Expeditus: Congestion-aware Load Balancing in Clos Data Center Networks

Peng Wang[1] Hong Xu[1] Zhixiong Niu[1] Dongsu Han[2] Yongqiang Xiong[3]

[1] NetX Lab, City University of Hong Kong [2]KAIST [3]Microsoft Research Asia

Abstract

Data center networks often use multi-rooted Clos topologies to provide a large number of equal cost paths between two hosts. Thus, load balancing traffic among the paths is important for high performance and low latency. However, it is well known that ECMP—the *de facto* load balancing scheme—performs poorly in data center networks. The main culprit of ECMP's problems is its congestion agnostic nature, which fundamentally limits its ability to deal with network dynamics.

We propose Expeditus, a novel distributed congestion-aware load balancing protocol for general 3-tier Clos networks. The complex 3-tier Clos topologies present significant scalability challenges that make a simple per-path feedback approach infeasible. Expeditus addresses the challenges by using simple local information collection, where a switch only monitors its egress and ingress link loads. It further employs a novel two-stage path selection mechanism to aggregate relevant information across switches and make path selection decisions. Testbed evaluation on Emulab and large-scale ns-3 simulations demonstrate that, Expeditus outperforms ECMP by up to 45% in tail flow completion times (FCT) for mice flows, and by up to 38% in mean FCT for elephant flows in 3-tier Clos networks.

Categories and Subject Descriptors C2.2 [*Computer - Communication Networks*]: Network Protocols

Keywords Datacenter networks, Load balancing, Network congestion

1. Introduction

Data center networks use multi-rooted Clos topologies to provide a large number of equal cost paths and abundant bandwidth between hosts [2, 17]. To load balance flows, the data plane runs ECMP—Equal Cost Multi-Path—that forwards packets among equal cost egress ports based on static hashing. ECMP is simple to implement and does not require per-flow state at switches. It is well known, however, that ECMP cannot satisfy stringent performance requirements in data centers. Hash collisions cause congestion, degrading throughput for elephant flows [3, 11, 14] and tail latency for mice flows [5, 7, 27, 37].

Distributed congestion-aware load balancing has recently emerged as a promising solution [4] in which switches monitor congestion levels of each path and direct flows to less congested paths. This approach has several practical advantages. Being a distributed scheme, it is more scalable and can deal with bursty traffic more gracefully than centralized scheduling (e.g. Hedera [3]). Being a data plane approach, it is independent of the host's networking stack (e.g. MPTCP) [30] and immediately benefits all traffic once deployed. Its end-to-end congestion visibility also makes it more robust to network asymmetry without control plane reconfiguration [19, 38].

The crux of designing a congestion-aware load balancing protocol is that we need to know real-time (in the order of RTT) congestion information from all paths between the flow's source and destination [4, 5]. A straightforward approach is to use end-to-end path-wise information: a ToR switch maintains end-to-end congestion metrics for all the paths from itself to other ToR switches in the network. Congestion metrics can be collected by data packet piggybacking. This works for the simple 2-tier leaf-spine topologies, as shown by for example CONGA [4]. A ToR switch keeps track of up to thousands of paths for a large-scale leaf-spine network [4].

Yet, path-wise feedback does not scale for complex 3-tier Clos topologies widely deployed in production data centers, such as Google's [32], Facebook's [8] and Amazon's [1]. Typically, hundreds of paths exist between any two ToR switches, and a ToR switch can communicate with hundreds of other ToR switches. Thus, per-path feedback requires storing $O(10^5)$–$O(10^6)$ path information at *each* ToR switch in a production scale 3-tier network. More importantly, it is impossible to collect real-time congestion information for all

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

SoCC '16, October 05 - 07, 2016, *Santa Clara, CA, USA.*

© 2016 Copyright held by the owner/author(s). Publication rights licensed to ACM.

ISBN 978-1-4503-4525-5/16/10. . . $15.00.
DOI: http://dx.doi.org/10.1145/2987550.2987560

these paths, as there would not be enough concurrent flows that happen to traverse all of them at the same time (§2.2).

The main contribution of this paper is the design, implementation, and evaluation of Expeditusa novel distributed congestion-aware load balancing protocol targeted at general 3-tier Clos topologies. Expeditus relies on two key designs, local congestion monitoring and two-stage path selection, to fundamentally overcome the design challenges discussed above (§2.3). Each switch only locally monitors the utilization of its uplinks to the upper tier switches. Local information collection ensures real-time congestion state is available without scalability or measurement overhead (§3.1).

Expeditus then applies two-stage path selection to aggregate relevant information across multiple hops and make load balancing decisions (§3.2). In the first stage, only the source and destination ToR switches are involved to select the best path from the ToR to the aggregation tier (i.e. the first and last hop). The source ToR switch sends its egress congestion metrics to the destination ToR, which combines them with its ingress congestion metrics to select the best path to the aggregation tier. In the second stage, the chosen aggregation switches then select the best core switch in a similar way based on congestion state of the second and third hops. The path selection decisions are then maintained at ToR and aggregation switches.

Essentially two-stage path selection uses partial-path information to heuristically find good paths for flows. By exploiting the structural properties of 3-tier Clos, two-stage path selection drastically reduces the complexity without much performance penalty. In fact, our evaluation shows that it performs effectively in many cases compared to a clairvoyant scheme with complete information. Expeditus performs path selection on a per-flow basis during the TCP threeway handshake, but does not cause packet reordering nor introduce any latency overhead.

We implement Expeditus using the Click software router [25]. We evaluate it on the Emulab testbed (§4) as well as in large-scale packet-level ns-3 simulations (§5), using empirical flow size distributions from two production networks. Our evaluation shows that Expeditus provides up to 45% reductions in 95%ile flow completion time (FCT) for mice flows compared to ECMP in 3-tier Clos. The average FCT of elephant flows ($\geq$1MB) is also reduced by $\sim$15%–38%. When applied to 2-tier leaf-spine networks, Expeditus outperforms CONGA working at per-flow granularity, because of its ability to observe global and timely congestion information. Expeditus advances state-of-the-art and has potential to make impact to the industry with its practical design. At the time of writing it is being considered by a major vendor for adoption into their next-generation data center switch ASICs.

2. Motivation and Design Choices

We start by explaining our motivation and justifying the design choices behind Expeditus in this section.

2.1 Motivation for Congestion-Awareness

ECMP—the *de facto* load balancing solution in practice—is well known to deliver poor performance, causing low throughput and bandwidth utilization for elephant flows and long tail latency for mice flows [3, 5, 7, 9, 12, 19, 20, 23, 33–35, 37]. In addition, it does not properly handle asymmetry in the network topology which often occurs in large-scale networks due to failures [38]. These problems are commonly attributed to the heavy-tailed nature of the flow size distribution. Thus, most solutions focus on attacking the tail, either by strategically routing elephant flows in a centralized [3] or distributed fashion [22] or splitting flows into sub-units of similar sizes [19, 23, 35].

While fixing the heavy tail improves performance, the main culprit is ECMP's *congestion agnostic* nature that fundamentally limits its ability to deal with inherent network dynamics. The congestion levels on the equal-cost paths are dynamic, making congestion-agnostic schemes, such as ECMP, ill-fitted for the job. For example, in a symmetric topology some paths may become temporarily congested due to hash collisions even with sub-flow level load balancing [19]. Moreover failure-induced topology asymmetry can arise any time in any part of the network and cause some paths to be persistently more congested than others.

This paper addresses the limitations of ECMP head-on, making a case for switch-based congestion-aware load balancing in light of the recent trend towards a smart data plane [4, 12, 21, 36]. An adaptive data plane is better suited to handle dynamic traffic and topological changes. In our design, switches dynamically choose less congested paths for each flow based on instantaneous congestion state, improving both throughput and tail latency.

2.2 Motivation for Scalable Design

We introduce background on 3-tier Clos topologies and motivate the need of a scalable design for congestion-aware load balancing in such topologies in this section.

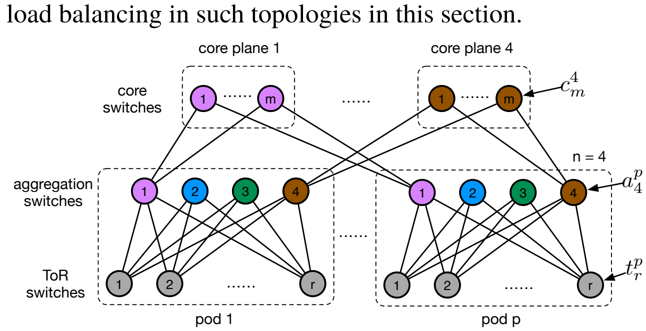


Figure 1: A general 3-tier Clos topology widely used in practice [8]. Not all switches or links are shown. There are p pods. Each pod has r ToR switches connecting to n aggregation switches. Here n is usually 4 as ToR switches have 4x40Gbps uplinks. There are m core switches in each of the n core planes that connect to the aggregation tier. Each switch has a unique ID configured as above.

