(Full Technical Report)

P4TE: P4 Based In-Band Traffic Engineering System for Fat-Tree Topology Data Center Networks

Debobroto Das Robin
Kent State University

Javed I. Khan
Kent State University

Abstract

In this work, we present P4TE, an in-band traffic sensing, load-aware forwarding, and rate control mechanism for fat- tree topology-based data center networks. It achieves sub-RTT reaction time to change in network conditions, achieves im- proved flow completion time and balanced link utilization. Unlike the classical probe-based monitoring approach, we used in- band sensing by using P4 supported programmable switches. It is implementable on emerging v1model.p4 architecture-based programmable data center switches. Our evaluation shows that P4TE outperforms ECMP in key traffic engineering objectives and achieves improved fairness.

1 Introduction

Traffic engineering (TE) is the prevalent mechanism to achieve application-level QoS requirements and optimal resource usage in data center networks (DCN). A large set of research works [26, 28] with *centralized*, *distributed* (entirely in end hosts or switches or a mix of both) and *hybrid* architectures have been proposed to address various aspects of TE in DCN. Among them, *switch-based* distributed TE systems can make better decisions for TE as flows from multiple sources pass through them. Moreover, these systems can react to changing network conditions in sub-RTT time. But, this kind of solution requires modification in the *switch* protocol stack. Until recent time, this was not possible. The recent emergence of programmable data plane devices/programmable switches (commonly referred to as the Protocol Independent Switch Architecture (PISA) switch [9]) along with P4 programming language, has opened up the possibilities for implementing customized algorithms for TE in the data plane. To maintain a high packet processing rate, these devices are built as a multi-staged pipeline with a limited per-packet processing budget and stateful memory access in each stage (one actio per header field, one memory op). Such architectural constraints impose few important restrictions (§ 4) on TE system's implementation over PISA switches.

—put this in footnote Even approximation algorithms for TE are also highly complex [10].

TE is computationally expensive; if designed without considering the architectural constraints ((§ 4), a TE algorithm can miss a PISA switch's line rate leading to throughput loss due to lack of cross-stage memory aceess. Moreover, using too many ALU and memory for TE algorithms leaves few resources for other functionalities. Altogether, these factors make the design and implementation of TE systems using programmable switches a non-trivial task. But the highly regular structure of DCN network topologies can help reduce the computational complexity [10] of TE algorithms, whereas the problem is intractable in general. This suggests that, instead of focusing on TE systems for generalized topology, it is more practical to work on TE systems for widely used DCN topologies (fat tree, bcube, etc.).

Several programmable switch-based systems focusing on various aspects of TE (P4 based systems cite) are proposed in recent times. But their (ECMP, W-ECMP [34], etc.) static path selection strategy fails to adapt with varying network conditions at a fine-grained timescale. Moreover, the use of probe-based monitoring mechanism and end-host-based rate-controlling schemes also make a sub-optimal decision in flow rate handling. In this work we have explored these issues to desing a TE system for fat-tree topology based DCN.

We summarized the key restrictions (§ 4) behind TE system implementation using P4 supported PISA switches. Then, we present the design and implementation of **P4TE**, a distributed TE system for **fat-tree** topology-based DCN, built using P4 supported switches. It monitors (§ 6.1) 4 important traffic statistics and link performance metrics. P4TE uses these monitoring results for load-aware path selection (§ 6.2.1) and flow rate adaption (§ 6.4).

Overall, P4TE achieves the following important objectives: P4TE doesn't require centralized controller as like as OpenFlow or many other PISA based works. It doesn't require any modification in the end-host transport layer protocol stack. Its *in-band* monitoring technique identifies traffic events happening in the data plane with fine-grained accuracy. P4TE uses monitored information received from the data plane to implement a traffic-aware forwarding path selection algorithm. This algorithm follows the traffic-class-based hop-by-hop packet forwarding policy, which helps keep the forwarding table size small. In the control plane (CP), P4TE uses a simple and efficient local CPU-based algorithm to respond to network events faster than centralized TE systems. P4TE's *switch-assisted* TCP window size-based rate control technique achieves sub-RTT reaction time to network events. We have implemented P4TE using $P4_{16}$ language (for data plane logic) and P4Runtime framework (for control plane logic). P4TE has a small resource usage footprint (§ 6.6) in the PISA pipeline. Furthermore, in our performance evaluation (§ 7), P4TE outperforms ECMP in key performance metrics. All these factors make P4TE a suitable candidate for real-life deployment in data center networks.

2 Related Work

In this section, we focus on important works related to TE in the context of intradomain DCN with emphasis on programmable switches.

Performance of TE systems in DCN depends mainly on 2 key factors: a) successful use of path diversity (load balancing) provided by the DCN topology and b) controlling flow rates in case of resource contention (congestion).

Load Balancing: ECMP is the simplest and most popular technique for load balancing in DCN. ECMP's simple stateless hash-based implementation consumes minimal resources in PISA pipeline. Moreover, its immunity to packet re-ordering is a big advantage. Due to these factors, ECMP is the dominant choice for path selection algorithms in P4 supported hardware. But ECMP often performs poorly in case of hash collision [8, 16] and leads to uneven traffic splitting. This leads to congestion in links despite having a spare capacity in alternate paths [7]. Moreover, ECMP's incapability in considering traffic characteristics and resource usage for path selection makes it oblivious to various TE objectives.

Few recent works attempted to improve ECMP's performance by using centralized controller's global knowledge [29, 31]. Due to their centralized nature, these systems are slow to react to fine-grained traffic variations in the data plane. On the other hand, distributed load balancing systems mostly rely on out-of-band probe-based results to choose the best path for a destination. HULA [24] uses probe-based results to add network condition awareness in path selection by recording the best path to a destination at each switch. This may lead to congestion in the best path rapidly. MP-HULA [5] tries to augment this limitation by tracking top k paths. But their knowledge of the best path is dependent on probe frequency. Contra [18] uses a policy-based probing technique to find performance-aware routes. In [34], the authors focused on augmenting ECMP's performance by dynamically calculating each path's weight based on the result from the out-of-band probe. But, periodic probe-based systems can not capture changing network conditions between successive probes. DASH [19] proposed a scheme to distribute traffic among multiple available paths. But it is not clear how to integrate DASH into TE systems.

Rate Controlling: All the systems mentioned above rely on the end host's transport layer protocol stack for controlling flow rates. Fully end host-based transport layer protocols (TCP, DCTCP, etc.) need RTT time to react on change in network condition. But microbursts are one of the main reasons behind the sub-optimal performance of DCN [7], and they may last for a duration shorter than RTT time. Switch-assisted rate control protocols [11,23] can perform better than their end-host-only counterparts by calculating the fair rate for flows using aggregate knowledge of traffic collected by the switches and explicitly informing it to the source. These protocols need to maintain explicit states [11, 21] about active flows in DCN. But, DCN carries a large number of flows. Among them, long-lived flows may remain dormant for a long time, which poses a challenge for calculating fair rates for a flow. Moreover, maintaining a per-flow state is not scalable and consumes higher resources in a PISA pipeline. Despite having interesting properties (fair rate sharing, quick rate adjustment, etc.), this kind of protocol has not seen widespread deployment. In [13] proposed use of explicit congestion notifications (NACK) packets for informing flow sends about congestion. But, the work is unable to use spare link bandwidth in the absence of congestion.

Most of the works mentioned in this section consider the problem of path selection and rate adaption separately. Moreover, the use of *probe-based* monitoring technique makes them prone to miss crucial traffic events between successive probes. But for effective traffic engineering in DCN all 3 of them are equally important. Given the advantages of path diversity and shortcomings of existing techniques to use it, P4TE's first design goal is *scalable in-band monitoring of traffic events in the data plane for collecting a better view on incoming traffic and link performances*. 2nd design goal is *a better use of path diversity in fat-tree topology for higher throughput*. The third design goal is to *use PISA switches better to achieve sub-RTT reaction time on varying network conditions and improving the performance of DCN.*

3 Background

3.1 Fat-Tree Topology

Fat-tree topology is the most widely adopted architecture in modern DCN. In a 2-tier fat-tree (a.k.a. leaf-spine) topology, all tier-1 switches (leaf switch) are connected to all tier-2 switches (spine switch). Using k-port switches, a leaf-spine topology can accommodate $\frac{k^2}{2}$ end-hosts. They can be extended to 3 or 5 tiers to accommodate more end-hosts. The single administrative authority of a DCN enables hierarchical

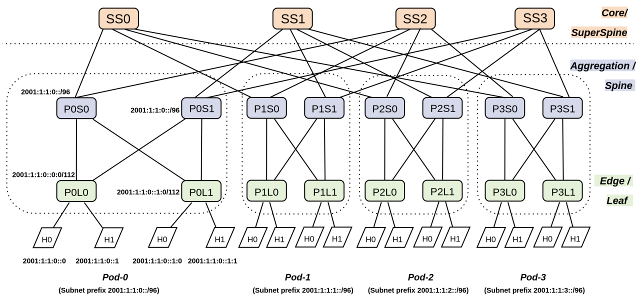


Figure 1: A fat-tree topology with k = 4 port switches and hierarchiacal IP address assignment

Prefix	Port
2001:1:1::0/128	1
2001:1:1::1/128	2

(a) P0L0's MAT_{Down}

Prefix	Port
2001:1:1::0:0/112	1
2001:1:1::1:0/112	2

(b) P0S0's MAT_{Down}

Prefix	Port
2001:1:1:0::/96	1
2001:1:1:1::/96	2
2001:1:1:2::/96	3
2001:1:1:3::/96	4

(c) SS0's MAT_{Down}

Traffic Class	Port	Safe Rate	Burst Size
0x008	3	35	5
0x010	4	20	5
0x008	3	35	5
0x010	4	20	5

(d) P0S0's MAT_{Rate}^{In} (*Rate in Gbps*)

Prefix	Routing Group	Priority
2001:1:1:1::/48	{3,4}	1
2001:1:1:1::/48	{}	2
2001:1:1:1::/48	{}	3
2001:1:1:1::/48	{}	4

(e) P0S0's MAT_{Up}^{Delay} *(low priority entry have higher priority)*

Port	CIR	CBS	PIR	PBS
3	70	10	90	10
4	70	10	90	10

(f) P0S0's $MAT_{Rate}^{Link\_Util}$ (*Rate in Gbps*)

- ■ 0x008 -- Short latency sensitive flow
- ■ 0x010 -- Large throughput oriented flow

Figure 2: Initial configurations for topology of Fig. 1

IP address assignment [1] with a single common IP prefix for the whole network (ex. Fig. 1). Here, all the end-hosts connected to a leaf switch are in the same subnet, and they share the same IP prefix. This IP prefix can be used for identifying a leaf switch uniquely. Similarly, all the leaf switches connected to a spine switch and spine switches connected to a super spine switch also share a common IP prefix. P4TE assumes the use of existing algorithms [1] for hierarchical IP address assignment in DCN.

3.2 DCN Traffic Characteristics

DCN traffic characteristics depend on the type of applications, their deployment pattern over the DCN, and their communication pattern. Studies [7,22,30] show, DCN traffic mainly comprises short latency-sensitive and large throughput-oriented flows. P4TE focuses on optimizing TE objectives for these 2 types of traffics. Our proposed packet forwarding algorithm uses two different heuristic-based strategies for these two traffic classes.

