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

Peng Wang, Hong Xu, Zhixiong Niu
NetX Lab, City University of Hong Kong

Dongsu Han
KAIST

ABSTRACT

Data center networks often use multi-rooted Clos topologies to provide a large number of equal cost paths between two hosts. Thus, load-blancing 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 (as used in CONGA) 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 congestion information across switches and make path selection decisions. Testbed evaluation on Emulab with Click implementation and large-scale ns-3 simulations demonstrate that, Expeditus outperforms ECMP by up to 2× in tail flow completion times (FCT) for mice flows, and by 10%–30% in mean FCT for elephant flows in 3-tier Clos networks.

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, 26, 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) [29] 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] and can work at different load balancing granularities (e.g. flowlets [23, 30] or flowcells [19]).

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 [31], 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 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 Expeditus,[1] a 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

[1] Latin for "expedited."

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. We believe this design achieves a desirable trade-off between practicality and performance. By exploiting the salient 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. It also handles persistent TCP connections as well as flowlets or flowcells for finer-grained load balancing [23] by using a small timeout value for routing entries and reordering-tolerant mechanisms at the edge [19].

We implement Expeditus using the Click software router [24]. We evaluate it on the Emulab testbed (§5) as well as in large-scale packet-level ns-3 simulations (§6), using empirical flow size distributions from two production networks. Our evaluation shows that Expeditus provides up to 2× reductions in 95%ile flow completion time (FCT) for mice flows compared to ECMP in 3-tier Clos. The average FCT of elephant flows (≥1MB) is also reduced by ∼10%–40%. 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, 32, 34, 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-unaware 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.

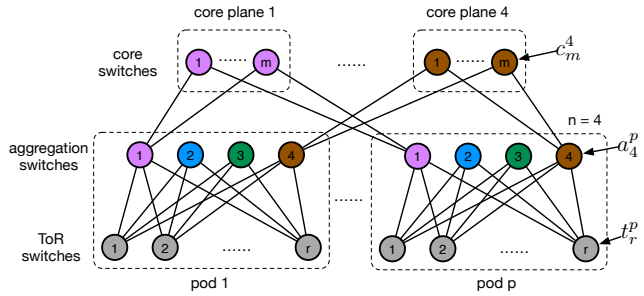


Figure 1: A general 3-tier Clos topology widely used in practice [8]. Not all switches or links are shown. There are p pods. Each pod has r ToR switches connecting to n aggregation switches. Here n is usually 4 as ToR switches have 4x40Gbps uplinks. There are m core switches in each of the n core planes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

3-tier Clos topologies [1, 8, 31] 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

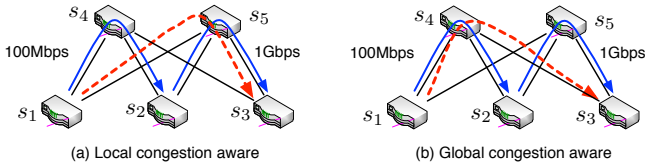


Figure 2: Two elephant flows (solid lines) are transmitting as shown here. When a new flow, shown in dashed line, arrives at s_1, a local congestion aware protocol sends it to the less congested s_5. This results in poor overall performance due to the severe downstream congestion at the link (s_5, s_3).

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 [31], 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. Figure 2 illustrates an example where decision based on local information leads to a sub-optimal result. A flow of 100Mbps is traversing the path (s_1, s_4, s_2), and another flow of 1Gbps is traversing (s_2, s_5, s_3). When a new flow (dashed line) arrives at s_1, local congestion-aware load balancing would send it to s_5 without knowing the downstream congestion, which results in poor performance.

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 in 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 when the number of paths increases significantly. 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 3. 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.

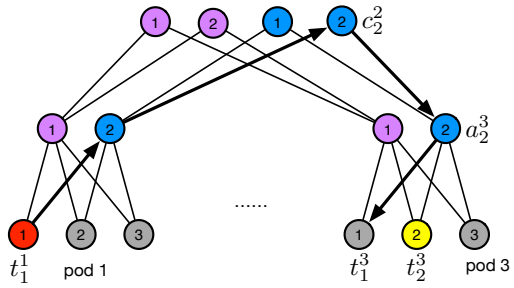


Figure 3: Per-pathlet congestion information collection is not timely enough. Here is a small Clos network with $n = m = 2$, $r = 3$. The link from c_2^2 to a_2^3 is congested due to a flow on the pathlet from c_2^2 to t_1^3 as shown. If t_1^1 has a new flow to send to t_2^3, as there is no flow on the 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. To overcome the above challenges, we carefully 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. 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 4. There are four hops between any two ToR 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 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 Sec. 6.1.