3-tier Clos topologies [1, 8, 32] are prevalently used in production data centers for its scalability and cost-effectiveness.

A general 3-tier Clos is shown in Figure 1. Each pod has r ToR switches, each of which connects to n aggregation switches. An aggregation switch belongs to a unique core plane, and connects to all m core switches on its plane. Note each of the p pods is a 2-tier leaf-spine network. For ease of exposition, we use t (ToR), a (aggregation), c (core) to denote the switch type, superscript to denote its pod/plane number, and subscript to denote its ID; e.g., t_r^p is the r-th ToR switch in pod p, a_4^p the 4-th aggregation switch in pod p, and c_m^4 the m-th core switch on plane 4 as in Figure 1.

This highly modular topology provides flexibility to scale capacity in any dimension. When more compute capacity is needed, the provider adds server pods. When more inter-pod network capacity is needed, it can add core switches on all planes. Commodity ToR switches typically have 4×40Gbps uplinks. Thus, each pod normally has 4 aggregation switches. Aggregation/core switches can have up to 96×40Gbps ports. With 96 pods, the topology can accommodate 73,728 10Gbps hosts. In Facebook's Altoona data center, each aggregation switch connects to 48 ToR switches in its pod, and 12 out of the 48 possible core switches on its plane, resulting in a 4:1 oversubscription[8]. Google has been using 3-tier Clos topologies [32], and Amazon's EC2 cloud is also deploying 10Gbps fat-tree [1], which is a special case of the 3-tier Clos depicted above [2].

The key to designing a congestion-aware load balancing protocol is to acquire global congestion information for choosing the optimal paths. Existing work addresses this problem by maintaining congestion metrics for all the paths between all pairs of ToR switches. For example, in CONGA [4], each ToR switch keeps congestion information of the paths to all other ToR switches. The congestion metrics are obtained by piggybacking on data packets as they traverse the paths, and then fed back to the source ToR switch. This works well in 2-tier leaf-spine topologies, because the number of states to keep is relatively small; even in an extremely large (hypothetical) topology with 576 40Gbps ports paired with 48-port leaf switches [4], each leaf switch only needs to track ∼7K paths.

However, this presents significant scalability issues in 3-tier Clos topologies. In 3-tier topologies, even collecting congestion information for all paths is challenging because the number of paths to monitor dramatically increases (by two orders of magnitude). As shown in Figure 1, a 3-tier Clos has nm paths between a pair of ToR in different pods. Thus, each ToR needs to track $O(nmpr)$ paths for all possible destination ToR. For Facebook's Altoona data center with 96 pods, this amounts to ∼220K different paths. This is quite expensive to implement in switch ASICs. Furthermore, the information collected must reflect real-time status of each path, as congestion can change rapidly at timescales of a few RTTs due to bursty flows and exponential rate throttling in TCP [4, 5, 10, 21]. However, keeping the information up to date becomes even more challenging: the simple per-path feedback design requires at least $O(nmpr)$ concurrent flows to cover all the paths of a ToR *at the same time*, which is exceedingly difficult to achieve in practice.

This paper presents a practical congestion-aware load balancing scheme that solves the above challenges and can be applied to both 3-tier and 2-tier topologies. In the following, we discuss and justify the key design choices we make for Expeditus.

2.3 Design Choices

For congestion-aware load balancing to work in 3-tier topologies, two key components must scale with minimal overhead: an information collection mechanism that monitors the congestion state and a path selection mechanism that makes congestion-aware decisions.

Information collection. One strawman solution for addressing the scalability issue is to decompose a path into two pathlets: north-bound segment (from the source ToR to a core switch) and south-bound segment (the core switch to the destination ToR). Congestion information can be collected at the granularity of pathlets. Then only $O(nm)$ pathlet's congestion information is maintained at a ToR switch. However, a per-pathlet approach still requires $O(nm)$ concurrent flows from the core tier to the *same* ToR switch to cover all its pathlets and piggyback the information, which is difficult to achieve in practice. For example, a flow from t_1^1 to t_1^3 congests the link from c_2^2 to a_2^3 as shown in Figure 2. Now t_1^1 has a new flow to send to t_2^3. Since there is no flow on the pathlet from c_2^2 to t_2^3, t_2^3 cannot observe congestion nor inform the source ToR to avoid choosing c_2^2.

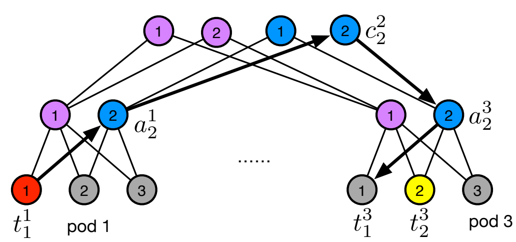


Figure 2: Per-pathlet congestion information collection is not timely enough. Here is a small Clos network with $n = m = 2$, $r = 3$. The link from c_2^2 to a_2^3 is congested due to a flow on the pathlet from c_2^2 to t_1^3 as shown. If t_1^1 has a new flow to send to t_2^3, as there is no flow on the pathlet from c_2^2 to t_2^3, t_2^3 cannot observe the congestion and inform t_1^1.

Design choice: Local information collection. We choose to rely on simple local congestion monitoring in Expeditus. Each switch only monitors the egress and ingress congestion metrics of all its uplinks. The scalability challenge is resolved by only maintaining $2n$ states at each ToR switch for the n uplinks connected to the aggregation tier, and $2m$ states at each aggregation switch for the m uplinks connected to the core tier. Real-time congestion state can be readily gathered and updated whenever a packet enters and leaves the switch, and does not require any piggybacking.

Path selection. With local information collection, we now need to aggregate information across the network in order to obtain the path-wise congestion. A strawman solution is then for the source ToR to ask all the $2n$ aggregation switches in the source and destination pods and the destination ToR for their local information. In total $2n+2$ switches have to participate in the process. This again does not scale well for large networks. Further, coordinating so many switches across four hops adds much implementation and latency overhead to the protocol.

Design choice: Two-stage path selection. We present a novel two-stage path selection mechanism to efficiently aggregate information with minimal overhead. Our design exploits a salient structural property of 3-tier Clos, which we call *pairwise connectivity*:

In 3-tier Clos, an aggregation switch of ID i only connects to aggregation switches of the same ID i in other pods, because they connect to the same core plane. Thus, no matter which aggregation switch the source ToR chooses, the packet always goes via an aggregation switch of the same ID in the destination pod (recall Figure 1).

Based on pairwise connectivity, we propose a two-stage path selection mechanism that works as shown in Figure 3. There are four hops between any two ToRs in different pods. Our scheme first selects the best aggregation switches (the first and last hop) using the congestion information of the first and last links, and then determines the best core switch (the second and third hop) using the congestion information of the second and third links. The process starts when the first packet of a flow arrives at a ToR. The source ToR tags the first packet to add the first-hop information, i.e. the egress congestion metrics of its uplinks. This data packet then serves as an Expeditus request packet (Exp-request in short). The destination ToR reads the congestion metrics from the Exp-request, aggregates with its ingress metrics of all its uplinks, and chooses the least congested aggregation switches. A response packet called Exp-response is then generated by the destination ToR and sent to the chosen destination aggregation switch. The destination aggregation switch feeds back the third-hop congestion metrics via Exp-response to the source aggregation switch, which similarly chooses the core switch with the least effective congestion before forwarding the Exp-response to the source ToR. Thus, Expeditus completes path selection with two messages (Exp-request and response) in just one round trip. The path selection decisions are maintained at the source ToR and aggregation switches.

This design requires only two switches at each stage to exchange information. For a new TCP connection, Expeditus selects paths for the two flows in both directions independently during the handshaking process and does not cause any packet reordering. Intuitively, two-stage path selection is a heuristic since it does not explore all available paths: it simplifies the problem from choosing the best combination of aggregation and core switch IDs to choosing the two

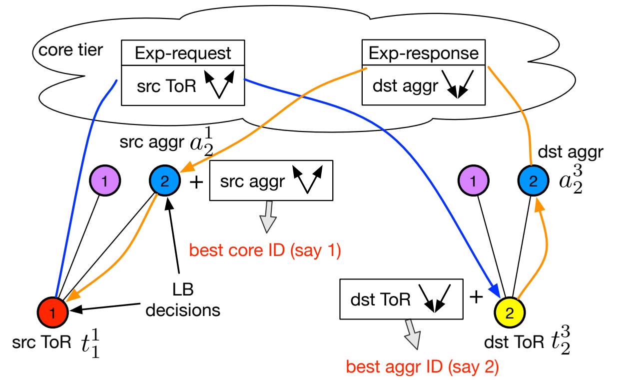


Figure 3: Design of two-stage path selection. The first stage is between the source ToR (t_1^1) and destination ToR (t_2^3) who chooses the best aggregation switch ID based on first and last hop congestion metrics. The the second stage involves the chosen destination and source aggregation switch (a_2^1) who chooses the best core switch ID. The information exchange is done via Exp-request and Exp-response.

sequentially. Nevertheless, by taking advantage of pairwise connectivity, it performs effectively compared to a clairvoyant scheme with complete information in 3-tier Clos as shown in §5.