3.3 What is TE

Any technique to optimize one or more performance objectives of a network can be termed Traffic Engineering (TE) system. Assume a DCN is a directed graph $G = (V, E, c)$, where V is the set of nodes (switches and en-hosts), E is the set of links connecting two nodes. and c is a function associated with each link for defining the capacity of the link. If there are p optimization objectives and q constraints for the DCN, the general form for TE can be expressed as:

$$minimize\ Or\ maximize\ \theta_i \quad (i \leq p) \tag{1}$$

$$subject\ to\ C_i \quad (i \leq q) \tag{2}$$

Optimization objectives (eq. 1) and constraints (eq. 2) can vary depending on operators, tenants, and end-users [15,25,28,32]. In practical scenarios, this optimization problem of TE (§3.3) is computationally intractable [10]. Exact or approximation algorithms are costly to implement in the data plane (especially in PISA hardware). Considering this, existing TE systems mostly apply heuristic-based techniques to achieve the optimization of the objectives. In this work, we also followed same approach in designing P4TE.

4 PISA & Lmitations for TE

The Protocol Independent Switch Architecture (PISA) is a pipeline of several components for packet processing. At first, a packet passes through a programmable packet header parser where the header fields are extracted and then the packet traverses the ingress stage of the pipeline. Ingress stage consists of n mat-action stages, where $(0 \leq n \leq total\ match\text{-}action\ stages\ available)$. Each match-action stage can match packet header fields, and metadata with control plane configured values using match-action-table (MAT). Depending on the result of matching. a limited number of actions (stateless or stateful) can be executed. In the ingress stage of the pipeline, the egress port is determined and written in the metadata field. After the ingress stage, the packet passes through the egress stage. The egress stage is similar to the ingress stage in architecture. But, the egress port can not be modified after a packet enters the egress stage. Moreover, egress queueing information is only available at this stage. Finally, a packet goes through the deparser and scheduler before being transmitted. High throughput switches deploy multiple pipelines for packet processing, and control plane runs in a separate thread.

To maintain Tbps scale throughput. PISA switches are designed with several key architectural restrictions [4,9]. These restrictions raise several challenges for TE algorithms. Below, we have listed a few of the most important among them. Though we have discussed these restrictions in the context of RMT architecture, these restrictions are highly likely to stay for future high throughput programmable switch architectures.

4.1 Limited Processing Power

To maintain fixed per-packet processing delay, RMT pipelines are designed with a limited number of stages and limited number of actions per stage. These devices do not support complex actions (i.e., floating-point operation or division) and complex branching (nested if-else) in a single stage. Existing complex algorithms for traffic optimizations often require more processing budget than what programmable switches can afford in the pipeline.

4.2 Memory Size & Access Limitation

In PISA, a small amount of costly SRAM is dedicated to each stage in the pipeline. In 1GHz speed, only one read-write can be accommodated in each stage. As a result, a packet has to face a strict stateful memory access budget in the pipeline, which may not be sufficient for implementing existing TE algorithms. To avoid read-write hazard caused by concurrent memory access to the same memory location, sharing access to same memory blocks from multiple stages are prohibited. As important link performance metrics (queue depth, port utilization rate, etc.) are available only at the egress stage, and from the ingress stages, they are not accessible. This directly impacts TE algorithms

4.3 Inability to Modify MAT from Data Plane

The main mechanism of programmability in RMT architecture is *MAT*, and they are configurable (write/update) only from the CP. Forwarding path selection logic is expressed through MAT. For traffic-aware path selection a TE scheme needs to adjust forwarding paths and their priorities based on the data plane's feedback. It results in a delay between actual events in the data plane and the CP's MAT modifications. Moreover, in most cases, the update process is not atomic, which leads to inconsistent MAT configurations.

4.4 Lack of High Level Programming Constructs

One direct implication of the first limitation (§ 4.1) is, loop-construct can not be supported in the data plane to maintain a small per-packet processing budget. But, various dynamic data structure (queue, stack, graph, etc.) or most basic CPU-based algorithms (sorting, searching, etc.) can not be implemented without a loop. Due to the lack of these constructs, the existing rich set of works on TE need to be redesigned for use in programmable switches. Moreover, important operations for event-based packet processing [20](i.e., timer, generic rate calculator, etc.) are yet to be available in commercially available programmable switches.

4.5 Lack of Event-Driven Programming

In PISA switches, any kind of activity or event identification can only be triggered by a packet that is undergoing processing in the pipeline. But some important TE algorithms need to execute operations independent of the packet in the processing pipeline. Such as periodic measurement of link utilization, measuring flow rate or queue occupancy in a specific time interval, clearing previous flow states from stateful memory once a flow timeout, etc. These features are not available in PISA switches [20].

4.6 Inability to Generate New Packet

For out-of-band communication (needed for conveying any traffic event-related information) with neighbor switches or the control plane, on-demand new packet generation from the data plane is required. In currently available PISA switches a new packet can only be generated by replicating or recirculating an existing packet. Due to run-to-completion style architecture, newly generated packets need to undergo full processing in ingress and (or) egress stage. If the number of packets generated by replicating or recirculating existing packets are not controlled, a packet and new packet generated from it, they together can reduce the effective throughput of a pipeline into half or even worse.

Altogether, these factors limit the spectrum of deployable algorithms for TE on programmable switches. Either existing algorithms are needed to be carefully redesigned or new algorithms have to be developed for conforming to the limitations of programmable switches.

5 P4TE: Overview

The key idea behind P4TE is *incoming traffic* and *link performance aware* decision making at each hop for *path selection* and *rate controlling* of the flows. At a high level, P4TE works in two phases:

- At the *initialization* phase, the control plane provides preliminary configuration for different modules of P4TE.
- At the *runtime* phase, the data plane of each switch monitors 3 link performance metrics (hop-to-hop delay, link utilization rate and queue depth) and reports relevant events to the CP (switch local CPU). The DP also monitors per traffic class (pre-configured traffic classes) incoming traffic rate for each port. Next, the CP reconfigures the path priorities based on these monitoring results. The DP selects an appropriate forwarding path for every packet using a heuristic-based distributed (switch-local) algorithm. Lastly, depending on spare link bandwidth availability, the DP increases/decreases a flow rate by sending a fake ACK (FACK) packet to the sender of the flow with a modified TCP window size.

6 P4TE: Details

In this section, we describe P4TE's architecture in details, challenges faced and design choices to overcome them.

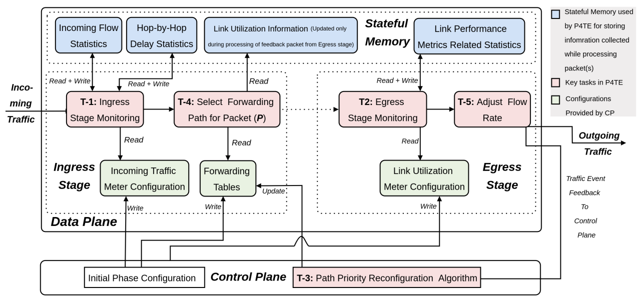

Incoming Flow Statistics
Hop-by-Hop Delay Statistics
Link Utilization Information (Updated only during processing of feedback packet from Egress stage)
Stateful Memory
Link Performance Metrics Related Statistics
Stateful Memory used by P4TE for storing infomration collected while processing packet(s)
Key tasks in P4TE
Configurations Provided by CP
Read + Write
Read + Write
Read
Read + Write
Inco-ming Traffic
T-1: Ingress Stage Monitoring
T-4: Select Forwarding Path for Packet (P)
T2: Egress Stage Monitoring
T-5: Adjust Flow Rate
Read
Read
Read
Outgoing Traffic
Ingress Stage
Incoming Traffic Meter Configuration
Forwarding Tables
Update
Link Utilization Meter Configuration
Egress Stage
Traffic Event Feedback To Control Plane
Write
Write
Write
Data Plane
Initial Phase Configuration
Control Plane
T-3: Path Priority Reconfiguration Algorithm

Figure 3: P4TE Architecture

P4TE's high-level architecture with mapping to major components of PISA switches is shown in Fig. 3. As depicted in 3, P4TE's architecture is organized around 5 key tasks (***T-1 to T-5***). It is organized into three components: monitoring (§ 6.1), path controlling (§ 6.2), and rate adaption (§ 6.4). Through the rest of this section, we discuss these 3 components in detail.

6.1 Monitoring

In P4TE, monitoring accomplishes 2 key tasks: a) **T-1** Ingress Stage Monitoring: monitors per port incoming traffic rate for preconfigured traffic classes (§ 6.1.4) and monitors hop-to-hop delay (§ 6.1.2) for each port. b) **T-2** Engress Stage Monitoring: monitors 2 link performance metrics: egress queue depth (§ 6.1.1) and link utilization rate (§ 6.1.3).

In P4TE, the goal of monitoring is to identify *events* happening in the data plane of a switch and provide quantitative information about them to the control plane. Achieving this in a scalable manner is a challenge due to PISA's architectural limitations (§ 4.6). Moreover, due to the lack of event-based programming support (§ 4.5), the execution of monitoring logics in the data plane can only be initiated through a data packet. Out-of-band probe packet-based monitoring is a widely used choice [18, 24] for this purpose,as this approach does not need to share resources in the pipeline with a normal data packet. However, during the interval between 2 successive probes, events happening in the data plane can be lost. To avoid these issues, P4TE uses a *gradient-based in-band* style event identification approach for monitoring.

In this approach, while processing each data packet, P4TE measures selected metrics and compare with previously stored value. If the previously stored value is different from the current value by some threshold, P4TE considers this an ***event*** and reports event-related information to CP or neighbor switch in a ***feedback*** packet. As in-band monitoring activity shares the processing budget in a pipeline with normal data packets, P4TE needs to minimize monitoring-related processing. Next, we describe P4TE's design choices behind monitoring selected metrics, challenges faced, and how P4TE handles them.

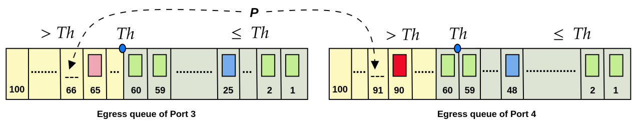


Figure 4: Static threshold (Th) based congestion marking

6.1.1 Egress Queue Depth

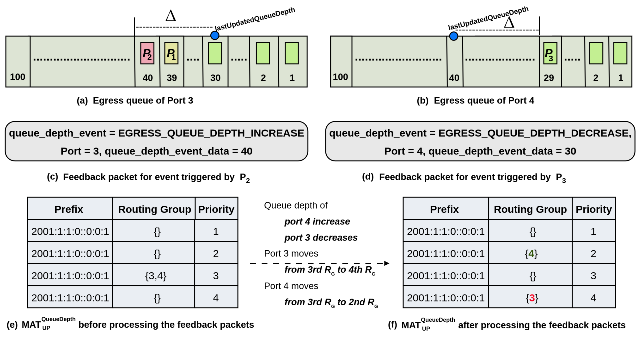
For measuring congestion over a single link, existing solutions mostly use a fixed number [3, 14] of static threshold values of queue depth. These schemes consume limited resources in the programmable switch pipeline but only suitable for binary style decision making (congestion exists or not). They can not support P4TE's goal of identifying congestion and quantitative information about a link's performance.