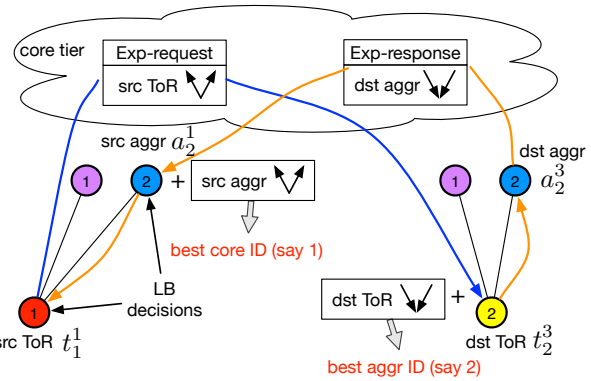


Figure 4: Design of two-stage path selection. The first stage is between the source ToR (t_1^1) and destination ToR (t_2^3) who chooses the best aggregation switch ID based on first and last hop congestion metrics. The the second stage involves the chosen destination and source aggregation switch (a_2^1) who chooses the best core switch ID. The information exchange is done via Exp-request and Exp-response.

Next, we present the details of each component.

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 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. A switch monitors its uplinks in both the egress and ingress directions.

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 $T_{dre} = 20\mu s$ 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 and delicate 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 path 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). An example of a PST is shown in Figure 5.

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 1, 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 (or flowlet, see Sec. 3.4), 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 mainly focus on per-flow load balancing, the time out value is set to 100ms which is large enough to filter out bursts of the same flow. Changing the time out value allows Expeditus to perform load balancing in different granularities and support persistent TCP connections and flowlets (Sec. 3.4). The timer can be implemented using just one extra bit for each entry and a global timer for the entire PST [4].

Flow ID	egress port	PSS	valid
0xF0A3	1	1	1
0x9EB2	4	1	0
...	...	...	...

Figure 5: Path selection table layout.

PST goes against conventional doctrine that says it is not practical to maintain per-flow state at switches. We argue that the implementation cost of tracking each flow 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 with low cost.

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 Ethernet header field is added between the source MAC address and the EtherType/length fields. The structure of the new field is shown in Figure 6, 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, with at most 128 which is 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.

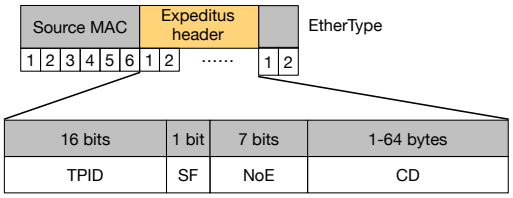


Figure 6: Format of the Expeditus tag for path selection.

Path selection algorithm. Figure 7 illustrates our two-stage path selection for two flows in the topology of Figure 3. There is a new TCP connection between hosts under ToR 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 own egress loads, and

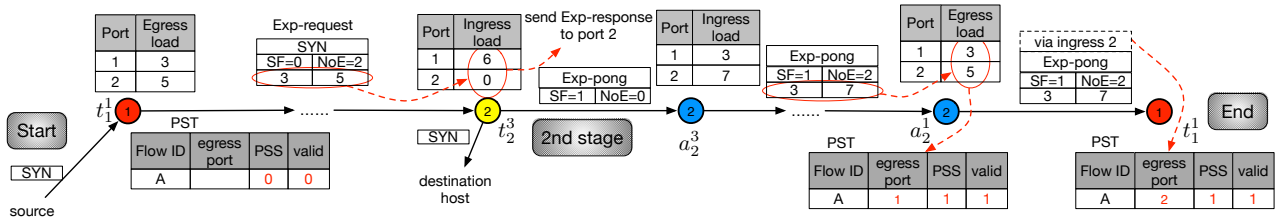


Figure 7: Two-stage path selection in Expeditus for the network shown in Figure 3. We only show the path selection process for flow A from the source ToR t_1^1 to t_2^3. Path selection for flow B of the same TCP connection in the reverse direction is done similarly.

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 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,[2] 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 ECMP.

[2]The Exp-response is generated immediately upon receiving the Exp-request. We can also prioritize Exp-request and Exp-response packets in practice.