3. Expeditus Design

Expeditus runs entirely in the data plane as a distributed protocol, with all functionalities residing in switches. It works on a per-flow basis for implementation simplicity, and uses link load as the congestion metric, which is shown to be effective and can be readily implemented in hardware [4].

3.1 Local Congestion Monitoring

In Expeditus only ToR and aggregation switches perform local congestion monitoring for all uplinks that connect them to upstream switches. This covers the entire in-network path without the links connecting the host and the ToR, which are unique and not part of load balancing. Both the egress and ingress directions are monitored.

Link load estimation. To measure the link load, we use Discounting Rate Estimator (DRE) [4]. DRE maintains a register X which is incremented every time a packet is sent/received over the link by the packet size in bytes, and is decremented every T_{dre} with a factor of α between 0 and 1. We follow the settings in [4] and set T_{dre} =20 µs and $\alpha = 0.1$. The link load is quantized into 3 bits relative to the link speed.

Congestion table. Link load information is maintained in a congestion table. In cases of link failures, the switch detects that a port is down, and simply sets the utilization of that port to the largest value.

3.2 Two-stage Path Selection

We now explain the working of two-stage path selection, the most important mechanism in Expeditus.

Path selection table (PST). Path selection decisions are maintained in a PST in each ToR and aggregation switch.

Only the northbound pathlet of a flow's path needs to be recorded, for there is a unique southbound pathlet from a given core or aggregation switch to a given ToR. Each PST entry records a flow ID obtained from hashing the five-tuple, the egress port selected, a path selection status (PSS) bit indicating whether path selection has been completed (1) or not (0), and a valid bit indicating whether the entry is valid (1) or not (0).

When a packet arrives, we look up an entry based on its flow ID. If an entry exists and is valid, the packet is routed to the corresponding egress port. If the entry is invalid and the PSS bit is one, or no entry exists, then the packet represents a new flow and starts a new round of path selection. The valid and PSS bits are set when a path has been selected. The PSS bit ensures path selection is performed only once; an invalid PST entry with a zero PSS bit does not trigger path selection when subsequent packets for the flow, if any, arrive before path selection completes (they are simply forwarded by ECMP).

PST entries time out after a period of inactivity in order to detect inactive flows and force a new path to be selected. When an entry times out the valid bit is also reset. Since we focus on per-flow load balancing, the timeout value is set to 100ms which is large enough to filter out bursts of the same flow. The timer can be implemented using just one extra bit for each entry and a global timer for the entire PST [4].

The implementation cost of tracking per-flow state using PST is low for a data center network, even at scale. As reported in [4], the number of concurrent flows would be less than 8K for an extremely heavily loaded switch. A PST of 64K entries should cover all scenarios.

Ethernet tags. Expeditus uses dedicated Ethernet tags on IP packets, similar to IEEE 802.1Q VLAN tagging, to exchange congestion information between switches during path selection. Specifically, a new field is added between the source MAC address and the EtherType/length fields. The structure of the new field is shown in Figure 4, and includes the following:

- **Tag protocol identifier, TPID (16 bits):** This field is set to 0x9900 to identify the packet as an Expeditus path selection packet.
- **Stage flag, SF (1 bit):** This field identifies which stage this packet serves (0 for first stage, 1 for second stage). A tagged packet for the first stage is called an "Exp-request", and for the second stage an "Exp-response."
- **Number of entries, NoE (7 bits):** This field identifies the number of congestion metrics carried in the packet (at most 128). Note 7 bits are more than enough in practice. Between the ToR and aggregation tiers, the Facebook Altoona topology has just 4 paths, and a 128-pod fat-tree with 524,288 hosts has 64 paths; between the aggregation and the core layer, the former has 48 paths and the latter 64 paths.
- **Congestion data, CD:** This field contains the actual congestion metrics. We use 4 bits for one entry with the first bit as padding.

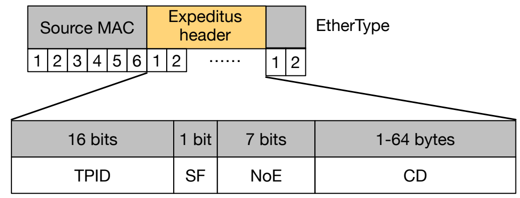


Figure 4: Format of the Expeditus tag for path selection.

Path selection algorithm. Figure 5 illustrates our two-stage path selection for two flows in the topology of Figure 2. There is a new TCP connection between hosts under ToR switches t_1^1 and t_2^3. Flow A is the flow in the forward direction from t_1^1 to t_2^3 that initiates the SYN, and flow B is in the reverse direction. Both flows use the two-stage path selection mechanism *independently* to establish their paths.

Let us start with flow A. Its first packet (SYN in this case) reaches its source ToR t_1^1 and triggers the path selection mechanism after checking the PST. t_1^1 tags the packet with its egress link loads of the uplinks, and sets SF to 0 and NoE correspondingly. It forwards the tagged packet, i.e. the Exp-request, using ECMP, and inserts a new entry into the PST with PSS bit set to 0. Aggregation switches ignore Exp-request packets and simply forward with ECMP. The destination ToR switch t_2^3 reacts to Exp-request. It checks the NoE field, pulls the congestion information, and aggregates it entry-by-entry with its ingress link loads (recall pairwise connectivity §2.3). The effective congestion of all n paths between the source ToR and the aggregation tier are determined simply as the maximum load of the two hops. t_2^3 then chooses the aggregation switch ID with the minimum effective congestion, which is 2 in this case. t_2^3 generates an Exp-response without payload by copying the TCP/IP header from the Exp-request and swapping the src and dst IP addresses. It tags the Exp-response with SF set to 1 and forwards to the chosen aggregation switch a_2^3. It also removes the tag from the Exp-request and forwards to the host. This completes the first stage of path selection.

The second stage is similar and involves the aggregation switches choosing the path to the core tier using the Exp-response. The aggregation switch a_2^3 handles the Exp-response with NoE set to zero, by adding its ingress loads and setting NoE accordingly. The source aggregation switch a_2^1 reacts to the Exp-response with a non-zero NoE value by comparing a_2^3's ingress loads with its egress loads, and choosing the best core switch ID (1 in this case). It computes flow A's ID by swapping the src and dst IP addresses, inserts a new PST entry or updates an existing entry for flow A, records 1 as the path selection result (by pairwise connectivity), and sets both PSS and the valid bit to 1. Finally, the source ToR t_1^1 receives the Exp-response, matches it with flow A's entry in

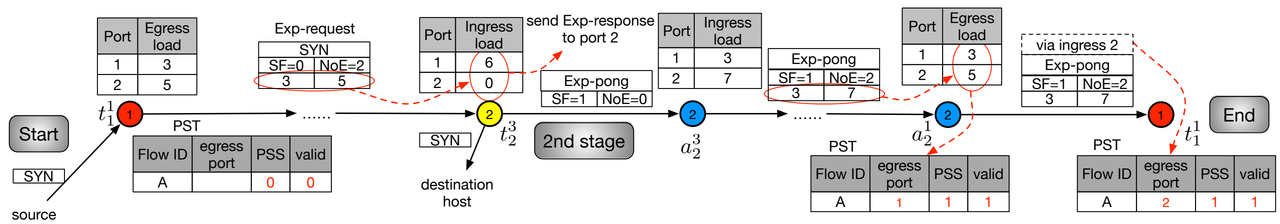


Figure 5: Two-stage path selection in Expeditus for the network shown in Figure 2. We only show the path selection process for flow A from the source ToR t_1^1 to t_2^3. Path selection for flow B of the same TCP connection in the reverse direction is done similarly.

PST, records its ingress port (2 in this case) in the entry as the path selection result, sets both PSS and the valid bit to 1, and discards the Exp-response. This finishes flow A's path selection.

Flow B's path selection is done in exactly the same way. The only difference is that the first packet of B is a SYN-ACK in this case. It arrives at B's source ToR t_2^3 and starts the process. Clearly the selected aggregation and core switches may be different from those for flow A.

Now to summarize, for the new TCP connection in our example, flow A's path is established before the SYN-ACK,[1] and before its second packet (the first ACK) arrives at the source ToR. Flow B's path is selected before the handshaking ACK, and thus before its second packet arrives. Therefore no packet reordering is caused by Expeditus, when path selection is done during TCP handshaking. Algorithms 1 and 2 rigorously present the path selection and packet processing logic for ToR and aggregation switches.