Consider, 2 upward ports of a switch (Fig. 4) are configured with a queue capacity of 100 packets and ECN threshold (Th) of 60 packets. Th divides the queue into 2 regions: $\leq Th$ packets in the queue and $> Th$ packets in the queue). Now, assume a new packet (P) arrives at this switch while the total number of packets waiting to be forwarded in these port's queue are 65 and 90. A single static threshold-based system will mark both the ports as equally congested, and P falls into region 2 for either port. But, port 4 is more congested, and pushing more traffic through this port increases the chance of an unbalanced load over the links. This clearly shows, static threshold-based yes/no style congestion identification can not provide enough information to compare multiple link's performance.

Using a single static threshold can not capture the traffic dynamics also. Consider, the switch (Fig. 4) is experiencing a low incoming traffic load compare to the previous case, and the upward ports have 25 and 48 packets waiting in queue (both $\leq Th$). For forwarding P, single static threshold-based systems will mark both the links as un-congested. Hence, both the links will be considered of the same quality and uncongested, which is clearly not true. The same argument is also applicable when the switch faces a high load (example discussed in the previous paragraph). Alternatively, thresholds can be configured dynamically (increase/decrease the threshold value proportionally with traffic load) through CP using stateful memory. This can help capture traffic dynamics in the cost of an extra memory operation (reading the threshold value from stateful memory before comparing) per threshold. However, it can not mitigate the previous problem.

Instead of a single threshold, the use of multiple static/dynamic (configured through CP) thresholds also can not overcome these problems totally. For example, the use of 2 thresholds (60 and 80 packets in the queue) divides a port's queue into 3 regions ([0,40], (40,60] and (60,100]). Problems mentioned above can occur for any two consecutive regions also.

As a dynamic approach, the queue depth observed by each


Δ
lastUpdatedQueueDepth
P_2
P_1
P_3
100
40
39
30
29
2
1
(a) Egress queue of Port 3
(b) Egress queue of Port 4
(c) Feedback packet for event triggered by P_2
(d) Feedback packet for event triggered by P_3
Queue depth of
port 4 increase
port 3 decreases
Port 3 moves
from 3rd R_G to 4th R_G
Port 4 moves
from 3rd R_G to 2nd R_G
(e) $MAT_{UP}^{QueueDepth}$ before processing the feedback packets
(f) $MAT_{UP}^{QueueDepth}$ after processing the feedback packets

queue_depth_event = EGRESS_QUEUE_DEPTH_INCREASE
Port = 3, queue_depth_event_data = 40

queue_depth_event = EGRESS_QUEUE_DEPTH_DECREASE,
Port = 4, queue_depth_event_data = 30

Prefix	Routing Group	Priority
2001:1:1:0::0:0:1	{}	1
2001:1:1:0::0:0:1	{}	2
2001:1:1:0::0:0:1	{3,4}	3
2001:1:1:0::0:0:1	{}	4

Prefix	Routing Group	Priority
2001:1:1:0::0:0:1	{}	1
2001:1:1:0::0:0:1	{4}	2
2001:1:1:0::0:0:1	{}	3
2001:1:1:0::0:0:1	{3}	4

Figure 5: Example queue depth monitoring (a) to (d). Corresponding path priority reconfiguration (e) to (f))

packet can be sent back to CP. CP can store and compare queue depth for multiple links to compare their relative quality. Due to the limitation mentioned in § 4.6, this will reduce the pipeline's throughput to half of its capacity. Even using static threshold (Th) can not help in this case. When more than Th packets are waiting in a port's queue; inserting a new packet in the queue will trigger an event, and a new packet is needed to be created for sending event information to CP. This process continues until queue depth drops below Th. This will also reduce the pipeline's throughput to half when the link is facing a high load.

Considering these factors, P4TE uses algorithm 1 (line 1-10) for monitoring egress queue depth. This algorithm tracks maximum or minimum queue depth seen by any packet at any port's queue within a window of 2Δ and stores it in an array of stateful memory ($oldQueueDepths$). If the queue depth seen by a packet at a port's queue differs from the stored value by a margin of Δ, P4TE considers it as an egress queue depth change event and updates the port's value in $oldQueueDepths$. Consider the example of Fig. 5-a) and 5-b), the upward ports (port 3 and 4) are configured with a queue capacity of 100 packets and $\Delta = 10$. 3 packets P_1, P_2 and P_3 enter into the switch; P_1 and P_2 are set to be forwarded through port 3 and P_3 through port 4. At port 3, P_1 observes egress queue depth 39, which doesn't differ with $lastUpdatedQueueDepth$ by Δ. Hence no event is identified by the switch. Whereas, egress queue depth seen by P_2 at port 3 is 40, which differs from $lastUpdatedQueueDepth$ by Δ. Hence, the switch considers this an egress queue depth increase event and stores it in packet metadata. Similarly, P_3 at port 4 sees egress queue depth of 30, which is a decline from $lastUpdatedQueueDepth$ by Δ. P4TE considers this as an *egress queue depth decrease* event. Information for these events is sent to CP through a feedback packet (Fig. 5-c and 5-d).

Use of Δ ensures at most one feedback packet is generated for Δ packets. This helps to keep the number of feedback packets under control and helps to avoid reduction in packet processing throughput of the pipeline. Moreover, the use of Δ effectively divides the whole range of egress queue depth into multiple regions, similar to the use of multiple thresholds. But with the added advantage of fewer comparisons and one stateful memory read-write without CP intervention. Comparing the queue depth seen by each packet with the last seen maximum or minimum queue depth value for that port (instead of static thresholds) gives 2 advantages. Firstly, the maximum/minimum queue depth at any port's queue is directly proportional to the traffic load. Hence, its use helps to cope with traffic dynamics. Secondly, once a traffic event is identified, relevant queue depth and port information are sent to CP. CP can use these values to compare the quality of the links (§ 6.3). These modifications over a static threshold-based ECN marking enables scalable monitoring of egress queue depth events and collecting relative quality of the links in the data plane.

Algorithm 1: Egress Stage Monitoring Algorithm (old_queue_depths & old_packet_colors both arrays are intitialized with 0)

```
1  queue_depth_event = Null ;
2  last_updated_queue_depth = old_queue_depths[P.egress_port] ;
3  if $P.queue\_depth \geq last\_updated\_queue\_depth + \Delta$ then
4      queue_depth_event = QUEUE_ DEPTH_ INCREASE;
5      queue_depth_event_data = P.queue_depth ;
6      old_queue_depths[P.egress_port] = P.queue_depth;
7  else if $P.queue\_depth < last\_updated\_queue\_depth - \Delta$ then
8      queue_depth_event = QUEUE_ DEPTH_ DECREASE;
9      queue_depth_event_data = P.queue_depth ;
10     old_queue_depths [P.egress_port] = P.queue_depth;
11 link_utilization_rate_event = Null ;
12 last_updated_packet_color = old_packet_colors[P.egress_port] ;
13 new_packet_color = match P.egress_port with $MAT_{Rate}^{Link\_Util}$ , execute
     matching direct meter and get the color ;
14 old_packet_colors[P.egress_port] = new_packet_color;
15 if $last\_updated\_packet\_color < new\_packet\_color$ then
16     link_utilization_rate_event = UTILIZATION_RATE_INCREASE ;
17 else if $last\_updated\_packet\_color > new\_packet\_color$ then
18     link_utilization_rate_event = UTILIZATION_RATE_DECREASE ;
```

6.1.2 Hop-to-Hop Delay

Hop-to-hop delay is a combined effect of various decision factors (scheduling algorithm, congestion, incoming traffic burst, and many more) at the source switch. For monitoring ***hop-to-hop delay***, after receiving a packet P from end-hosts, leaf switches add *ingress-timestamp* as a header field (T^{Source}) in P before forwarding it toward the destination switch (next hop) through $P^{Source}_{Egress\_port}$. After reciving P, the destination switch parses and subtracts T^{Source} from P's own *ingress-timestamp* (T^{Dest}). $T^{Dest} - T^{Source}$ is the ***hop-to-hop*** delay for $P^{Source}_{Egress\_port}$'th port of source switch. For monitoring **hop-to-hop** delay, issues faced in monitoring *egress queue depth* arises again and P4TE follows a similar algorithm (algorithm 2: line 1-10). On identifying a hop-to-hop delay event, the destination switch sends the event information to the source switch. Source switch redirects the feedback to its local control plane for further processing (§ 6.3).

6.1.3 Link Utilization Rate

P4TE monitors each link's utilization rate at the egress stage and use the monitored information for 2 purposes: a) for reconfiguring the path priorities based on their utilization rate (§ 6.3) and b) in the path selection algorithm to verify whether a selected link is overutilized or not. Issues arising in queue depth monitoring (§ 6.1.1) also arise here. Moreover, instead of monitoring the exact rate, current programmable switches only support fixed-rate (1 or 2 rates) based meters (§ 4.5). Rate configurations for these meters can be updated only from CP. But, as the update process is not atomic [], meters can provide inconsistent information during the update. Consider, port 3 and 4 of a switch are configured with 40 Gbps CIR and 80 Gbps PIR. At some point, traffic load changes, and we want to update these rates to 50 and 90 Gbps for both ports. Assume, meters for port 3 are updated with new rate information while the update for port 4 is still pending. At this moment, if both the link utilization rate is 45 Gbps, packet forwarded through port 3 will be marked in GREEN color while packet forwarded through port 4 will be marked in YELLOW color. This leads to a wrong perception that port 3 is underutilized compared to port 4. Moreover, Moreover, P4-meters do not provide any quantitative information; they only color mark the packets. As a result, monitoring and comparing link qualities based on utilization rate using dynamically configured p4-meters is not possible.

Considering these factors, P4TE uses *2 rate 3 color*-based meter [17] for measuring the utilization rate of each port. To avoid the problem of the inconsistent update, rate (*Committed Information Rate* (CIR) and *Peak Information Rate* (PIR)) configurations are kept *fixed* over the whole life cycle of P4TE. For identifying *link utilization rate* related events, P4TE follows algorithm 1 (line 11-18). Due to fixed-rate configurations, this algorithm can not adapt with varying traffic loads like queue depth monitoring (§ 6.1.1). But, P4 meters support burst (*Committed Burst size* (CBS) and *Peak Burst Size* (PBS)) handling capability near CIR and PIR. Burst rates in meter configuration serve the same purpose as Δ in queue depth monitoring (§ 6.1.1). It triggers only one traffic event for a traffic burst of size CBS (or PBS) near CIR (or PIR). This prevents reduction in throughput caused by exccesive number of feedback packet near the CIR or PIR.

For avoiding overutilization of a link, P4TE's path selection algorithm (§ 6.2.1) verifies the utilization rate of a link in the ingress stage before selecting it for a flowlet. But link utilization information is only available in the egress stage. Due to limitation (§ 4.2) of PISA switches this information is not available in the ingress stage. To make them available at ingress stage, on identifying link utilization rate related event, besides sending a feedback to CP, P4TE recirculates a copy of the feedback packet to the ingress stage (algorithm 1 line 11-18). Once the recirculated feedback packet reaches ingress stage, P4TE stores the link utilization information of the port in an array of stateful memory (*port_utilizations*) for use in the path selection algorithm.