Algorithm 1 Expeditus, ToR switch

```
1: procedure ToR_Switch_Processing(packet p)
2:     if p is northbound to the aggregation tier then
3:         if a PST entry e exists for p then
4:             if e.valid_bit == 1 then
5:                 forward p according to e, return
6:             else if PSS == 0 then
7:                 forward p by ECMP, return
8:         add the Expeditus tag to p                    ▷ Start path selection
9:         SF ← 0, add the egress loads, forward p by ECMP
10:        insert a new PST entry or update the existing one for p, PSS
   ← 0, valid_bit ← 0, return
11:    else                                              ▷ southbound packet
12:        if p is Expeditus tagged then
13:            if p.SF == 0 then                         ▷ Exp-request received
14:                check NoE, pull CD
15:                choose the best aggregation switch ID f*
16:                generate an Exp-response p', p'.SF← 1, p'.NoE ← 0
17:                p'.src_ip ← p.dst_ip, p'.dst_ip ← p.src_ip
18:                forward p' to aggregation switch f*
19:                remove the tag from p, forward it, return
20:            else                                      ▷ Exp-response received
21:                record p's ingress port p_i
22:                find the PST entry e for the flow f, f.src_ip=p.dst_ip,
   f.dst_ip=p.src_ip
23:                e.egress_port ← p_i, e.PSS ← 1, e.valid_bit ← 0,
   discard p, return
```

We illustrate this using the example of Figure 8. Suppose the link from a_1^1 to c_1^1 is down. This causes all links from c_1^2 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_1^2 link is more 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 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 switches.

Algorithm 2 Expeditus, aggregation switch

```
1: procedure AGGR_SWITCH_PROCESSING(packet p)
2:     if p is northbound to the core tier then
3:         if p is Expeditus tagged, p.SF == 1 then   ▷ Exp-response, first hop
4:             add the switch's ingress loads to p, set p.NoE
5:         if a PST entry e exists for p then
6:             if e.valid_bit == 1 then
7:                 forward p according to e, return
8:         forward p by ECMP, return
9:     else                                           ▷ southbound packet
10:        if p is Expeditus tagged, p.SF == 1, p.NoE is non-zero then
11:                                                   ▷ Exp-response, third hop
12:            check NoE, pull CD
13:            choose the best core switch ID f*
14:            record port p_i connected to core switch f*
15:            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
16:            e.egress_port ← p_i, e.PSS ← 1, e.valid_bit ← 1
17:        forward p, return
```

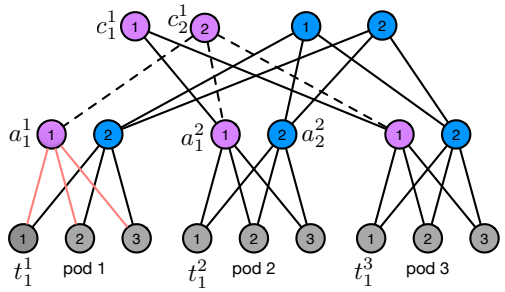


Figure 8: The link between a_1^1 and c_1^1 fails. This causes all links from c_1^2 to the first aggregation switches of each pod to be congested, as they are the only paths to reach a_1^1. A link load multiplier of 2 is applied to links from a_1^1 to each ToR in pod 1 as shown for inter-pod traffic.

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 8. 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_1^2 and re-distributes traffic properly.

3.4 Discussions

2-tier leaf-spine topologies: Here, we briefly explain how Expeditus handles 2-tier leaf-spine topologies. This also applies to intra-pod traffic in a 3-tier Clos. The two-stage path selection 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 path 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.

Finer-grained load-balancing: Our current Expeditus design works on a per-flow basis. It can be readily extended for finer-grained load balancing, such as flowlet [23] or flowcell [19] switching, by just using a smaller PST time out value of say 500μs. The path selection mechanism requires no change because we can tag any packet. A side effect is that a small time out introduces packet reordering, since the first few packets of a new flowlet is routed by ECMP before a path is selected. The extent of reordering caused by Expeditus is extremely mild though, as we can configure Exp-request and Exp-response with high priority to speed up the process. A more complete solution can be developed by either modifying the TCP stack or the host hypervisor [19] to tolerate reordering, which is beyond the scope of this paper. By the same token, a fairly small time out (10ms) can easily allow Expeditus to support persistent TCP connections.