Lost Exp-request/Exp-response. As discussed we do not have any retransmission mechanism, in case the control packets are dropped. This simply means one opportunity of load balancing is lost; it does not affect the flow at all. There are two possibilities: (1) the path is not established at all. Expeditus is robust in this case since any packet that sees an invalid entry is routed using ECMP; (2) part of the path is established at the aggregation switch, but not at the ToR switch. Expeditus is still robust since the PST entry at the aggregation switch will just time out. If we are fortunate and ECMP at the ToR switch happens to direct the flow to the chosen aggregation switch, we can still use the least congested path.

3.3 Handling Failures

Failures are the norm rather than exception in large-scale networks with thousands of switches. Expeditus automatically routes traffic around the congestion caused by failures, thanks to the congestion-aware nature, delivering better performance over congestion-agnostic schemes.

We illustrate this using an example in Figure 6. Suppose the link from a_1^1 to c_1^1 is down. This causes all links from c_2^1 to the first aggregation switches of each pod (dashed links) to be congested, as they are the only paths to reach a_1^1. For example, if there are flows from t_1^2 to t_1^1, the a_1^2-c_2^1 link is more

[1] The Exp-response is generated immediately upon receiving the Exp-request.

Algorithm 1 Expeditus, ToR switch

procedure ToR_Switch_Processing(packet p)
  if p is northbound to the aggregation tier **then**
    if a PST entry e exists for p **then**
      if e.valid_bit == 1 **then**
        forward p according to e, **return**
      else if PSS == 0 **then**
        forward p by ECMP, **return**
    add the Expeditus tag to p ▷ Start path selection
    SF ← 0, add the egress loads, forward p by ECMP
    insert a new PST entry or update the existing one for p, PSS ← 0, valid_bit ← 0, **return**
  else ▷ southbound packet
    if p is Expeditus tagged **then**
      if p.SF == 0 **then** ▷ Exp-request received
        check NoE, pull CD
        choose the best aggregation switch ID $f*$
        generate an Exp-response p', p'.SF← 1, p'.NoE ← 0
        p'.src_ip ← p.dst_ip, p'.dst_ip ← p.src_ip
        forward p' to aggregation switch $f*$
        remove the tag from p, forward it, **return**
      else ▷ Exp-response received
        record p's ingress port p_i
        find the PST entry e for the flow f, f.src_ip=p.dst_ip, f.dst_ip=p.src_ip
        e.egress_port ← p_i, e.PSS ← 1, e.valid_bit ← 1, discard p, **return**

congested than the two links from a_2^2 to the core tier. Now traffic from a_1^2 to other pods, say pod 3, will be routed to c_1^1 in order to avoid the congested link by Expeditus. In contrast, ECMP still evenly distribute traffic and further congest c_2^1.

Yet, since the two-stage path selection only utilizes partial information of a path, it can make sub-optimal decisions for certain flows especially with topological asymmetry. Consider the same example again. Suppose there is traffic from pod 1 to pod 2. Due to the failure a_1^1 suffers a 50% bandwidth reduction, and uplinks from ToRs to a_1^1 (red lines) cannot achieve their full capacity when transmitting inter-pod traffic. Thus these uplinks are actually more likely to be selected in favor of their low loads, which exacerbates congestion on a_1^1 and c_2^1.

To address this issue, we set *link load multipliers* for ToR uplinks based on the effective capacity at aggregation

Algorithm 2 Expeditus, aggregation switch

procedure AGGR_SWITCH_PROCESSING(packet p)
  if p is northbound to the core tier **then**
    if p is Expeditus tagged, p.SF == 1 **then** ▷ Exp-response, first hop
      add the switch's ingress loads to p, set p.NoE
    if a PST entry e exists for p **then**
      if e.valid_bit == 1 **then**
        forward p according to e, **return**
    forward p by ECMP, **return**
  else ▷ southbound packet
    if p is Expeditus tagged, p.SF == 1, p.NoE is non-zero **then**
    ▷ Exp-response, third hop
      check NoE, pull CD
      choose the best core switch ID $f*$
      record port p_i connected to core switch $f*$
      find the PST entry e for the flow f, f.src_ip=p.dst_ip, f.dst_ip=p.src_ip, or insert a new entry if not found
      e.egress_port ← p_i, e.PSS ← 1, e.valid_bit ← 1
    forward p, **return**

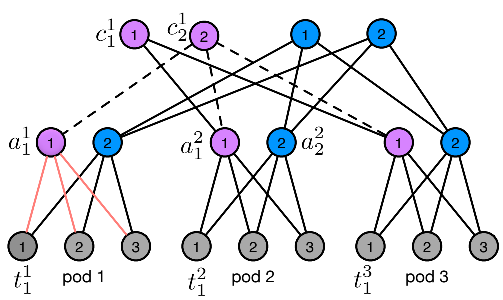


Figure 6: The link between a_1^1 and c_1^1 fails. This causes all links from c_2^1 to the first aggregation switches of each pod to be congested, as they are the only paths to reach a_1^1. A link load multiplier of 2 is applied to links from a_1^1 to each ToR in pod 1 as shown for inter-pod traffic.

switches. This effectively makes network bottlenecks visible at the ToR switches. We assume the underlying routing protocol or the control plane notifies ToR switches of the aggr-core link failure [19, 38]. The ToR switches then set a link load multiplier of 2 for the uplink to a_1^1 as highlighted in Figure 6. The multipliers only affect inter-pod traffic at ToRs. The link loads are scaled with the multipliers when they are used in the first stage of path selection. Effectively, this translates the capacity reduction at the aggregation layer proportionally to the ToR layer. In this way, ToRs are more likely to choose uplinks to a_2^1 and re-distributes traffic properly.

3.4 Discussions

2-tier leaf-spine topologies: We briefly explain how Expeditus works in 2-tier leaf-spine topologies. This also applies to intra-pod traffic in a 3-tier Clos. The two-stage path selection naturally reduces to one-stage path selection because only the ToR switches are involved. The Exp-request carries egress loads of the source ToR, and the destination ToR aggregates it with its ingress loads to obtain the *end-to-end* congestion information for *all* possible paths, and chooses the best one. Aggregation switches ignore southbound Exp-response with a zero NoE and simply forward it. The chosen aggregation switch ID is obtained as the Exp-response's ingress port when it reaches the source ToR.

Handling UDP traffic: Expeditus can be directly used without change to load balance UDP flows. The first packet of a new UDP flow (without a PST entry or with an invalid PST entry) triggers path selection. Packets sent before the path is established are forwarded by ECMP, and UDP does not suffer from re-ordering.

4. Implementation and Testbed Evaluation

We implement Expeditus on Click and run our prototype on a small-scale Emulab testbed to answer the following questions: (1) Is it practical to implement Expeditus, and how much overhead does it introduce (§4.1)? (2) How does Expeditus actually perform in practice (§4.2)?

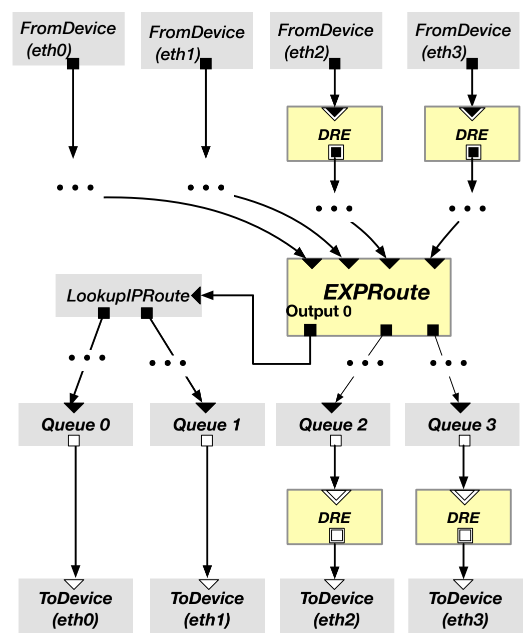


Figure 7: Packet processing pipeline in Click.

Prototype implementation: We implement a prototype of Expeditus in Click, a modular software router for fast prototyping of routing protocols [25]. We develop two new Click elements, `DRE` to measure link load and `EXPRoute` to conduct two-stage path selection. A `DRE` element is added to the `FromDevice` and `ToDevice` elements of each port in the Click configuration. `EXPRoute` can obtain ingress/egress link loads from upstream/downstream `DRE` elements. Figure 7 shows the packet processing pipeline of Expeditus for a ToR or aggregation switch with 4 ports.

Note that unlike actual routers whose timers have microsecond resolution in hardware, Click can only achieve millisecond resolution. This affects accurate estimation of link load, which makes DRE react slower to link load changes in our testbed. To compensate for this, we additionally use the queue length information together with the DRE working at T_{dre} of 1 ms.