6.1.4 Traffic Class Based Incoming Rate

P4TE follows a traffic class-based fairness approach. If flows from a traffic class are allocated too much resource, in case of resource contention, it can starve flows of other traffic class . To avoid this, each traffic class is allocated a *safe-rate* for each port, and the incoming traffic is monitored. Issues arising in § 6.1.3 also arises here, and P4TE uses 1 rate 2 color-based P4 meter. At the *initialization phase*, P4TE configures this per traffic class rate information (Fig. 2-d) for P4 meters of each port. At the *runtime phase*, if incoming flows through a port crosses the configured rate for its traffic class, P4TE marks (algorithm 2 line- 11) the packet in YELLOW (GREEN otherwise) color to indicate the flow as *unsafe (safe otherwise)*. The color mark is stored in packet metadata (*incoming_packet_color*) for use in rate control (§ 6.4).

Algorithm 2: Ingress Stage Monitoring Algoroithm

```
1  delay_event = Null;
2  old_delay = old_delays[P.egressPort] ;
3  if (P.ingress_timestamp - $T^{Source}$ ) $\geq$ oldD_delay + $\delta$ then
4      delay_event = HOP_BY_HOP_DELAY_INCREASE ;
5      delay_event_data = P.ingress_timestamp - $T^{Source}_{Ingress}$ ;
6      old_delays[P.egress_port] = delay_eventdata;
7  else if (P.ingress_timestamp - $T^{Source}$ ) < old_delay - $\delta$ then
8      delay_event = HOP_BY_HOP_DELAY_DECREASE ;
9      delay_event_data = P.ingress_timestamp - ${T_{Ingress}}^{Source}$ ;
10     old_delays[P.egress_port] = delay_event_data;
11 incoming_packet_color = match P.ingress_port & P.traffic_class with
     $MAT^{In}_{Rate}$ , execute matching P4 direct-meter and get the color ;
```

6.2 Path Controlling

Path controlling includes 2 tasks of P4TE: a) T-4: path selection for a packet b) T-3: priority reconfiguration of available paths. P4TE uses distributed (using switch local CPU) algorithm for both of them.

6.2.1 Path Selection

P4TE uses a modified version of the existing 2 level route lookup algorithm [1]. Next, we describe the algorithm in 2 parts: downward and upward .

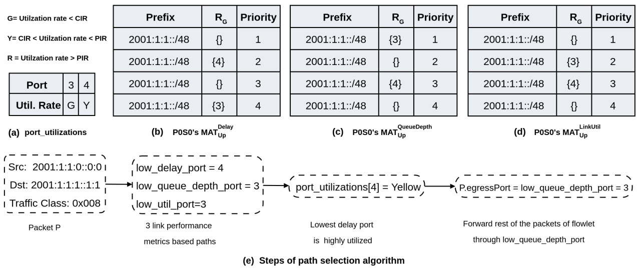
6.2.1.1 Downward: Fat-tree topology contains no path diversity in the downward direction, and P4TE uses IP prefix-based matching here. During the *initialization phase*, at each leaf switch, P4TE configures the match-action-table (MAT_{down}) for downward route lookup with an explicit path for all connected end-hosts with their corresponding IP address and port. Similarly, spine switches and super spine switches are initialized with the common IP prefix and port of the subnets covered by connected leaf switches and spine switches. Fig. 2a, 2b and 2c show examples of MAT_{down} for leaf, spine, and super spine switch. Once configured, entries in (MAT_{down}) remains fixed through the full life cycle of P4TE. For path selection, a packet is at first matched against the match-action-table (MAT_{down}) for downward route lookup. If a matching entry is found, the packet is forwarded through the corresponding port. Otherwise, the packet goes through upward routing algorithm (§ 6.2.1.2) for path selection.

6.2.1.2 Upward:

Fat-tree topology provides path diversity in the upward direction at each switch. TE's success depends on the performance-aware split of traffic over multiple paths in the existence of path diversity. In programmable switches, this is a non-trivial task due to several reasons. Assume the simplest scenario at any hop; there are n available links toward a packet's destination with only one performance metric to monitor. Based on the monitoring result, maintaining the links in sorted order and selecting the best one entirely in the data plane requires loop support. Which is unavailable (§ 4.4) in programmable switches. The only viable option is sending monitored information to CP. CP can arrange the n links in match-action-tables (MAT) in sorted order based on that information. The DP selects the best next hop for a packet using the MAT. Once traffic is forwarded, the relative quality of the links may get changed. The process of updating the relative order of n links in the data plane from CP may lead to inconsistent behavior (§ 4.3).

Now, assume for a flow (f), the best path selected using MAT is f_{path}. Due to the reconfiguration delay (§ 4.3), f_{path} may not be the best path for achieving TE objectives. From selecting f_{path} for f to *verifying*, whether using f_{path} will achieve TE objectives or not requires the following steps: a) select f_{path} for f using MAT/stateful memory b) read f_{path}'s condition according to performance metrics for verifying whether it is conforming to the TE objectives or not c) if f_{path}'s use works against achieving TE objectives, then select a new path ($\bar{f}_{path}$) for the flow and, d) finally, update $\bar{f}$'s state in stateful memory so that next packets of f uses $\bar{f}_{path}$. Clearly, steps a), b), and d) create *true dependency* and can not be rewritten in any other order. But a pipeline with 2 memory ports can afford only one read-write in a single stage at 1 GHz clock speed (§ 4.2). As a result, this *true dependency* is impossible to overcome in the data plane. Even if the number of memory access per clock cycle increases in the future, the number of memory accesses required for step b) can always cross the available memory access capacity. Hence, the fundamental problems for the traffic-aware path selection and verification will always remain the same.

To overcome these limitations, P4TE modifies existing flowlet based upward-route lookup techniques for fat-tree topologies in 3 ways. Firstly, instead of maintaining a direct mapping from IP prefix to upward ports in match-action-table (MAT), P4TE maintains the upward ports in n link performance metrics based **Routing-Groups** (R_G). The key idea is dividing the range of a link performance metrics into n predefined sub-ranges and allocating one *routing-group* for each sub-range. Each *routing-group* is a set of ports/links through which a packet can be forwarded. Every port in same *routing-group* has an equal priority of being selected (Hash based selection). However, each of the $n(n \geq 1)$ *routing-group* have a different priority, which determines their order of being selected in MAT. Membership of ports in a *routing-group* is determined based on performance metrics of the links collected by the monitoring component (§ 6.1). Performance metrics to *routing-group* mappings are defined independently of data plane logic. Hence, each port can be mapped to any *routing -roup* independently of other ports. The removes the problem of inconsistent state caused by *non-atomicity* of updating MAT. Moreover, the CP can use any customized algorithm to map the ports to routing groups to achieve other TE objectives. Secondly, instead of using only one routing table, P4TE uses 3 separate routing tables (MAT_{Up}^{Delay}, $MAT_{Up}^{Queue\_Depth}$ and $MAT_{Up}^{Link\_Util}$) to arrange upward links in routing-groups according to 3 monitored link performance metrics (hop-to-hop delay, egress queue depth and link utilization rate). Instead of ECMP like blind path selection, this provides the opportunity of selecting a path for a flow based on one of these 3 metrics. Finally, instead of using static hash-based or only one best path [24] for all flowlets, P4TE uses a heuristic-based strategy for selecting the path for flowlets of different traffic classes.


G= Utilzation rate < CIR
Y= CIR < Utilzation rate < PIR
R = Utilzation rate > PIR
(a) port_utilizations
(b) P0S0's MAT$_{Up}^{Delay}$
(c) P0S0's MAT$_{Up}^{QueueDepth}$
(d) P0S0's MAT$_{Up}^{LinkUtil}$
Src: 2001:1:1:0::0:0
Dst: 2001:1:1:1::1:1
Traffic Class: 0x008
Packet P
low_delay_port = 4
low_queue_depth_port = 3
low_util_port=3
3 link performance metrics based paths
port_utilizations[4] = Yellow
Lowest delay port is highly utilized
P.egressPort = low_queue_depth_port = 3
Forward rest of the packets of flowlet through low_queue_depth_port
(e) Steps of path selection algorithm

Port	3	4
Util. Rate	G	Y

Prefix	R_G	Priority
2001:1:1::/48	{}	1
2001:1:1::/48	{4}	2
2001:1:1::/48	{}	3
2001:1:1::/48	{3}	4

Prefix	R_G	Priority
2001:1:1::/48	{3}	1
2001:1:1::/48	{}	2
2001:1:1::/48	{4}	3
2001:1:1::/48	{}	4

Prefix	R_G	Priority
2001:1:1::/48	{}	1
2001:1:1::/48	{3}	2
2001:1:1::/48	{4}	3
2001:1:1::/48	{}	4

Figure 6: Upward path selection algorithm

At the ***Initialization Phase***, P4TE initializes all the 3 match-action-table (MAT) for upward routing: MAT_{Up}^{Delay}, $MAT_{Up}^{Queue\_Depth}$ and $MAT_{Up}^{Link\_Util}$. For each of the MAT, P4TE creates n empty **Routing-Groups** (R_G). For each of these groups, a ternary match entry containing the common IP prefix of the whole DCN, a pointer to the routing-group, and priority of the routing-group is inserted in the MAT. Initially, there is no traffic flow through the upward ports. Hence, P4TE considers all upward ports of the same performance at startup. These ports are inserted in the highest priority routing-group and all other routing-groups remain empty. Fig. 2 shows an example of upward routing tables just after the initialization phase.

At the ***Runtime phase***, P4TE executes algorithm 3 to select a forwarding path for a flowlet. In selecting the path for flowlets, P4TE always tries to use a path that has a low chance of overutilization. For *latency-sensitive short flows*, it tries to use the port that can forward the packet with the lowest delay. However, if that port has already seen high utilization (YELLOW or RED status), the flowlet is assigned a port with the lowest queue depth or a port with the least utilization. Both the selection reduces the chances for overutilization of a link. On the other hand, for *throughput-oriented large flows*, P4TE gives the highest priority on selecting the least utilized port with the hope that this will leave enough link capacity for pushing more packets of the flow without over-utilizing the link. Fig. 6 shows an example explaining how the path for a flowlet of *latency-sensitive short flow* is selected.