4 Prototype Implementation

We implement a prototype of Expeditus in Click, a modular software router for fast prototyping of routing protocols [24]. We develop two new Click elements, `DRE` to measure link load and `EXPRoute` to conduct two-stage path selection. Figure 9 shows the packet processing pipeline of Expeditus for a ToR or aggregation switch with 4 ports. Here devices *eth0* and *eth1* are connected by point-to-point links to lower-tier routers or hosts in the topology; *eth2* and *eth3* to upper-tier routers.

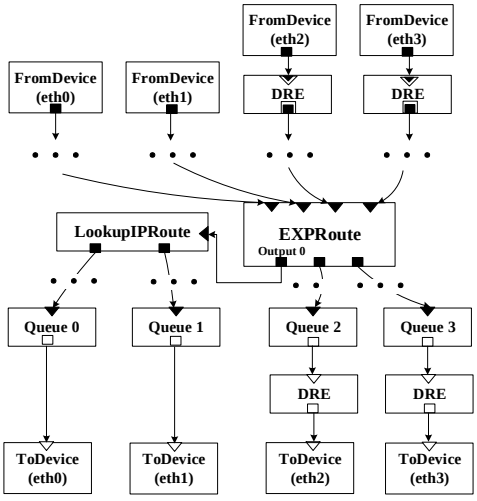


Figure 9: Packet processing pipeline in Click.

In the Click configuration, `EXPRoute` handles all incoming packets. If the destination IP address of the packet matches the destination subset of this switch, i.e. the packet is southbound, it is emitted on `output 0` and passed to `LookupIPRoute`

as in Figure 9. The `LookupIPRoute` element then matches destination address with routing table entries, and forwards it to the correct downstream egress port. Otherwise, if the packet is northbound, `EXPRoute` chooses an egress port according to its two-stage path selection mechanism.

Implementation of the `DRE` element is quite simple. It sits next to `FromDevice` and `ToDevice` elements for *eth2* and *eth3* in the Click configuration, and can accurately detect packets sent from/to each link. `EXPRoute` can obtain ingress link loads from upstream `DRE` elements and egress link loads from downstream `DRE` elements. 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 than a hardware implementation 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 in our implementation.[3]

5 Testbed Evaluation

We run our Click implementation 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? (2) How does Expeditus actually perform in practice?

5.1 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.

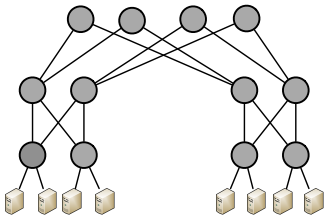


Figure 10: Emulab testbed topology.

Constrained by the number of NICs per node in Emulab, 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 as shown in Figure 10. The core tier is oversubscribed at 4:1 by rate limiting the core links to emulate a realistic setting in practice [8].

[3]Since the Click routers are output buffered, the ingress queue length is remote and obtained by piggybacking in data packets in the implementation.

5.2 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 [24].

Element	**Time** (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.

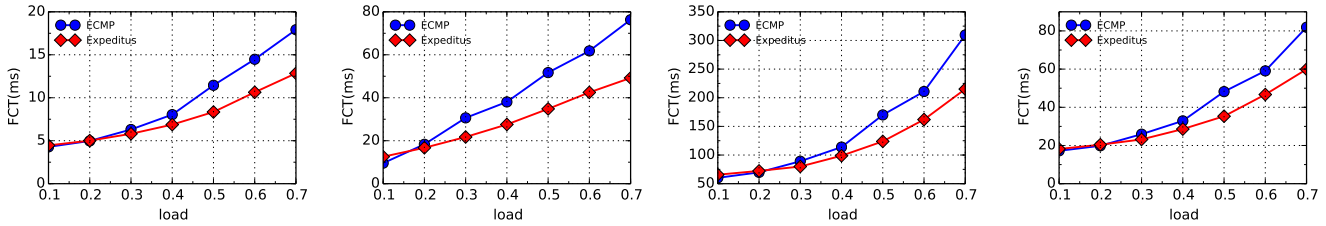
We also look at the latency overhead of two-stage path selection to TCP handshake. We start a TCP connection and measure how long it takes for the sender to start sending the first data packet, by which time path selection for both directions are done and cannot affect the flow anymore. Expeditus adds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

5.3 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 1 MB; 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 in order to simulate different loads. Our setup is consistent with existing work [4, 9, 19].