Testbed setup: Our Emulab testbed uses PC3000 nodes to host Click routers, with 64-bit Intel Xeon 3.0GHz processors, 2GB DDR2 RAM, and 4 1GbE NICs. All nodes run CentOS 5.5 with a patched Linux 2.6.24.7 kernel and a patched Intel e1000-7.6.15.5 NIC driver to improve Click performance. We use the default TCP Cubic implementation in the kernel. Our Click implementation is running in kernel space, and we verify that TCP throughput between two Click routers is stable at 940+ Mbps.

Constrained by the number of NICs an Emulab node has, we setup a small-scale 3-tier Clos network with 2 pods of 2 aggregation and ToR switches each, i.e. $n = m = r = p = 2$. Each aggregation switch connects to 2 core switches, and each ToR switch connects to 2 hosts . The core tier is oversubscribed at 4:1 by rate limiting the core links to emulate a realistic setting in practice [8].

4.1 Microbenchmarks of Overheads

Firstly, to demonstrate viability, we evaluate the packet processing overhead of Expeditus. Table 1 shows the average CPU time of forwarding a packet through each element at the source ToR switch sending at line rate, measured with Intel Xeon cycle counters. We report the average value obtained from the total processing time divided by the number of packets (∼550k). For comparison, we implement a `HashRoute` element to perform ECMP and measure its processing time. The additional overhead incurred by `ExpRoute` and `DRE` elements is only hundreds of nanoseconds. This is comparable to the overhead of a vanilla IP forwarding element [25].

Element	ns/pkt
`HashRoute`	275
`EXPRoute`	473
`DRE`	151

Table 1: Average packet processing time comparison.

Scheme	Avg Time (us)
`ECMP`	623.90
`Expeditus`	638.33

Table 2: Average time it takes for the sender to start sending the first data packet after sending SYN.

We also look at the latency overhead of two-stage path selection. We start a TCP connection and measure how long it takes for the sender to start sending the first data packet, by which time path selection for both directions are done and cannot affect the flow anymore. As in Table 2, Expeditus adds a negligible 15 µs delay on average over 100 runs. This overhead is mainly due to the additional processing in the `EXPRoute` element. The latency overhead will be much smaller if Expeditus is implemented in ASIC.

4.2 Testbed Evaluation

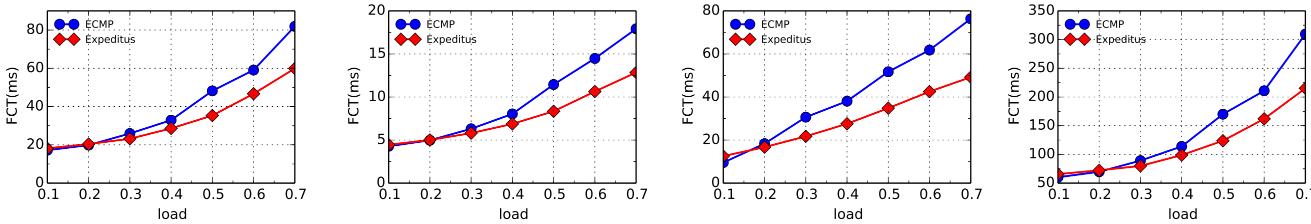
We conduct experiments to evaluate Expeditus's performance with two realistic workloads from production datacenters. The first is from a cluster running mostly web search as reported in [5]. The second is from a large cluster running data mining jobs [17]. Both distributions are heavy-tailed: in the web search workload, over 95% of bytes are from 30% flows larger than 1MB; in the data mining workload, 95% of bytes are from about 3.6% flows that are larger than 35MB, while more than 80% of flows are less than 10KB. We generate flows between random senders and receivers in different pods according to Poisson processes with varying arrival rates. Our setup is consistent with existing work [4, 9, 19].

Figures 8 and 9 show the FCT results of Expeditus and ECMP for both workloads. We vary loads from 0.1 to 0.7 beyond which the results become unstable in our testbed. We emphasize FCT statistics for mice (<100KB) and elephant flows (>1MB). The results of medium flows between 100KB and 1MB are largely in line with elephant flows, and we do not show them for brevity. Each data point is the average of 3 runs.

We make the following observations: First, for both workloads, Expeditus outperforms ECMP in both average and 95%ile tail FCT for mice flows. For loads between 0.4 and 0.7, Expeditus reduces the average FCT by ∼14%–30% in the web search workload and 17%–25% in the data mining workload. The reduction in tail FCT is even larger: by ∼30%–45% in web search workload and 5%–30% in data mining workload. Second, Expeditus also improves throughput for elephant flows in medium and high loads. The reduction in average FCT is 9%–38% for web search workload and 11%–18% for data mining workload. The average FCT is much longer in data mining workload as the elephant flows are much larger (mostly >10MB) than those in web search workload. Across all flows, Expeditus reduces the average FCT by 12%–32% and 15%–22%, respectively, for the two workloads with loads from 0.4 to 0.7. Third, when network load is light (<0.4), Expeditus provides relatively small benefit compared to ECMP. This is mainly due to the limitation of our Click-based implementation in detecting transient congestion. Our implementation updates the link utilization using Click timers that operate at coarse-grained timescales (1ms granularity). However, in lightly loaded networks, congestion is much more transient and often happens at small timescales. In our simulations where the link utilization is updated frequently, we observe that the benefit of Expeditus does not degrade even when the network is lightly loaded (§5.3).

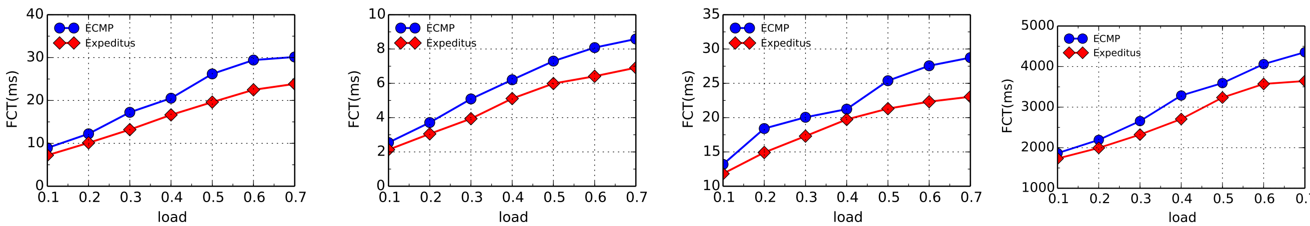
5. Large-scale Simulation

We conduct extensive packet-level ns-3 simulations to evaluate Expeditus in large-scale networks. We seek to answer the following key questions: (1) How does Expeditus perform under various traffic patterns at scale (§5.2, §5.3)? (2) Is Expeditus sensitive to network bottleneck conditions (§5.4)?



(a) Overall: Avg **(b)** (0,100KB]: Avg **(c)** (0,100KB]: 95-th percentile **(d)** (1MB, ∞): Avg

Figure 8: FCT for web search workload in the Emulab testbed. Core tier oversubscription 4:1.



(a) Overall: Avg **(b)** (0,100KB]: Avg **(c)** (0,100KB]: 95-th percentile **(d)** (1MB, ∞): Avg

Figure 9: FCT for data mining workload in the Emulab testbed. Core tier oversubscription 4:1.

(3) Is Expeditus effective in handling topological asymmetry due to failures (§5.5)? (4) How does it perform in leaf-spine networks compared to existing work (§5.6)?

5.1 Methodology

Simulation Setup. We use a 12-pod fat-tree [2] as the baseline topology. As mentioned in §2.2 fat-tree is a 3-tier Clos topology. There are 432 hosts and 36 core switches, i.e. 36 equal-cost paths between any pair of hosts in different pods. Each ToR switch has 6 ports to the aggregation tier and hosts, respectively. All links are running at 10Gbps. We vary the number of core switches to obtain different oversubscription ratios at the core tier, for production networks often oversubscribe the core to reduce costs [8]. The baseline oversubscription ratio is 2:1. The fabric RTT is ∼32 µs across pods. Each run of the simulation generates more than 10K flows, and we report the average over 5 runs unless otherwise noted.

We use several different traffic patterns.

Realistic: We use the same realistic workloads from production datacenters as in our testbed experiments: a web search workload [5] and a data mining workload [17]. Each host continuously generates flows based on the empirical flow size distributions and sends to a random receiver. The flow inter-arrival time follows an exponential distribution.

Stride: We index the hosts from left to right. Server[i] sends to server[(i+M) mod N] where M is the number of hosts in a pod and N the total number of hosts.

Bijection: Each host sends to a random destination in a different pod. Each host only receives data from one sender.

Random: Each host sends to a random destination not in the same pod as itself. Different from bijection, multiple hosts may send to the same destination.