Algorithm 3: Upward Path Selection Algortihm for Packet P

```
1  low_delay_port = select matching port from $MAT_{Up}^{Delay}$ ;
2  low_queue_depth_port = select matching port from $MAT_{Up}^{Queue\_Depth}$ ;
3  low_util_port = select matching port from $MAT_{Up}^{Link\_Utilization}$ ;
4  flowlet_Id = calculate 5 tuple (src IP, src Port, dest IP, dest Port, flow label)
     based hash code ;
5  last_seen_time = last_seen_time_register_array[flowlet_Id] ;
6  last_seen_time_register_array[flowlet_Id] = P.ingress_timestamp ;
7  flowlet_interpacket_gap = P.metadata.ingress_timestamp - last_seen_time ;
8  if P.traffic_class == TRAFFIC_CLASS_LOW_DELAY then
9      if flowlet_interpacket_gap $\geq$
         FLOWLET_INTER_PACKET_GAP_THRESHOLD then
10         status = port_utilizations[low_delay_port] ;
11         if status == GREEN then
12             P.egress_port = low_delay_port ;
13         else if status == YELLOW then
14             P.egress_port = low_queue_depth_port ;
15         else if status == RED then
16             P.egress_port = low_util_port ;
17         last_used_ports[flowlet_Id] = P.egress_port ;
18     else
19         P.egress_port = last_used_ports[flowlet_Id] ;
20 else if P.traffic_class == TRAFFIC_CLASS_HIGH_THROUGHPUT then
21     if flowlet_interpacket_gap $\geq$
         FLOWLET_INTER_PACKET_GAP_THRESHOLD then
22         status = port_utilizations[low_util_port] ;
23         if status == GREEN then
24             P.egress_port = low_util_port ;
25         else if status == YELLOW then
26             P.egress_port = low_queue_depth_port ;
27         else if status == RED then
28             P.egress_port = low_delay_port ;
29         last_used_ports[flowlet_Id] = P.egress_port ;
30     else
31         P.egress_port = last_used_ports[flowlet_Id] ;
32 else
33     P.egress_port = low_queue_depth_port ;
```

6.3 Path Priority Reconfiguration

Initially, P4TE assumes all upward ports have the same quality, and they are in the same routing-group. With traffic flow, the link's relative performance gets change, and the monitoring component reports these events to CP through a feedback packet. A feedback packet may contain event information for more than one link performance metric. For all link performance INCREASE/DECREASE event information for link performance metrics M ($M \in Delay, Queue\_Depth, Link\_Utilization$) in the feedback packet (f), CP executes algorithm 4. This algorithm moves a port to a lower (or higher) priority routing-group if the link performance metrics decreases (or increases). Fig. 5-(e) to (f) shows an example of reconfiguring priorities of the links w.r.t. their queue depth for the feedback packets shown in 5-c) and 5-d).

When all the links face a high load, they are assigned to the lowest priority routing-group. Similarly, all links are assigned to the highest priority routing-group (or any other routing-group) in low load. In both cases, all the links exist in the same routing group, and the forwarding path for a packet is selected using hash-based mapping. Hence, in such cases, P4TE behaves like ECMP. It ensures, P4TE does not perform worse than ECMP when extreme congestion occurs on any switch in the DCN.

Algorithm 4: Path priority reconfiguration algorithm for event E

```
1  Initialization Phase:
       /* This segment is executed by CP */
2      for $i \leftarrow 1$ to Total Ports in Switch do
3          port_to_routing_group_map$^M$[i] = Highest priority
             routing-group ;
4      end
5  end
6  Runtime Phase:
       /* Following segment is executed by CP after feedback
          message is receivied from DP for a metrics $\mathcal{M}$ */
7      old_routing_group_id = port_to_routing_group_map$^M$[E.port] ;
8      new_routing_group_id = Null ;
9      if E.event_type == M_INCREASE then
10         if old_routing_group_id < max_group_id then
11             new_routing_group_id = old_routing_group_id + 1;
12     else if P.event_type == M_DECREASE then
13         if old_routing_group_id > min_group_id then
14             new_routing_group_id = old_routing_group_id - 1;
15     if new_routing_group_id != Null then
16         Use P4Runtime API to remove E.port from the routing-group
             with old_routing_group_id;
17         enter E.port into routing-group with id new_routing_group_id;
18 end
```

6.4 Rate Adaptation

P4TE's path selection algorithm tries to avoid overutilization of the links. But congestion starts to build up when total incoming traffic volume is greater than the outgoing capacity of the links and/or an overutilized link is selected for further traffic. On sensing congestion, TCP stacks at end hosts react by decreasing congestion window size. Similarly, in the absence of congestion, the TCP congestion window is increased to utilize the link capacities. However, these end host-based schemes require at least RTT times to react to network events. On the contrary, while forwarding a packet, if any P4TE enabled switch senses congestion or spare capacity available to forward more packets, it sends a *fake acknowledgment packets (FACK)* to the sender of the flow with decreased/increased window size. In the best case, the sender receives the FACK with an updated rate from the first switch in its path and in the worst case from the last switch. It helps the sender to react to changing network conditions faster than RTT times. Before sending a FACK, a switch needs to answer 2 questions: a) how to identify candidate flow for rate control, and b) what should be the appropriate rate for a flow?

6.4.1 Flow Selection

Assume, for packet P of traffic class TC and flow f the egress port selected by path selection algorithm (§ 6.2.1) is i. To determine whether f's sending rate should be controlled or not, P4TE uses 2 important pieces of information collected by monitoring component: a) whether all the incoming flow of a traffic class TC is crossing the safe-rate for that class ($ 6.1.4) (*incoming_packet_color* is GREEN/YELLOW) and b) utilization rate of link i ($old\_packet\_colors[i]$) (§ 6.1.3).

If f finds the *safe-rate* for its traffic class and incoming port is crossed (*incoming_packet_color* is YELLOW) and the link i is overutilized ($old\_packet\_colors[i] \neq GREEN$) P4TE considers this as a sign of congestion. P4TE enabled switch immediately tries to reduce the flow's sending rate by sending a FACK with reduced window size on sensing congestion and also ECN marks the original data packet toward destination. Now crossing the safe-rate for traffic class TC implies that traffic has already used the allocated portion of that class. However, there may be no other competing flow for the selected egress port. If port i is not overutilized ($old\_packet\_colors[i] == GREEN$), P4TE tries to utilize the selected link's spare capacity by increasing the flow's sending rate by sending a FACK with increased window size. On the other hand, if flows from traffic class TC have not crossed the safe rate ($old\_packet\_colors[i] == GREEN$), but the port is i overutilized, P4TE does not take any flow control action. The justification behind this is, flows of traffic class TC has not used their allocated rate, and next packet's from other competing traffic classes will find port i is overutilized. In such a case, P4TE will reduce their rate.

To not increase the rate of a flow too much or penalize a flow unfairly compare to other flows of the same class, P4TE uses an approach similar to DCTCP [6]. But instead of end-host, all leaf switches (directly connected to hosts) maintain a record of the last TCP sequence number indicating when rate control was applied on a flow. The leaf switch marks a flag in the header to not apply rate control on the flow for the next B bytes. All other switches in the path do not apply rate control on this flow if the flag is set. Moreover, if a switch decides to reduce (or increase) the rate of a flow, it marks the header with a special flag to prevent other switches in the path from applying rate control on the flow. On the other hand, when a leaf switch recieves a FACK packet sent toward a host (connected to the downward ports of the leaf switch), the leaf switch updates the TCP sequence number These ensure a flow is not penalized or gets an unfair advantage multiple times within a window of B bytes.

6.4.2 Rate Control

Flow rate control needs participation from the protocol stack of both end-host and switch. Sending FACK from a switch with a modified window size reaches the sender of a flow earlier than the usual RTT time. Besides this, the sequence number in the FACK packet is not a real reflection of bytes acknowledged (also bytes in flights) by the receiver of the flow. Both the factors play a crucial role in determining sender-side congestion window by end-host transport layer protocol stacks. Hence, sending FACK does not provide exact information to the sender, impacting the sender-side congestion window calculation. Various end-host-based transport layer protocol stacks adjust flow rates using different formulas [35]. Controlling the broad dynamics of end-host-based protocol stack's congestion control scheme from the switch is a complicated topic. We leave it as future research scope. Considering these factors, P4TE does not make any assumption about the use of any specific end-host side trans- port layer protocol stack. P4Te follows the AIMD scheme, where the congestion window is reduced more aggressively and increased more cautiously. On identifying a candidate flow for rate control, P4TE reduces its window $\frac{1}{4}$ or increases the size of its window by $\frac{1}{256}$ of it's current size.

6.5 Robustness

Link failure and high traffic load are 2 of the main reason behind the underperformance of TE systems in DCN. Routing-Group based mechanism of P4TE handles each link independently and does not need a readjustment of other links due to failure in one link. When a link fails, the operator can simply remove the link from its current routing-group; P4TE will automatically readjust the load on other links. When the incoming load is too high, all the links can fall into the lowest priority routing-group of corresponding metrics. Or when the load is too low, all links can fall into the highest priority routing-group. In both cases, P4TE behaves similar to ECMP, and performance is at least equivalent to ECMP.

6.6 Mapping to Hardware

In this section, mapping of P4TE's workflow to RMT pipeline is discussed. Consider, $|k|$ = total number of ports in a switch, $|TC|$ = the total number of traffic classes and $|n|$ = total number of *routing-group* for a link performance metrics.

•**Monitoring (§6.1)**: Algorithm 1 and 2 both uses *if-else* logic and one operation per header/metadata field. Hence, each one of the 2 algorithms can be easily accommodated in 2 stages. Stateful memory used in Algorithm 1 (old_queue_depths, old_packet_colors, and direct meters for $MAT_{Rate}^{Link\_Util}$) are $O(k)$. Stateful memory consumed by Algorithm 2 (*old_delays*) is at most $O(k)$ and total MAT entries required (for MAT_{Rate}^{In}) is $O(k * TC)$. Considering the fact that, *number of traffic classes* is small; table size, stateful memory size, and the number of meters used in both algorithms are very small compared to available capacity in programmable switches [9].

•**Path Controlling (§6.2)**: Total number of entries required for the downward route lookup table (MAT_{Down}) is at most k. Each of the 3 link performance metrics-based tables used by P4TE for upward route lookup are independent of each other and require l entries. For each entry, there will be a routing-group of size at k. Capacity of 1024 entries in a TCAM based match-action-table is pretty common. For a link performance metrics 1024 routing-groups are more than enough. As an example, consider the case of 5Mb buffer capacity for a port. With 64B packet size, 5Mb buffer can accommodate at most 10k packets. With $\delta = 200$ in algorithm 1, maximum number of routing-groups required are around 50. Number of upward ports in a fat-tree topology rarely crosses half of the total ports available in a switch. Hence, Each of the routing-groups need to hold at most $k/2$ upward ports. For a k=512 port switch, required space for 50 routing groups is around 13K, which can be easily accommodated in a single stage. Moreover, as the 3 tables for upward route lookup are independent of each other, they can be executed parallelly. The traffic class based path selection logic (Algorithm 3) requires nested if-else logic. But all the values used in the *logical expressions* are available beforehand in metadata, and the number of traffic classes is very limited. Hence, the whole logic for class-based path selection can be easily implemented in a TCAM. Stateful memory requirement for maintaining flowlet id and timestamp for each flow is the same as ECMP algorithm.

•**Rate Control (§6.4)**: Though the total number of flows in a DCN can be large, the actual per-flow state for rate control is solely maintained by the leaf switches. For each flow, keeping a record of the last TCP sequence number when rate control was applied requires only 4 bytes. For 1M flows total stateful memory requirement is 4MB which is less than 2 x per stage available memory in programmable switches [9]. The logic for rate control requires only simple if-else logic based on some values already available in the previous stage of the pipeline and can be easily implemented in a single-stage.

–Besides these, some other common tasks for packet replication, NDP packets, some basic packet initialization tasks and to recirculation are required for each packet (ref to our gthub repo). Those logics are also based on some values already available in the previous stage of the pipeline and implementable in a limited number of MAT stages. A more detailed discussion regarding the resource consumption by P4TE in RMT pipeline is avaialble at (ref to our github directory). Altogether, P4TE can be deployed on programmable switches available in the market. Total resource consumed by P4TE in an RMT pipeline is small and leaves plenty of room for other functionality. Analysis of P4TE's data plane program and detailed discussion of mapping the code to RMT hardware is available at https://github.com/drobinkent/FullTechnicalReports/blob/master/P4TE/HW_mapping.pdf.