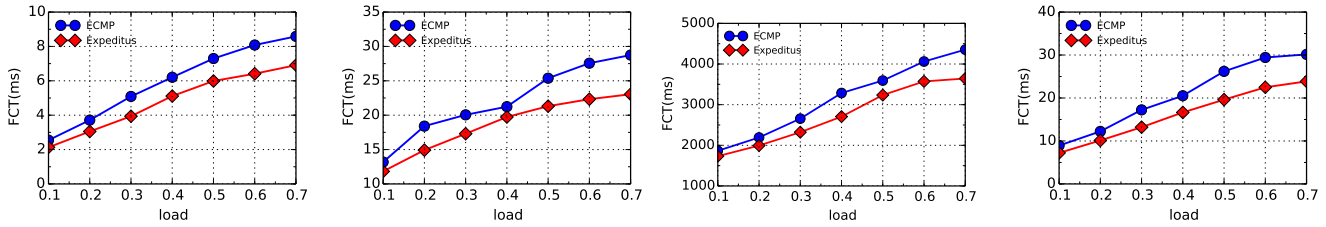
Figure 11 and 12 show the FCT results of Expeditus and ECMP for both workloads. We vary loads from 0.1 to 0.7 beyond which the results become unstable in our testbed. We emphasize FCT statistics for mice (<100KB) and elephant flows (>1MB). The results of medium flows between 100KB and 1MB are largely 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



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

Figure 11: FCT for web search workload in the Emulab testbed. Core tier oversubscription 4:1.



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

Figure 12: FCT for data mining workload in the Emulab testbed. Core tier oversubscription 4:1.

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 substantially 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.[4] 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 that only happens for a very short period of time. Our implementation updates the link utilization using timers that operate at coarse-grained timescales (1ms granularity). However, in lightly loaded networks, congestion is much more transient and often happens at finer-grained timescales compared to heavily loaded cases. Meanwhile, the queue length information, used in conjunction with link load (DRE §4), also does not help as much because there is very little queue build-up. 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 (§6.3).

6 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 (§6.2, §6.3)? (2) Is Expeditus sensitive to network bottleneck conditions (§6.4)? (3) Is Expeditus effective in handling topological asymmetry caused by link failures (§6.5)? (4) How does it perform in leaf-spine networks compared to existing work (§6.6)?

6.1 Methodology

Simulation Setup. We use a 12-pod fat-tree [2] as the baseline topology. There are 36 core switches, i.e. 36 equal-cost paths between any pair of hosts at different pods and 432 hosts. Each ToR switch has 6 connections 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 3 runs unless otherwise noted.

We use a number of different workloads.

Realistic: We use two realistic workloads from production datacenters: a web search workload [5] and a data mining workload [17]. Each host continuously samples flow sizes from the two empirical distributions and inter-arrival times from an exponential distribution, and sends the flow to a random receiver in a different pod.

For completeness, we also evaluate Expeditus with traditional synthetic communication patterns. They are only used to perform stress tests, similar to previous work [2, 3, 19]:

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.

[4] Benefits for medium flows between 100KB and 1MB are largely in line: average FCT is improved by ∼11%–32% and ∼12%–21% in the two workloads, respectively.

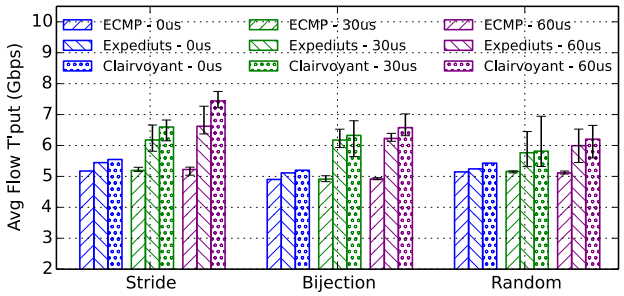


Figure 13: Stress test results for different traffic patterns and mean inter-arrival times in a 4-pod 10Gbps fat-tree.

Benchmarks. We compare four load balancing schemes in our simulations.

- *Expeditus*: Our distributed congestion-aware protocol.
- *Clairvoyant*: An "ideal" congestion-aware scheme that uses complete path-wise information of all possible paths. Whenever a new flow comes, the source ToR chooses the best path. As discussed in Sec. 2.2 such a scheme introduces prohibitive overhead for practical implementation.
- *CONGA-Flow*: The state-of-the-art distributed congestion-aware protocol. CONGA is integrated with flowlet switching which by itself also helps load balancing [23]. To make the comparison fair, we implement CONGA-Flow that works on a per-flow basis. Our implementation is faithfully based on all details provided in [4]. CONGA-Flow only works for leaf-spine networks and is not included in experiments with 3-tier Clos.
- *ECMP*: This is our baseline.