We mainly use the Realistic workload. The other three workloads are traditional synthetic communication patterns that are only used to perform stress tests, similar to previous work [2, 3, 19]:

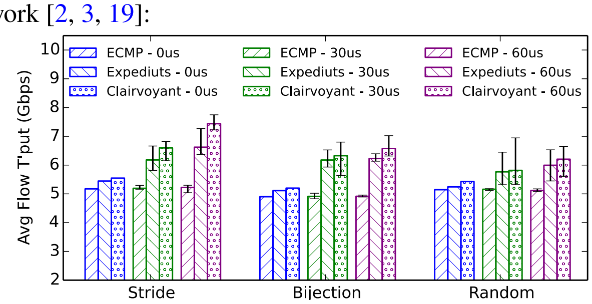
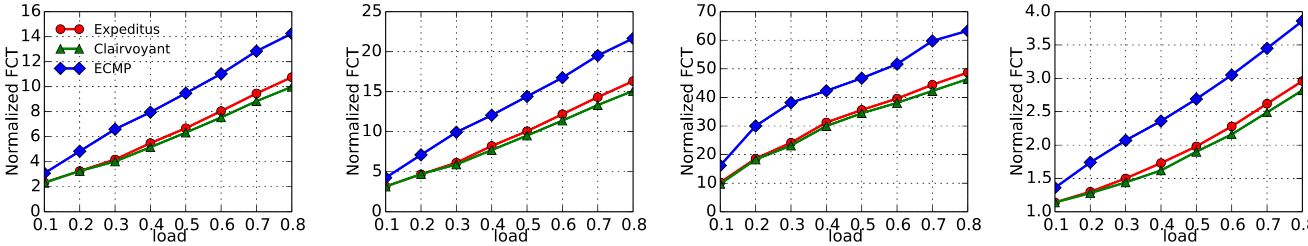


Figure 10: Stress test results for different traffic patterns and mean inter-arrival times in a 4-pod 10Gbps fat-tree.

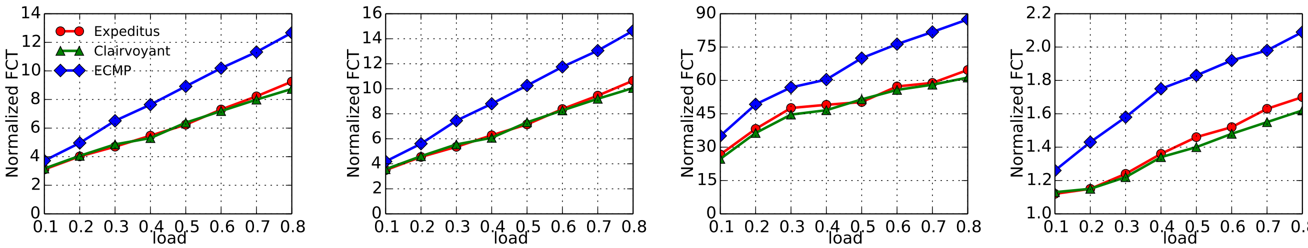
Benchmarks. We compare four load balancing schemes.

- *Expeditus*: Our distributed congestion-aware protocol.
- *Clairvoyant*: An "ideal" congestion-aware scheme that uses complete end-to-end information of all possible paths between the two pods. Whenever a new flow comes, the source ToR chooses the best path. As discussed in §2.2 such a scheme introduces prohibitive overhead in 3-tier Clos for practical implementation.
- *CONGA and CONGA-Flow*: The state-of-the-art distributed congestion-aware protocol. CONGA is integrated with flowlet switching [23]. Flowlets are detected when the inter-arrival time between two consecutive packets exceeds a certain threshold T_{fl}. Our implementation is faithfully based on all details provided in [4]. CONGA uses T_{fl} =100 µs to split flows into flowlets. Different from CONGA, CONGA-Flow makes per-flow load balancing decisions and guarantees packet ordering. Both are applied in leaf-spine topologies only.
- *ECMP*: This is our baseline.



(a) Overall: Avg **(b)** (0,100KB]: Avg **(c)** (0,100KB]: 95-th percentile **(d)** (1MB, ∞): Avg

Figure 11: Normalized FCT for web search workload in the baseline 12-pod fat-tree with core tier oversubscription of 2:1.



(a) Overall: Avg **(b)** (0,100KB]: Avg **(c)** (0,100KB]: 95-th percentile **(d)** (1MB, ∞): Avg

Figure 12: Normalized FCT for data-mining workload in the baseline 12-pod fat-tree with core tier oversubscription of 2:1.

5.2 Stress Tests with Synchronized Flows

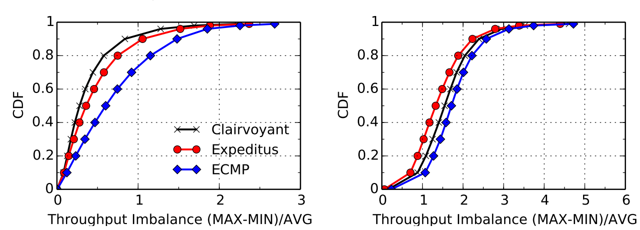
We first perform stress tests. We generate synchronized flows with the three synthetic traffic patterns. From each sender, we generate a flow of 50MB. To vary the degree of synchronization, we conduct three sets of simulation with flow inter-arrival times following exponential distributions with means 0, 30 µs, and 60 µs.

Figure 10 shows the average throughput for different schemes with error bars over 5 runs. When all flows start at the same time, Expeditus and Clairvoyant perform on par with ECMP. This is expected as all paths are idle at the same flow arrival time, and congestion-aware schemes essentially reduce to ECMP. Clearly perfectly synchronized flow arrivals seldom happen in practice. When flows are loosely synchronized, Expeditus is able to choose better paths than ECMP and improve performance substantially. It improves the average throughput by ∼23%–42% for Stride and Bijection, and ∼20% for Random with mean inter-arrival times of 30 µs and 60 µs. The Random traffic pattern is challenging for Expeditus as multiple senders may send to the same receiver, in which case throughput is limited at the edge of the network. Expeditus is very close to Clairvoyant with a performance gap of at most 9%.

5.3 Baseline Performance in 3-tier Clos

We now investigate the performance of Expeditus in a large-scale Clos network. We use the Realistic traffic pattern with network load varying from 0.1 to 0.8. Figure 11 and Figure 12 show the normalized FCT (NFCT) for both realistic workloads in the baseline 12-pod fat-tree with core tier oversubscribed at 2:1. NFCT is the FCT value normalized to the best possible completion time achieved in an idle network where each flow can transmit at the bottleneck link capacity [4, 7].

We observe that for mice flows, Expeditus provides 16%–38% FCT reduction at the 95%ile over ECMP for both workloads. Meanwhile, for elephants, Expeditus is also ∼15%–30% faster on average. Moreover, it is clear that performance of Expeditus closely tracks that of Ideal in both workloads. In most cases the performance gap is less than 10%, demonstrating the near-optimality of the heuristic path selection design.



(a) Aggregation tier imbalance. **(b)** ToR tier imbalance.

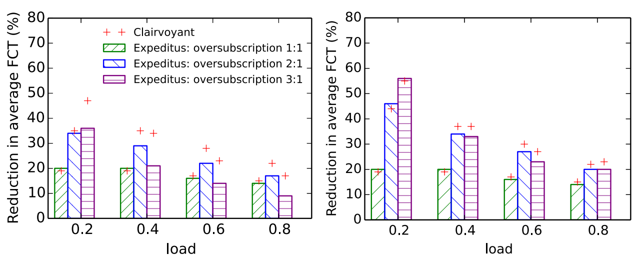
Figure 13: CDF of throughput imbalance at different tiers of a 12-pod fat-tree with 2:1 core tier oversubscription for the web search workload at 50% load.

Load balancing efficiency. To further understand Expeditus's benefits, we assess its load balancing efficiency over ECMP. Figure 13 shows the CDF of *throughput imbalance* across uplinks of a ToR or aggregation switch in the baseline topology for the web search workload at 50% load. The throughput imbalance is defined as the maximum throughput minus the minimum divided by the average among all uplinks of a switch: (MAX − MIN)/AVG. This is calculated from synchronous samples of each uplink at a randomly chosen ToR/aggregation switch over 1ms intervals, with more than 2K sample points in total.

We make two observations. First, the results confirm that Expeditus is more efficient in balancing loads across the network than ECMP. It reduces the average throughput imbalance significantly compared to ECMP. Note the imbalance is at most 3 at the core tier instead of 6 as in the ToR tier, because the core tier is oversubscribed at 2:1 in the baseline topology. The second observation is that Clairvoyant balances loads better at the aggregation tier, while Expeditus performs better than Clairvoyant at the ToR tier. This is mainly due to the two-stage path selection design. Expeditus first chooses the best path from the ToR to the aggregation tier, without considering the aggregation-core links. This naturally results in a better balanced ToR tier. On the other hand Clairvoyant uses global information and can balance the congested aggregation tier better.