7 Performance Evaluation

In this section, we evaluate the performance of P4TE by comparing it with ECMP. We emulated a k=8-port switch-based layer-2 fat-tree DCN (leaf-spine topology) using Mininet and P4 bmv2 software switch. The switch's data plane and control plane program was developed using P4$_{16}$ and Python language. For CP to DP communication P4Runtime framework has been used. The source code for P4TE is available at https://github.com/drobinkent/P4TE. For creating a 2:1 link oversubscription ratio, the link bandwidth capacity of end-host to leaf switches, leaf to spine switches, and spine to super spine switches were set to 40, 20 and 10 pps (packets per second) respectively. Buffer capacities were configured as 2xlink bandwidth capacity with tail drop policy. All the experiments have been conducted on a machine with Intel(R) Core(TM) i7-9750H CPU (all 6 cores were config- ured at clock speed 2.4 GHz), 24 GB RAM, running Ubuntu 20.04. We emulated short latency-sensitive *short* flows (traffic class 0x10) of size 50KB and throughout-oriented *large* flow (traffic class 0x08) of size 256KB using IPerf3 [27].

DCN traffic pattern is dependent on various factors. Emulating various traffic distribution patterns using a large number of Iperf flows to simulate the exact DCN traffic scenario is impossible. Instead, we focused on 3 important traffic scenarios (§7.2.1, §7.2.2, §7.2.3) to put stress on the DCN. Similar to prior works [12] we consider FCT as the main performance metrics. Our goal was to focus on whether P4TE can improve FCT of the short flows without degrading FCT for large flows too much. But, IPerf clients start a flow and transmits all the data up to configured flow size. Once all data are transmitted in T seconds, IPerf client closes the TCP connection and any data unacknowledged at that time is counted as lost (T is considered as *flow completion time* (FCT), amount of unacknowledged data at the end of the experiment as *data loss* and amount of data acknowledged to the sender as the *data successfully sent*). For all the scenarios, we observed the amount of data loss are within twice the window size. This can be safely attributed to IPerf's early client socket closing behavior and we omitted a comparison of these metrics. As a side effect of the rate-control scheme applied by P4TE, FACKs are sent to the sender of a flow. It can increase the number of retransmissions. Considering these factors, besides FCT, we also compared total number of retransmissions for each scenario. Moreover we also compared P4TE's ability to distribute load over multiple paths compare to ECMP. All the values were taken over 10 iterations. We have conducted the simulations with various traffic patterns and different flow properties. WE have observed similar kind of performance gain from P4TE. Due to space concern, only the most important among them are presented in this paper. Rest of the results are accessible through our full technical report (**Will add all our results in the report and add refference**).

7.1 P4TE Paramaeters Setup

Traffic Class-Based Incoming Rate (§6.1.4): Study results [7, 22, 30] show that, more than 50% of DCN traffic is short flow, and the rest are large flows. To give more importance to short flows, we allocated 65% capacity of a link to short flows and rest for large flows.

Link Utilization Rate (§6.1.3): We configured the meters (Fig. 2f) to consider utilization of 70% of a link capacity as not over-utilized, 90% as a high chance of future overutilization and 100% as fully overutilized.

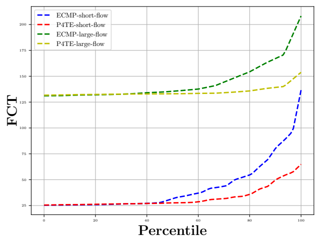
Number of routing-groups: For *hop-to-hop delay*, we measured the range of delay between 2 hops in the DCN. We divided this value into 3 sub-ranges and configured$\Delta = 15ms$ for algorithm 1. Similarly, for *queue depth*, we divided the maximum buffer capacity of the super spine switch ports into 3 ranges and configured $\delta = 2$ $(10/3 \approx 4)$ for algorithm 2. For measuring *link utilization rate*, we used 2-rate-3-color based P4 meters. As the meters can mark packets in only one of the 3 colors, we used 3 sub-ranges for link utilization rate. Algorithm 3 or 4 is not dependent on any fixed number of routing-groups. Hence it is possible to use a wide variety of scales (linear, exponential, etc.) to define the number of routing-groups and corresponding mapping of the ports according to link performance metrics.

7.2 Results Discussion

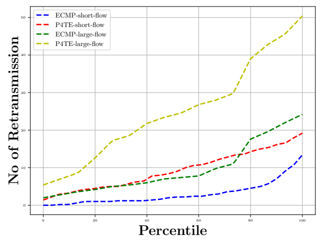
7.2.1 Stride Pattern

In this scenario, we initiated traffic flows according to stride pattern (similar to [2, 33]) for analyzing P4TE's performance over the whole DCN. We repeated the test for 2 different load factors: a) **low load**: 2 sets (8 flows each) of large flows at 75s intervals and 4 sets (8 flows each) of short flows at 100s intervals. b) **high load**: 2 sets (16 flows each) of large flows at 85 intervals and 4 sets (16 flows each) of short flows at 125 intervals.

Fig. 7a and 8a show that for both load factors, around 50 % flows achieves a shorter FCT. Data presented in table 1 P4TE achieves a better average flow completion time with a smaller standard deviation in FCT in both low and high load scenarios. In the low load scenario, short flows complete in average 30.57 seconds which is a 41% improvement over ECMP. P4TE achieves an average FCT of 134.63 sec for large flows, a 10.99% improvement over ECMP. The range of minimum and maximum FCT in both Fig. 7a and 8a illustrates P4TE's ability to improve fairness compare to ECMP. Data presented

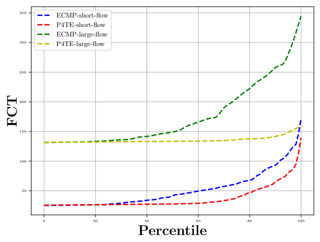


(a) Percentiles of flow number vs FCT

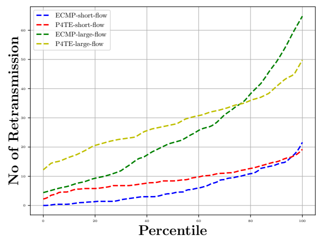


(b) Percentiles of flow number vs retranmissions

Figure 7: Performance comparison for **low load** traffic scenario of § 7.2.1



(a) Percentiles of flow number vs FCT



(b) Percentiles of flow number vs retranmissions

Figure 8: Performance comparison for **high load** traffic scenario of § 7.2.1

in table 1 (row 1) also shows that in both small and large load scenarios, compared to ECMP, P4TE has a smaller FCT standard deviation, indicating better fairness among the same traffic class flows.

Fig. 7b and 8b show that P4TE has a higher number of retransmissions compare to ECMP. This is caused by FACK packets used by P4TE for rate control. P4TE sends the last sequence number in FACK packets for which the sender has received acknowledgment from the receiver, which is not a correct reflection of real data acknowledged by the receiver. On receiving FACK packets with already acknowledged sequence numbers, end-host protocol stacks can initiate retransmission. A higher average number of retransmissions under P4TE for both short and large flows (table 1) reflects the design choice of P4TE. For large flows, we configured P4TE to consider congestion when large flows consume 35% of a link capacity. However, for short flows, this rate is 65%. Hence, short flows face rate control less frequently and in later hops of their path (due to oversubscription). As a result, P4TE imposes rate control more frequently for large flows, and it increases the average number of retransmissions for them. Under P4TE, a higher average number of retransmissions for large flows (18.375 for low load and 31.84 for high load scenario) compare to short flows (10.875 for small load and 9.65 for high load scenario) (table 1) illustrates this.

7.2.2 High Congestion

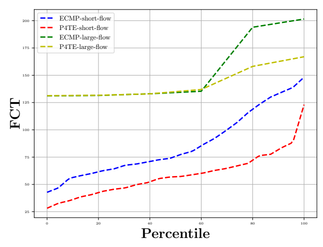
In this scenario, we started 6 large flows and 24 short flows from 2 hosts connected to the same leaf switch to 3 different hosts connected to 3 different leaf switches. The goal was to create congestion at various leaf and spine switches of the DCN and observe P4TE's performance at congestion scenario. Fig. 9a shows that both short and large flows achieve shorter FCT in congestion scenario under P4TE. P4TE achieves a smaller standard deviation of FCT for both short and large flows (table 1) row 5). This illustrates P4TE's capability of improving fairness among flows from the same traffic class.

Here, we started a large number of flows from the same set of source and destination, and they create a high load in all the switches from source to destination. Hence, all the upward links face high load and fall in the same routing-group (§ 6.3). In this case, P4TE starts behaving like ECMP but not worse than ECMP. As a result, FCT improvement under P4TE in a high congestion scenario is also better or equivalent to ECMP, but not worse. This behavior reflects the correctness of P4TE's path priority reconfiguration logic.

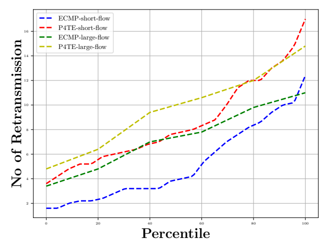
Fig. 9b and table 1) row 6 shows the number of retransmissions is higher under P4TE compare to ECMP. This can be explained based on the reasoning provided § 7.2.1. For large flows, the difference in the number of retransmissions between P4TE and ECMP in this scenario is less than the previous scenario. Because in this scenario, congestion starts happening at the early stages in the path. P4TE enabled switches identifies congestion early and reacts by sending FACK. As s result, the sender hosts get notified about congestion earlier than the RTT time delay of ECMP. This helps the senders to adjust the TCP windows size early. This scenario validates our goal of achieving sub-RTT reaction time (compare to RTT) to congestion and better performance than ECMP. But we allocated a larger portion of the link capacity as safe-rate for the short flows and in the leaf switches they do not face rate reduction (host to leaf switches have higher link capacity due to 2:1 oversubscription). Hence they try to send larger volume of data without facing rate controlling in the early stages of the path. But they face rate controlling in later stages and as a result of the fake ACK for rate controlling the number of retranmissions increases.

7.2.3 Incast

In this scenario, 16 short flows and 4 large flows were started from 4 different hosts to a *single host*. to observe P4TE's performance in TCP incast scenario. As all the 20 flows have the same destination, they create high congestion in the last hop in the path, directly connected to the destination host. Fig. 10a shows that in incast scenario, both short and large flows achieve improved FCT. Table 1 (row 7) reflects this behavior. Here, P4TE achieves a shorter average FCT for both short and large flows. Both types of flows also experiences reduced

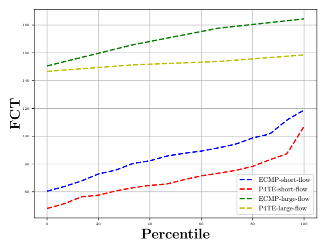


(a) Percentiles of flow number vs FCT

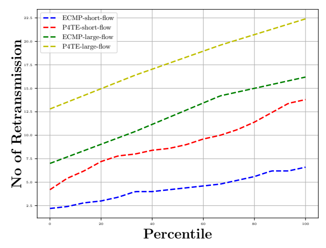


(b) Percentiles of flow number vs retransmissions

Figure 9: Performance results for traffic scenario of § 7.2.2



(a)



(b)

Figure 10: Performance results for traffic scenario of § 7.2.3

dtandard deviation indicating improved fairness among the flows same traffic class.