6.2 Stress Tests with Synchronized Flows

We first perform stress tests by generating synchronized flows with the three synthetic traffic patterns to evaluate the load balancing performance of Expeditus. 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 sampled from exponential distributions with means 0, 30μs, and 60μs.

Figure 13 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 in this case. Clearly this perfectly synchronized flow arrival seldom happens in practice. When flows are slightly 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 30μs and 60μs. 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 network edge. Note, Expeditus is very close to Clairvoyant, showing a performance gap of at most 9%.

6.3 Baseline Performance in 3-tier Clos

We now investigate the performance of Expeditus in a large-scale Clos network, using realistic traffic traces while varying the network loads from 0.1 to 0.8. Figure 14 shows the normalized FCT (NFCT) times for the web search workload in the baseline 12-pod fat-tree with core tier oversubscribed at 2:1. Note, 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]. For mice flows, Expeditus provides 20%–50% FCT reduction at the 95%ile over ECMP. For elephants, Expeditus is also ~25%–30% faster on average. The tail and average FCT improvements are more substantial in smaller loads. This is because an idle path is more likely to be found at small loads. Moreover, it is clear that performance of Expeditus closely tracks that of Clairvoyant. In most cases the performance gap is less than 10%, demonstrating the effectiveness of the heuristic path selection design. We observe similar improvements for the data mining workload whose result is reported in our technical report [33].

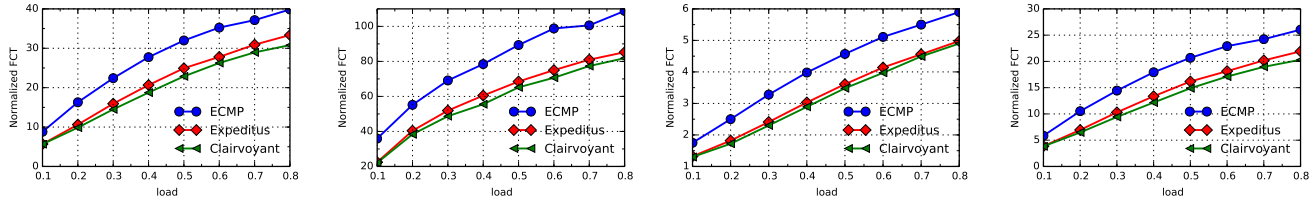
We make two observations. First, the results confirm that Expeditus is more efficient in balancing loads across the network than ECMP. It reduces the average throughput imbalance significantly compared to ECMP. Note the imbalance is at most 3 at the core tier instead of 6 as in the ToR tier, because the core tier is oversubscribed at 2:1 in the baseline topology (12-pod fat-tree). 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 than Expeditus.

6.4 Impact of Network Bottleneck

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

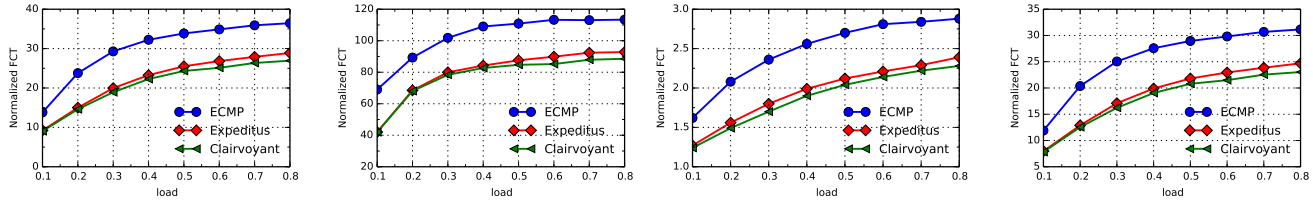
We first consider the impact of bottleneck severity. Figure 17a shows Expeditus's average FCT improvement over ECMP for all flows in the baseline topology with varying oversubscription ratios at the core tier. Only results for web search workload is shown for brevity. In general, we find that Expeditus 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 17b shows the result when the ToR tier is oversubscribed



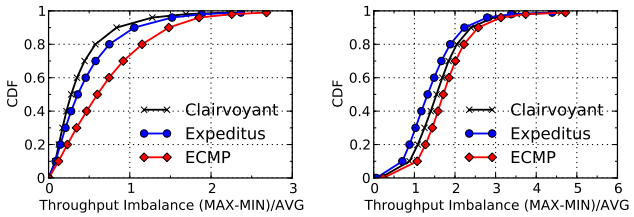
(a) (0,100KB]: Avg **(b)** (0,100KB]: 95-th percentile **(c)** (1MB, ∞): Avg **(d)** Overall: Avg

Figure 14: Normalized FCT for web search workload in the baseline 12-pod fat-tree with core tier oversubscription of 2:1.