5.4 Impact of Network Bottleneck

In this section we evaluate the impact of network bottleneck on Expeditus. We vary the severity and location of bottlenecks by varying the oversubscription ratios at different tiers of the topology.



(a) Oversubscription at the core tier.

(b) Oversubscription at the ToR tier.

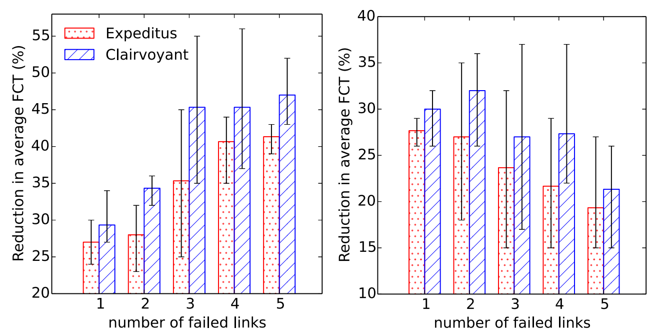
Figure 14: Average FCT reduction by Expeditus over ECMP for all flows with varying oversubscription at different tiers of a 12-pod fat-tree for the web search workload.

We first consider the impact of bottleneck severity. Figure 14a shows Expeditus's average FCT improvement over ECMP for all flows in the baseline topology with varying oversubscription ratios at the core tier. Only results for web search workload is shown for brevity. In general, we find that Expeditus provides more benefits with more oversubscription at first, and then the improvements decrease with an oversubscription of 3. This is because when the network is heavily oversubscribed, many elephant flows consistently occupy the paths, diminishing the congestion diversity across equal-cost paths, as well as the benefit of being congestion-aware.

Next we consider the impact of bottleneck location. Figure 14b shows the result when the ToR tier is oversubscribed with more hosts. Now uplinks of ToR switches are the bottleneck instead of aggregation switches. Interestingly, we observe that Expeditus performs even better in this setting, and the performance gap between Expeditus and Clairvoyant is even smaller. The reason is as follows. Recall that Expeditus always chooses paths starting from the ToR tier. Thus it works better when the ToR tier is the bottleneck. When instead the core tier is bottlenecked, it is more likely that the first stage path selection directs a flow to an aggregation switch that connects to congested core switches, and Expeditus is unable to change the decision in the second stage.

5.5 Impact of Link Failures

In this section, we study the impact of link failures and topology asymmetry. To emphasize the performance of victim flows, the following experiments involve only two pods in the 12-pod non-oversubscribed fat-tree. We vary the number of failed links. We choose links in one pod to fail uniformly at random in each run, with each switch having at most 2 failed links. Only results with the web search workload are shown. Figure 15 shows the overall average FCT reduction of both Clairvoyant and Expeditus over ECMP for all flows at 0.5 load. We observe qualitatively similar results for other loads.



(a) Aggr-core link failure.

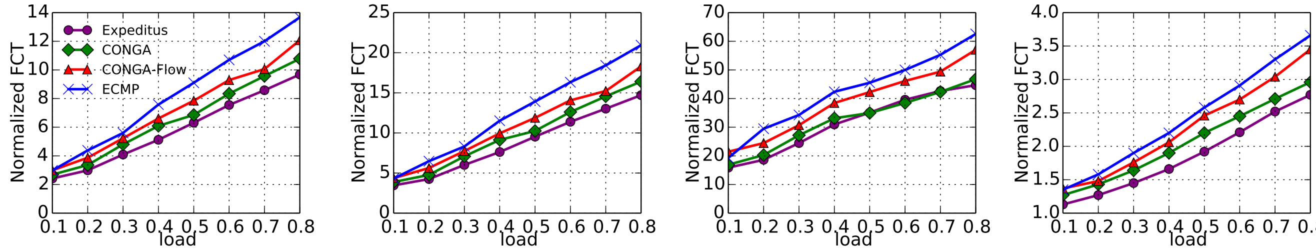
(b) ToR-aggr link failure.

Figure 15: Average FCT reduction by Clairvoyant and Expeditus over ECMP for all flows with link failures.

We consider two scenarios: aggr-core and ToR-aggr link failures. Observing from Figure 15a that when aggr-core links fail, performance gains of Expeditus and Clairvoyant become more significant with more failures. This is because ECMP always hashes flows evenly to all paths without considering the asymmetric uplink bandwidth, thus aggravating the congestion. Expeditus proactively detects high utilization links due to failures using the link load multipliers, and diverts traffic away from hot spots to balance the load. Figure 15b shows the result when failures happen at the ToR-aggr links. Comparing with aggr-core link failures, a ToR-aggr link failure causes more network capacity loss because one aggregation switch becomes completely unaccessible. This leaves less room for load balancing to improve. This explains why performance gains of Expeditus and Clairvoyant diminishes with more ToR-aggr link failrues. Still, across different scenarios Expeditus provides moderate (20%–27%) performance gains. In all, Expeditus is particularly robust against failures in 3-tier Clos networks.

5.6 Performance in Leaf-spine Topologies

We finally turn to 2-tier leaf-spine topology, and compare Expeditus to CONGA, CONGA-Flow and ECMP. Expeditus



(a) Overall: Avg **(b)** (0,100KB]: Avg **(c)** (0,100KB]: 95-th percentile **(d)** (1MB, ∞): Avg

Figure 16: Normalized FCT for web search workload with 8 spine and 8 leaf switches. Network oversubscription 2:1.

is equivalent to Clairvoyant in leaf-spine topologies (§3.4). The topology has 8 leaf switches, 8 spine switches, and 128 hosts. Note that we double the number of hosts to achieve 2:1 oversubscription at the leaf level.

Figure 16 shows the FCT results for the web search workload. Expeditus achieves favorable performance gains over ECMP, ranging from 18%–32% for all flows across all loads. More interestingly, it provides comparable performance as CONGA and outperforms CONGA-Flow in all cases. The trend is consistent with results in CONGA [4]. It is intuitive that CONGA achieves good performance since it benefits from fine-grained load balancing: flowlet switching. The main reason for the degraded performance of CONGA-Flow is that CONGA-Flow can only observe the congestion state of the paths when there are flows traversing on them. As there may not be enough concurrent flows to cover all paths at all times (§2.3), CONGA-Flow fails to observe the up-to-date, global state of the network.

6. Related Work

We discuss related work other than CONGA now. Prior work in general falls into two categories to grapple with ECMP's drawbacks. The first is centralized routing that pins elephants to good paths based on global network state [3, 10, 13, 31]. This approach poses scalability challenges for OpenFlow switches [13], and is too slow to improve latency for mice flows. The second approach is fine-grained load balancing. Packet spraying adopts per-packet load balancing [11, 14, 16], which induces packet reordering and interacts poorly with TCP. Flare [23], LocalFlow [31], and Presto [19] split a flow into many small bursts, and load balances them to avoid or alleviate packet reordering. FlowBender [22] opportunistically re-routes flows by simple hacks to commodity switches. These approaches only use *local* information at best, which is sub-optimal in handling flash congestion in the network.

A recent independent work also propose a congestion-aware load balancing protocol called HULA for Clos topologies [24]. Similar to distance-vector routing, congestion information is periodically propagated to all switches in HULA, and each switch maintains the best next hop to the destination. The design has drawbacks that prevent it from scaling to large networks. First, flooding introduces significant bandwidth overhead. Congestion information of all links is broadcast to all switches every 200 µs [24], even if there is no traffic between a pair of ToR switches. Second, each switch needs to maintain state for all racks, and process congestion information periodically. This brings vast storage and processing overhead.

Proposals such as MPTCP [30] split flows in the transport layer and send them to multiple paths to improve throughput. MPTCP cannot effectively improve latency for mice flows. In fact it degrades latency quite significantly [4, 19]. Its throughput performance for elephant flows is also demonstrated to be inferior to CONGA [4].

In a related thread, beyond fixing ECMP, much work focuses on the latency issue of mice flows. DCTCP [5], HULL [6], RCP [15], FCP [18], TIMELY [28], and DX [26] aim to reduce queue length through better congestion control. D^3 [34], PDQ [20], D^2TCP [33], pFabric [7], PASE [29], and PIAS [9] prioritize mice flows in packet scheduling and rate assignment of TCP based on either deadlines or flow sizes. DIBS [36] and CP [12] develop data plane techniques to accelerate packet retransmissions. Expeditus complements these proposals and can be used to construct a better network stack for data centers.

7. Conclusion

We presented Expeditus, a novel data plane congestion-aware load balancing protocol for data centers. Expeditus collects local congestion information, and applies two-stage path selection to aggregate such information and route flows. Both Click implementation and packet-level ns-3 simulation show that Expeditus can improve network performance for both mice and elephant flows. Expeditus advances state-of-the-art by enabling congestion-aware load balancing for general 3-tier Clos topologies widely used in industry, whose complexity presents non-trivial challenges on the scalability and effectiveness of the design.