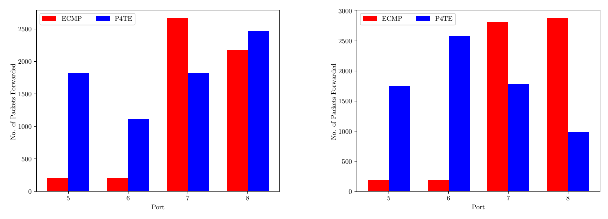
In this scenario, the main bottleneck link is at the last hop of the source-destination path. Hence, P4TE enabled switch imposes rate control at the last leg of the path, while the original data packet reaches the destination at a closer time. As an impact of this phenomenon, the FACK packet sent from the last hop switch and the acknowledgment packet sent by the destination host reaches source of the flow at a closer time (but later than the FACK). It gives the sender a more updated and accurate estimation of the window size. This helps to reduce the number of retransmissions (closer to ECMP) in this case. Table 1 shows that in incast scenario, the average number of retransmissions under P4TE (8.0 for short flows and 14.25 for large flows) and ECMP (4.6875 for short flows and 15.50 for large flows) are very close (compare to the previous 2 scenarios). This reflects the validity of P4TE's rate control algorithm.

7.2.4 Link Utilization

One of the key requirements for TE is to spreading traffic over multiple paths (when available). ECMP often fails to use the path diversity in DCN properly. P4TE's dynamic path priority reconfiguration mechanism, together with the path selection algorithm, helps to move away from static path selection of ECMP. In this scenario, we experimented using a stride pattern (similar to § 7.2.1) to observe P4TE's ability to spread traffic over multiple paths. Here we started a total 48 flows

Table 1

		Avg. for short flows		Avg. for large flows		SD for short flows		SD for large flows	
Traffic Scnearios	Performance Metrics	ECMP	P4TE	ECMP	P4TE	ECMP	P4TE	ECMP	P4TE
Stride pattern low load(§ 7.2.1)	FCT (in sec.)	52.02	30.57	151.26	134.63	30.79	7.81	20.90	5.12
	# of Retransmissions	1.84	10.875	6.5	18.375	1.277	5.31	3.42	11.89
Stride pattern high load(§ 7.2.1)	FCT (in sec.)	47.37	37.53	150.63	136.69	22.43	18.17	26.88	9.74
	# of Retransmissions	5.43	9.65	20.5	31.84	4.96	3.71	0.95	2.21
High Congestion (§ 7.2.2)	FCT (in sec.)	86.88	65.02	156.66	145.26	33.30	23.48	31.88	15.26
	# of Retransmissions	5.375	18.45	4.5	8.3	1.77	2.62	8.51	7.33
Incast (§ 7.2.3)	FCT (in sec.)	89.31	69.59	161.19	139.98	17.57	12.28	11.62	6.88
	# of Retransmissions	4.6875	8.0	15.50	14.25	1.26	2.52	4.71	5.30



(a) Switch-1's Upward Port Packet Counter

(b) Switch-2's Upward Port Packet Counter

Figure 11: Upward link utilization for traffic scenario of § 7.2.4

(short and large) from hosts connected with 2 different leaf switches. We counted the total number of packets forwarded through each of the upward ports (where path diversity exists) of these switches. Fig. 11a and 11b shows, in the case of ECMP packet forwarding algorithm, load on the 4 ports, are highly biased. Here, 2 ports carry the majority of the packets in the upward direction. Whereas P4TE distributes the traffic through all available ports in a balanced manner.

8 Conclusion

This paper presented P4TE, a switch-centric distributed approach for traffic engineering in the fat-tree topology-based data center networks. P4TE leverages programmable switch-based in-band mechanisms to monitor link performance metrics at fine-grained accuracy. P4TE uses fully distributed algorithm for link performance-aware routing and rate control of the flows using monitored metrics. Our evaluations show that P4TE offers superior performance compare to ECMP without any support from the end-host transport layer protocol stack. P4TE is oblivious to the end-host transport layer protocol stack and uses a fake acknowledgment packet as a measure to convey feedback to end hosts. As a result, P4TE faces increased number of retransmission. P4TE achieves improved flow completion time and fairness among flows of the same traffic class. In future, we plan to work on designing algorithms to achieve other traffic engineering objectives (energy awareness, QoS, etc.). We also plan to work on switch-end host protocol stack codesigning for traffic engineering in data center networks.

References

[1] Mohammad Al-Fares, Alexander Loukissas, and Amin Vahdat. A scalable, commodity data center network architecture. *ACM SIGCOMM computer communication review*, 38(4):63–74, 2008.

[2] Mohammad Al-Fares, Sivasankar Radhakrishnan, Barath Raghavan, Nelson Huang, Amin Vahdat, et al. Hedera: dynamic flow scheduling for data center networks. In *Nsdi*, volume 10, pages 89–92, 2010.

[3] Mohammad Alizadeh, Albert Greenberg, David A Maltz, Jitendra Padhye, Parveen Patel, Balaji Prabhakar, Sudipta Sengupta, and Murari Sridharan. Data center tcp (dctcp). In *Proceedings of the ACM SIGCOMM 2010 conference*, pages 63–74, 2010.

[4] Ran Ben-Basat, Xiaoqi Chen, Gil Einziger, and Ori Rottenstreich. Efficient measurement on programmable switches using probabilistic recirculation. In *2018 IEEE 26th International Conference on Network Protocols (ICNP)*, pages 313–323. IEEE, 2018.

[5] Cristian Hernandez Benet, Andreas J Kassler, Theophilus Benson, and Gergely Pongracz. Mp-hula: Multipath transport aware load balancing using programmable data planes. In *Proceedings of the 2018 Morning Workshop on In-Network Computing*, pages 7–13, 2018.

[6] Stephen Bensley, Lars Eggert, Dave Thaler, Praveen Balasubramanian, and Glenn Judd. Datacenter tcp (dctcp):

Tcp congestion control for datacenters. *Internet Draft*, 2017.

[7] Theophilus Benson, Aditya Akella, and David A Maltz. Network traffic characteristics of data centers in the wild. In *Proceedings of the 10th ACM SIGCOMM conference on Internet measurement*, pages 267–280, 2010.

[8] Theophilus Benson, Ashok Anand, Aditya Akella, and Ming Zhang. Understanding data center traffic characteristics. *ACM SIGCOMM Computer Communication Review*, 40(1):92–99, 2010.

[9] Pat Bosshart, Glen Gibb, Hun-Seok Kim, George Varghese, Nick McKeown, Martin Izzard, Fernando Mujica, and Mark Horowitz. Forwarding metamorphosis: Fast programmable match-action processing in hardware for sdn. *ACM SIGCOMM Computer Communication Review*, 43(4):99–110, 2013.

[10] Marco Chiesa, Guy Kindler, and Michael Schapira. Traffic engineering with equal-cost-multipath: An algorithmic perspective. *IEEE/ACM Transactions on Networking*, 25(2):779–792, 2016.

[11] Nandita Dukkipati. *Rcp: Congestion control to make flows complete quickly*. PhD thesis, PhD Thesis, Department of Electrical Engineering, Stanford University, 2006.

[12] Nandita Dukkipati and Nick McKeown. Why flow-completion time is the right metric for congestion control. *ACM SIGCOMM Computer Communication Review*, 36(1):59–62, 2006.

[13] Anja Feldmann, Balakrishnan Chandrasekaran, Seifeddine Fathalli, and Emilia N Weyulu. P4-enabled network-assisted congestion feedback: A case for nacks. In *Proceedings of the 2019 Workshop on Buffer Sizing*, pages 1–7, 2019.

[14] Sally Floyd and Van Jacobson. Random early detection gateways for congestion avoidance. *IEEE/ACM Transactions on networking*, 1(4):397–413, 1993.

[15] Molka Gharbaoui, Barbara Martini, Davide Adami, Gianni Antichi, Stefano Giordano, and Piero Castoldi. On virtualization-aware traffic engineering in openflow data centers networks. In *2014 IEEE Network Operations and Management Symposium (NOMS)*, pages 1–8. IEEE, 2014.

[16] Keqiang He, Eric Rozner, Kanak Agarwal, Wes Felter, John Carter, and Aditya Akella. Presto: Edge-based load balancing for fast datacenter networks. *ACM SIGCOMM Computer Communication Review*, 45(4):465–478, 2015.

[17] Juha Heinanen and Roch Guérin. A two rate three color marker. Technical report, RFc 2698, SepTeMbeR, 1999.

[18] Kuo-Feng Hsu, Ryan Beckett, Ang Chen, Jennifer Rexford, and David Walker. Contra: A programmable system for performance-aware routing. In *17th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 20)*, pages 701–721, 2020.

[19] Kuo-Feng Hsu, Praveen Tammana, Ryan Beckett, Ang Chen, Jennifer Rexford, and David Walker. Adaptive weighted traffic splitting in programmable data planes. In *Proceedings of the Symposium on SDN Research*, pages 103–109, 2020.

[20] Stephen Ibanez, Gianni Antichi, Gordon Brebner, and Nick McKeown. Event-driven packet processing. In *Proceedings of the 18th ACM Workshop on Hot Topics in Networks*, pages 133–140, 2019.

[21] MD Iftakharul Islam and Javed I Khan. Leveraging domino to implement rcp in a stateful programmable pipeline. In *2019 IEEE 20th International Conference on High Performance Switching and Routing (HPSR)*, pages 1–6. IEEE, 2019.

[22] Srikanth Kandula, Sudipta Sengupta, Albert Greenberg, Parveen Patel, and Ronnie Chaiken. The nature of data center traffic: measurements & analysis. In *Proceedings of the 9th ACM SIGCOMM conference on Internet measurement*, pages 202–208, 2009.

[23] Dina Katabi, Mark Handley, and Charlie Rohrs. Congestion control for high bandwidth-delay product networks. In *Proceedings of the 2002 conference on Applications, technologies, architectures, and protocols for computer communications*, pages 89–102, 2002.

[24] Naga Katta, Mukesh Hira, Changhoon Kim, Anirudh Sivaraman, and Jennifer Rexford. Hula: Scalable load balancing using programmable data planes. In *Proceedings of the Symposium on SDN Research*, pages 1–12, 2016.

[25] Mirza Mohd Shahriar Maswood, Chris Develder, Edmundo Madeira, and Deep Medhi. Energy-efficient dynamic virtual network traffic engineering for north-south traffic in multi-location data center networks. *Computer Networks*, 125:90–102, 2017.

[26] Alaitz Mendiola, Jasone Astorga, Eduardo Jacob, and Marivi Higuero. A survey on the contributions of software-defined networking to traffic engineering. *IEEE Communications Surveys & Tutorials*, 19(2):918–953, 2016.

[27] Mathijs Mortimer. iperf3 documentation, 2018.

[28] Mohammad Noormohammadpour and Cauligi S Raghavendra. Datacenter traffic control: Understanding techniques and tradeoffs. *IEEE Communications Surveys & Tutorials*, 20(2):1492–1525, 2017.