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

Figure 15: Normalized FCT for data-mining workload in the baseline 12-pod fat-tree with core tier oversubscription of 2:1.



(a) Imbalance at aggregation tier. **(b)** Imbalance at ToR tier.

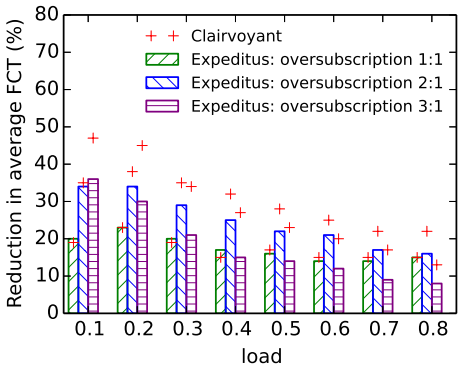
Figure 16: CDF of throughput imbalance at different tiers of a 12-pod fat-tree with 2:1 core tier oversubscription for the web search workload at 50% load.

with more hosts. Now uplinks of ToR switches instead of aggregation switches are the bottleneck. 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. This 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 may direct a flow to an aggregation switch that connects to congested core switches, and Expeditus is unable to change the decision at the second stage.

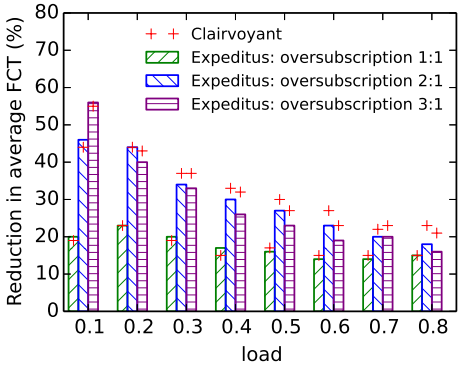
In sum, Expeditus consistently outperforms other approaches in many different topology settings, different severity and location of network bottleneck.

6.5 Impact of Link Failure

In this section, we study the impact of link failures and topology asymmetry. To emphasize the performance of affected 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 20 shows the overall average FCT reduction of both Clairvoyant and Expeditus over ECMP for all flows at 0.5 load. We observe qualitatively similar

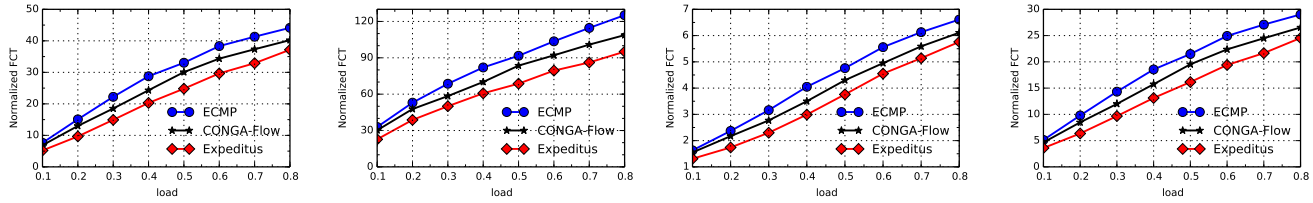


(a) Oversubscription at the core tier.



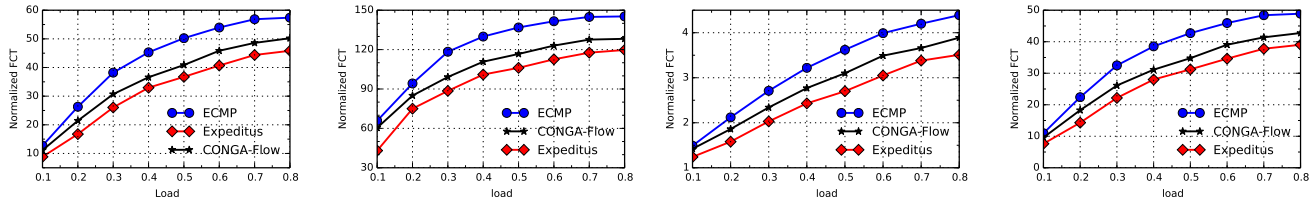
(b) Oversubscription at the ToR tier.