Acknowledgments

We thank the anonymous reviewers for their comments that improved the paper. This work was supported by the Research Grants Council, University Grants Committee of Hong Kong (awards ECS-21201714, GRF-11202315, and CRF-C7036-15G), the ICT R&D program of MSIP/IITP of Korea [B0101-16-1368], and National Research Foundation of Korea funded by MSIP (NRF- 2013R1A1A1076024).

References

[1] Agache, A., Deaconescu, R., and Raiciu, C. Increasing Datacenter Network Utilisation with GRIN. In *Proc. USENIX NSDI* (2015).

[2] Al-Fares, M., Loukissas, A., and Vahdat, A. A scalable, commodity data center network architecture. In *Proc. ACM SIGCOMM* (2008).

[3] Al-Fares, M., Radhakrishnan, S., Raghavan, B., Huang, N., and Vahdat, A. Hedera: Dynamic flow scheduling for data center networks. In *Proc. USENIX NSDI* (2010).

[4] Alizadeh, M., Edsall, T., Dharmapurikar, S., Vaidyanathan, R., Chu, K., Fingerhut, A., Lam, V. T., Matus, F., Pan, R., Yadav, N., and Varghese, G. CONGA: Distributed Congestion-Aware Load Balancing for Datacenters. In *Proc. ACM SIGCOMM* (2014).

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

[6] Alizadeh, M., Kabbani, A., Edsall, T., Prabhakar, B., Vahdat, A., and Yasuda, M. Less is more: Trading a little bandwidth for ultra-low latency in the data center. In *Proc. USENIX NSDI* (2012).

[7] Alizadeh, M., Yang, S., Sharif, M., Katti, S., Prabhakar, N. M. B., and Shenker, S. pFabric: Minimal near-optimal datacenter transport. In *Proc. ACM SIGCOMM* (2013).

[8] Andreyev, A. Introducing data center fabric, the next-generation Facebook data center network. https://code.facebook.com/posts/360346274145943/introducing-data-center-fabric-the-next-generation-facebook-data-center-network/, November 2014.

[9] Bai, W., Chen, L., Chen, K., Han, D., Tian, C., and Wang, H. PIAS: Practical information-agnostic flow scheduling for data center networks. In *Proc. USENIX NSDI* (2015).

[10] Benson, T., Anand, A., Akella, A., and Zhang, M. Microte: Fine grained traffic engineering for data centers. In *Proc. ACM CoNEXT* (2011).

[11] Cao, J., Xia, R., Yang, P., Guo, C., Lu, G., Yuan, L., Zheng, Y., Wu, H., Xiong, Y., and Maltz, D. Per-packet load-balanced, low-latency routing for Clos-based data center networks. In *Proc. ACM CoNEXT* (2013).

[12] Cheng, P., Ren, F., Shu, R., and Lin, C. Catch the Whole Lot in an Action: Rapid Precise Packet Loss Notification in Data Centers. In *Proc. USENIX NSDI* (2014).

[13] Curtis, A. R., Mogul, J. C., Tourrilhes, J., Yalagandula, P., Sharma, P., and Banerjee, S. Devoflow: Scaling flow management for high-performance networks. In *Proc. ACM SIGCOMM* (2011).

[14] Dixit, A., Prakash, P., Hu, Y. C., and Kompella, R. R. On the impact of packet spraying in data center networks. In *Proc. IEEE INFOCOM* (2013).

[15] Dukkipati, N., and McKeown, N. Why flow-completion time is the right metric for congestion control. *SIGCOMM Comput. Commun. Rev. 36*, 1 (January 2006), 59–62.

[16] Ghorbani, S., Godfrey, B., Ganjali, Y., and Firoozshahian, A. Micro Load Balancing in Data Centers with DRILL. In *Proc. ACM HotNets* (2015).

[17] Greenberg, A., Hamilton, J. R., Jain, N., Kandula, S., Kim, C., Lahiri, P., Patel, P., and Sengupta, S. VL2: A Scalable and Flexible Data Center Network. In *Proc. ACM SIGCOMM* (2009).

[18] Han, D., Grandl, R., Akella, A., and Seshan, S. FCP: A Flexible Transport Framework for Accommodating Diversity. In *Proc. ACM SIGCOMM* (2013).

[19] He, K., Rozner, E., Agarwal, K., Felter, W., Carter, J., and Akella, A. Presto: Edge-based Load Balancing for Fast Datacenter Networks. In *Proc. ACM SIGCOMM* (2015).

[20] Hong, C.-Y., Caesar, M., and Godfrey, P. B. Finishing flows quickly with preemptive scheduling. In *Proc. ACM SIGCOMM* (2012).

[21] Jeyakumar, V., Alizadeh, M., Geng, Y., Kim, C., and Maziéres, D. Millions of little minions: Using packets for low latency network programming and visibility. In *Proc. ACM SIGCOMM* (2014).

[22] Kabbani, A., Vamanan, B., Hasan, J., and Duchene, F. FlowBender: Flow-level Adaptive Routing for Improved Latency and Throughput in Datacenter Networks. In *Proc. ACM CoNEXT* (2014).

[23] Kandula, S., Katabi, D., Sinha, S., and Berger, A. Dynamic load balancing without packet reordering. *SIGCOMM Comput. Commun. Rev. 37*, 2 (April 2007), 51–62.

[24] Katta, N., Hira, M., Kim, C., Sivaraman, A., and Rexford, J. HULA: Scalable Load Balancing Using Programmable Data Planes. In *Proc. ACM SOSR* (2016).

[25] Kohler, E. *The Click Modular Router*. PhD thesis, Massachusetts Institute of Technology, 2001.

[26] Lee, C., Park, C., Jang, K., Moon, S., and Han, D. Accurate Latency-based Congestion Feedback for Datacenters. In *Proc. USENIX ATC* (2015).

[27] Liu, S., Xu, H., and Cai, Z. Low latency datacenter networking: A short survey. http://arxiv.org/abs/1312.3455, 2014.

[28] Mittal, R., Lam, V. T., Dukkipati, N., Blem, E., Wassel, H., Ghobadi, M., Vahdat, A., Wang, Y., Wetherall, D., and Zats, D. TIMELY: RTT-based Congestion Control for the Datacenter. In *Proc. ACM SIGCOMM* (2015).

[29] Munir, A., Baig, G., Irteza, S. M., Qazi, I. A., Liu, A. X., and Dogar, F. R. Friends, not Foes – Synthesizing Existing Transport Strategies for Data Center Networks. In *Proc. ACM SIGCOMM* (2014).

[30] Raiciu, C., Barre, S., Pluntke, C., Greenhalgh, A., Wischik, D., and Handley, M. Improving datacenter performance and robustness with multipath tcp. In *Proc. ACM SIGCOMM* (2011).

[31] Sen, S., Shue, D., Ihm, S., and Freedman, M. J. Scalable, optimal flow routing in datacenters via local link balancing. In *Proc. ACM CoNEXT* (2013).

[32] Singh, A., Ong, J., Agarwal, A., Anderson, G., Armistead, A., Bannon, R., Boving, S., Desai, G., Felderman, B., Germano, P., Kanagala, A., Provost, J., Simmons, J., Tanda,

E., Wanderer, J., Hölzle, U., Stuart, S., and Vahdat, A. Jupiter Rising: A Decade of Clos Topologies and Centralized Control in Google's Datacenter Network. In *Proc. ACM SIGCOMM* (2015).

[33] Vamanan, B., Hasan, J., and Vijaykumar, T. Deadline-aware datacenter TCP (D2TCP). In *Proc. ACM SIGCOMM* (2012).

[34] Wilson, C., Ballani, H., Karagiannis, T., and Rowstron, A. Better never than late: Meeting deadlines in datacenter networks. In *Proc. ACM SIGCOMM* (2011).

[35] Xu, H., and Li, B. TinyFlow: Breaking elephants down into mice in data center networks. In *Proc. IEEE LANMAN* (2014).

[36] Zarifis, K., Miao, R., Calder, M., Katz-Bassett, E., Yu, M., and Padhye, J. DIBS: Just-in-time Congestion Mitigation for Data Centers. In *Proc. USENIX EuroSys* (2014).

[37] Zats, D., Das, T., Mohan, P., Borthakur, D., and Katz, R. DeTail: Reducing the flow completion time tail in datacenter networks. In *Proc. ACM SIGCOMM* (2012).

[38] Zhou, J., Tewari, M., Zhu, M., Kabbani, A., Poutievski, L., Singh, A., and Vahdat, A. WCMP: Weighted cost multipathing for improved fairness in data centers. 2013.