[29] Marcelo Pizzutti and Alberto E Schaeffer-Filho. Adaptive multipath routing based on hybrid data and control plane operation. In *IEEE INFOCOM 2019-IEEE Conference on Computer Communications*, pages 730–738. IEEE, 2019.

[30] Arjun Roy, Hongyi Zeng, Jasmeet Bagga, George Porter, and Alex C Snoeren. Inside the social network's (datacenter) network. In *Proceedings of the 2015 ACM Conference on Special Interest Group on Data Communication*, pages 123–137, 2015.

[31] Belma Turkovic, Fernando Kuipers, Niels van Adrichem, and Koen Langendoen. Fast network congestion detection and avoidance using p4. In *Proceedings of the 2018 Workshop on Networking for Emerging Applications and Technologies*, pages 45–51, 2018.

[32] Nedeljko Vasić and Dejan Kostić. Energy-aware traffic engineering. In *Proceedings of the 1st International Conference on Energy-Efficient Computing and Networking*, pages 169–178, 2010.

[33] Xin Wu and Xiaowei Yang. Dard: Distributed adaptive routing for datacenter networks. In *2012 IEEE 32nd International Conference on Distributed Computing Systems*, pages 32–41. IEEE, 2012.

[34] Jin-Li Ye, Chien Chen, and Yu Huang Chu. A weighted ecmp load balancing scheme for data centers using p4 switches. In *2018 IEEE 7th International Conference on Cloud Networking (CloudNet)*, pages 1–4. IEEE, 2018.

[35] Jianchuan Zhang, Zhenjie Yao, Yanhui Tu, and Yixin Chen. A survey of tcp congestion control algorithm. In *2020 IEEE 5th International Conference on Signal and Image Processing (ICSIP)*, pages 828–832. IEEE, 2020.

A Performance Evaluation

In §7 we explained P4TE's performance gain on several important scenario. Here, we discuss some other traffic patterns and P4TE's performance in those scenarios. In section §7, all the tests were executed with the following configuration: host to link facing links of leaf switches with 40 packets per second processing rate, all flows were capped to 16Kbps rate, all short flows were started with 16KB TCP buffer size, and all large flows were started with 24 KB TCP buffer window size. The reason behind this is while simulating the DCN we need to maintain an oversubscription ratio of 2:1. To do that, we configured the host to leaf switch facing port with 40 pps and leaf to spine switch with 20pps processing rate. Now, If we start a flow with 50KB volume to send and 65KB TCP buffer (typical linux TCP socket buffer size) IPerf3 will send the whole 50 KB volume of data at once. As this is a simulation scenario and all flows share the same set of CPU, other flows will get the chance to send their packets later. Clearly, this is not a reflection of real-life case. Consider a 6.4 Tbps Tofino ASIC leaf switch with 100 Gbps link speed. In this case, a flow will send a maximum 100Gbps speed, much lower than the total capacity in upward ports. So starting a flow with a TCP buffer size larger than the link's processing rate is not a realistic scenario. Alternatively, we can increase the link's processing rate. However, Bmv2 has limited scalability, hence increasing the link processing rate affects its performance. In such cases, simulation result varies significantly. As another solution, we can add packet pacing to IPerf3 flows, but adding packet pacing for many flows needs many computations. In a simulation environment, all flows share the same CPU set; hence, adding packet pacing needs more computation and more CPU context switch. Like the previous scenario, here simulation performance will also degrade.

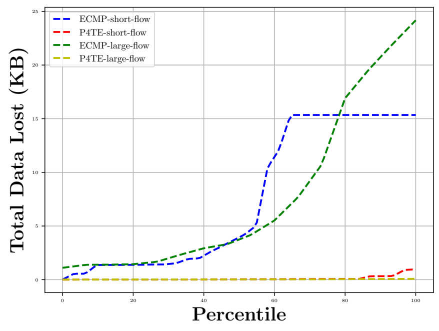
To test P4TE's performance with a higher flow rate and win- dow size, we added 2 more sets of tests, and they are discussed here.

A.1 Stride Pattern (48Kbps flow Rate and 65KB TCP Buffer)

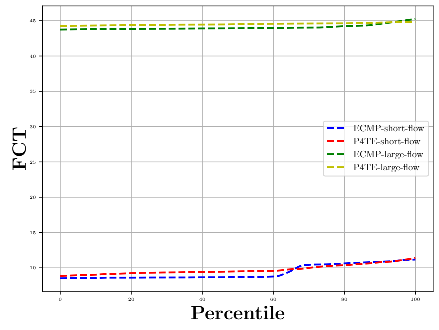
Here we started 32 short flows and 16 large flows using stride pattern. Each of the short flows was configured to transfer 50KB, and large flows were configured to transfer 256KB of data. Each flow was configured to start with 65KB TCP buffer size. In this case, under P4TE all flows faced almost zero data loss (un-acknowledged + lost) with a slight increase in FCT. On the other hand, ECMP sends a whole flow volume of data for short flows at once. Once all data of a short flow were transmitted IPerf3 closed the TCP windows, and none of the packets were acknowledged (which is a clear form 0 retransmissions for all short flows and 0 retransmissions for 90% of the large flows) (Fig. 12c). As packets were sent at such high speed, they may get dropped in the path. On the other hand, in P4TE both short and large flows almost no data. The full volume of flows was acknowledged. Hence P4TE faces a slight increase in FCT (Fig. 12b). Though this is not a realistic scenario; still, P4TE performed well.

A.2 Stride Pattern (48Kbps flow Rate and 65KB TCP Buffer with Large flow Volume)

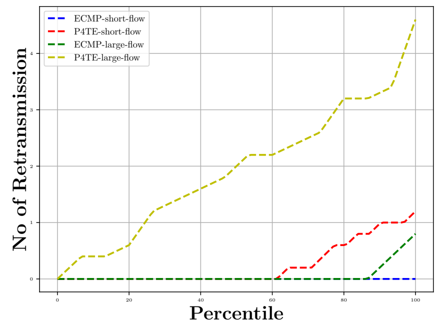
Though short flow should be 50KB in size, but starting large number of such flows to saturate the DCN in simulation environment is hard. Because Iperf closes the window immedi-



(a) Percentiles of flow number vs Data Loss



(b) Percentiles of flow number vs FCT



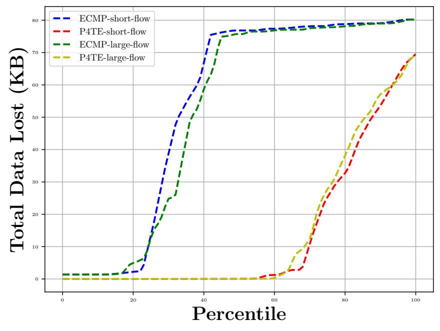
(c) Percentiles of flow number vs no. of retranmissions

Figure 12: Performance results for traffic scenario of § A.1

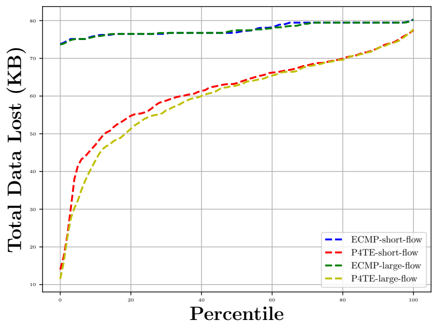
ately after transmitting the entired flow volume of data. To the end we increased the short flow size and large flow size to simulate the classful behavior of P4TE and test performance. Here we started 32 short flows and 16 large flows using stride pattern. Each of the short flows was configured to transfer 1MB and large flows were configured to transfer 3MB of data. Each flow was configured to start with 65KB TCP buffer size. Fig. 13b shows, in this case, P4TE performs better compared to ECMP. Here, most of the short flows (more than 50% of total short flows) and large flows (more than 70% of total large flows) face increased FCT under ECMP. As the TCP buffer size was large and IPerf flows do not waited for the data t be acknowledged, none of the flows faced any retransmissions (fig. 13c). On the other hand, due to the rate control scheme, flows faced increased retransmissions under P4TE. Also, in this scenario, P4TE achieved improved shorter FCT compared to ECMP.

A.3 Stride Pattern (Another Variation))

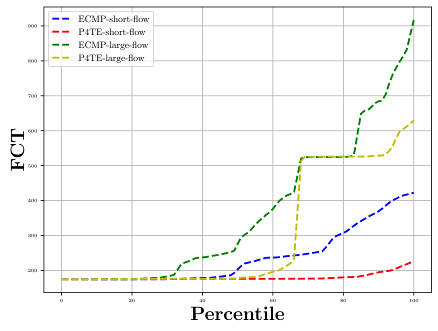
At one end, generating flow with volume less than TCP window size is hard due to Iperf3 behavior. On the other hand, generating such flow in large numbers is another problem due to limited CPU power. TCP Pacing is also problematic for the same reason. To this end, we changed the size of short and large flows to 256K and 1024K, respectively. For both the flow types, we set the flow rate at 128 Kbps. Here, we started 3 sets of short flow and 1 set of large flow using stride pattern. This leads to at least 4 flows from hosts under the same leaf switch started together. Then while large flows are continuing, we repeated the short flows 3 times to create the same traffic load level on each switch. These flows together can create around 65packets per second traffic load on each leaf switch. This load is more than the capacity of upward ports of each leaf switch. Fig. 14a, 14b and 14c shows the performance of ECMP and P4TE in this case. The performance comparisons show that P4TE achieves performance gain in FCT at the cost of more retransmissions like all other traffic scenarios.



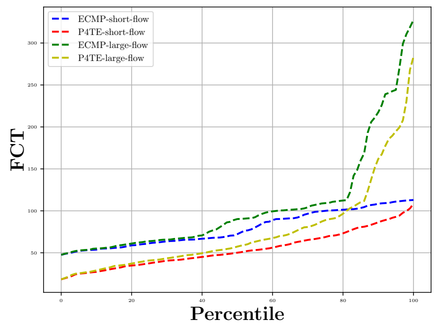
(a) Percentiles of flow number vs Data Loss



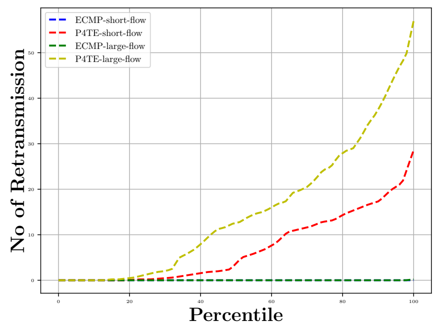
(a) Percentiles of flow number vs Data Loss



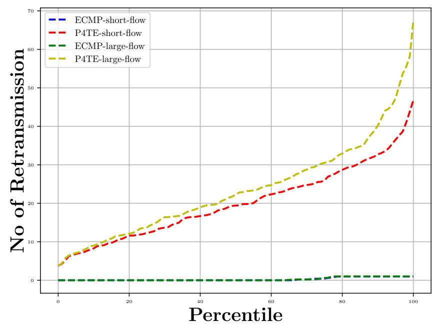
(b) Percentiles of flow number vs FCT



(b) Percentiles of flow number vs FCT



(c) Percentiles of flow number vs no. of retranmissions



(c) Percentiles of flow number vs no. of retranmission

Figure 13: Performance results for traffic scenario of § A.2

Figure 14: Performance results for traffic scenario of §A.3