3–14, ACM.

[16] P. Bosshart, G. Gibb, H.-S. Kim, G. Varghese, N. McKeown, M. Izzard, F. Mujica, and M. Horowitz, "Forwarding Metamorphosis: Fast Programmable Match-action Processing in Hardware for SDN," in *SIGCOMM*, 2013.

[17] "Intel FlexPipe." http://www.intel.com/content/dam/www/public/us/en/documents/product-briefs/ethernet-switch-fm6000-series-brief.pdf.

[18] "Cavium and XPliant introduce a fully programmable switch silicon family scaling to 3.2 terabits per second." http://tinyurl.com/nzbqtr3.

[19] P. Bosshart, D. Daly, G. Gibb, M. Izzard, N. McKeown, J. Rexford, C. Schlesinger, D. Talayco, A. Vahdat, G. Varghese, and D. Walker, "P4: Programming protocol-independent packet processors," *SIGCOMM Comput. Commun. Rev.*, vol. 44, pp. 87–95, July 2014.

[20] T. Issariyakul and E. Hossain, *Introduction to Network Simulator NS2*. Springer Publishing Company, Incorporated, 1st ed., 2010.

[21] R. Niranjan 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," SIGCOMM 2009, pp. 39–50, ACM.

[22] "Cisco's massively scalable data center." http://www.cisco.com/c/dam/en/us/td/docs/solutions/Enterprise/Data_Center/MSDC/1-0/MSDC_AAG_1.pdf, Sept 2015.

[23] "High Capacity StrataXGS®Trident II Ethernet Switch Series." http://www.broadcom.com/products/Switching/Data-Center/BCM56850-Series.

[24] S. Hu, K. Chen, H. Wu, W. Bai, C. Lan, H. Wang, H. Zhao, and C. Guo, "Explicit path control in commodity data centers: Design and applications," NSDI 2015, pp. 15–28, USENIX Association.

[25] A. Greenberg, J. R. Hamilton, N. Jain, S. Kandula, C. Kim, P. Lahiri, D. A. Maltz, P. Patel, and S. Sengupta, "Vl2: A scalable and flexible data center network," *SIGCOMM Comput. Commun. Rev.*, vol. 39, pp. 51–62, Aug. 2009.

[26] 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," SIGCOMM 2009, pp. 63–74, ACM.

[27] E. Athanasopoulou, L. X. Bui, T. Ji, R. Srikant, and A. Stolyar, "Back-pressure-based packet-by-packet adaptive routing in communication networks," *IEEE/ACM Trans. Netw.*, vol. 21, pp. 244–257, Feb. 2013.

[28] B. Awerbuch and T. Leighton, "A simple local-control approximation algorithm for multicommodity flow," pp. 459–468, 1993.

[29] "P4 Specification." http://p4.org/wp-content/uploads/2015/11/p4-v1.1rc-Nov-17.pdf.

[30] S. Radhakrishnan, M. Tewari, R. Kapoor, G. Porter, and A. Vahdat, "Dahu: Commodity switches for direct connect data center networks," ANCS 2013, pp. 59–70, IEEE Press.

[31] A. Sivaraman, M. Budiu, A. Cheung, C. Kim, S. Licking, G. Varghese, H. Balakrishnan, M. Alizadeh, and N. McKeown, "Packet transactions: A programming model for data-plane algorithms at hardware speed," *CoRR*, vol. abs/1512.05023, 2015.

[32] "Protocol-independent switch architecture." http://schd.ws/hosted_files/p4workshop2015/c9/NickM-P4-Workshop-June-04-2015.pdf.

[33] "Members of the p4 consortium." http://p4.org/join-us/.

[34] "P4's action-execution semantics and conditional operators." https://github.com/anirudhSK/p4-semantics/raw/master/p4-semantics.pdf.

[35] Private communication with the authors of CONGA.

[36] M. Alizadeh, A. Greenberg, D. A. Maltz, J. Padhye, P. Patel, B. Prabhakar, S. Sengupta, and M. Sridharan, "Data center tcp (dctcp)," SIGCOMM 2010, pp. 63–74, ACM.

[37] K. He, E. Rozner, K. Agarwal, W. Felter, J. Carter, and A. Akella, "Presto: Edge-based load balancing for fast datacenter networks," in *SIGCOMM*, 2015.

[38] C. Raiciu, S. Barre, C. Pluntke, A. Greenhalgh, D. Wischik, and M. Handley, "Improving datacenter performance and robustness with multipath tcp," SIGCOMM 2011, pp. 266–277, ACM.

[39] W. Bai, L. Chen, K. Chen, D. Han, C. Tian, and H. Wang, "Information-agnostic flow scheduling for commodity data centers," NSDI 2015, pp. 455–468, USENIX Association.

[40] D. Zats, T. Das, P. Mohan, D. Borthakur, and R. Katz, "Detail: Reducing the flow completion time tail in datacenter networks," SIGCOMM 2012, pp. 139–150, ACM.

[41] S. Kandula, D. Katabi, B. Davie, and A. Charny, "Walking the tightrope: Responsive yet stable traffic engineering," SIGCOMM 2005, pp. 253–264, ACM.

[42] A. Elwalid, C. Jin, S. Low, and I. Widjaja, "Mate: Mpls adaptive traffic engineering," in *IEEE INFOCOM 2001*, pp. 1300–1309 vol.3.

[43] N. Michael and A. Tang, "Halo: Hop-by-hop adaptive link-state optimal routing," *Networking, IEEE/ACM Transactions on*, vol. PP, no. 99, pp. 1–1, 2014.

[44] R. Gallager, "A minimum delay routing algorithm using distributed computation," *Communications, IEEE Transactions on*, vol. 25, pp. 73–85, Jan 1977.