Figure 17: 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.



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

Figure 18: Normalized FCT for web search workload with 8 spine and 8 leaf switches. Network oversubscription 2:1.

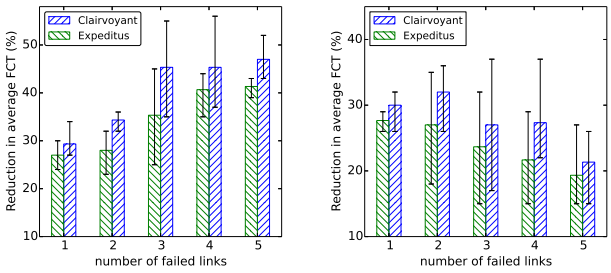


(a) (0,100KB]: Avg **(b)** (0,100KB]: 95-th percentile **(c)** (1MB, ∞): Avg **(d)** Overall: Avg

Figure 19: Normalized FCT for data mining workload with 8 spine and 8 leaf switches. Network oversubscription 2:1.

results for other loads.

We consider two scenarios: aggr-core and ToR-aggr link failures. Observe from Figure 20a 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 20b shows the result when failures happen at the ToR-aggr links. Across different scenarios Expeditus provides moderate (20%–27%) performance gains. In all, Expeditus is particularly robust against failures in 3-tier Clos networks.



(a) Aggr-core link failure. **(b)** ToR-aggr link failure.

Figure 20: Average FCT reduction by Clairvoyant and Expeditus over ECMP for all flows with link failures.

6.6 Performance in Leaf-spine Topologies

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

Figures 18 and 19 show the FCT results for the web search and data mining workload. Expeditus achieves favorable performance gains ranging from 10%–30% for all flows across all loads. More interestingly, it outperforms CONGA-Flow in all cases. The main reason is that CONGA-Flow can observe the congestion state of the paths only if there are flows. As there may not be enough concurrent flows to cover all paths at all times (§2.3), CONGA-Flow does not observe the up-to-date, global state of the network.

7 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 uncongested paths based on global network state [3,10,13,30]. This approach poses scalability challenges for OpenFlow switches [13], and is too slow to improve latency for mice flows. The second approach is finer 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 [30], and Presto [19] split a flow into many small bursts, and load balances them to avoid 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 dealing with flash congestion in the network.

Proposals such as MPTCP [29] split flows in the transport layer and send them to separate network 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]. For these reasons we do not include it in our evaluation.

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 [27], and DX [25] aim to reduce queue length through better congestion control. D^3 [34], PDQ [20], D^2TCP [32], pFabric [7], PASE [28], 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.

8 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. We are committed to making our Click implementation code and experiment scripts open source after the paper review for the community.

9 References

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

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

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

[4] 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. In *Proc. ACM SIGCOMM*, 2014.

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

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

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

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

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

[11] 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. In *Proc. ACM CoNEXT*, 2013.

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

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

[14] A. Dixit, P. Prakash, Y. C. Hu, and R. R. Kompella. On

the impact of packet spraying in data center networks. In *Proc. IEEE INFOCOM*, 2013.

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

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

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

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

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

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

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

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

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

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

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

[26] S. Liu, H. Xu, and Z. Cai. Low latency datacenter networking: A short survey. http://arxiv.org/abs/1312.3455, 2013.

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

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

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

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

[31] A. Singh, J. Ong, A. Agarwal, G. Anderson, A. Armistead, R. Bannon, S. Boving, G. Desai, B. Felderman, P. Germano, A. Kanagala, J. Provost, J. Simmons, E. Tanda, J. Wanderer, U. Hölzle, S. Stuart, and A. Vahdat. Jupiter Rising: A Decade of Clos Topologies and Centralized Control in Google's Datacenter Network. In *Proc. ACM SIGCOMM*, 2015.

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

[33] P. Wang, H. Xu, Z. Niu, and D. Han. Expeditus: Distributed Congestion-aware Load Balancing in Clos Data Center Networks. , Technical report, 2016.

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

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

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

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

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