TinyFlow: Breaking Elephants Down Into Mice in Data Center Networks

Hong Xu*, Baochun Li†
henry.xu@cityu.edu.hk, bli@ece.toronto.edu
* Department of Computer Science, City University of Hong Kong
† Department of Electrical and Computer Engineering, University of Toronto

***Abstract*—Current multipath routing solution in data centers relies on ECMP to distribute traffic among all equal-cost paths. It is well known that ECMP suffers from two deficiencies. ECMP does not differentiate between elephant and mice flows, creates head-of-line blocking for mice flows in the egress port buffer, and results in long tail latency. Further it does not fully utilize available bandwidth due to hash collision among elephant flows. We propose TinyFlow, a simple yet effective approach that remedies both problems. TinyFlow changes the traffic characteristics of data center networks to be amenable to ECMP by breaking elephants into mice. In a network with a large number of mice flows only, ECMP naturally balances load and performance is improved. We conduct NS-3 simulations and show that TinyFlow provides 20%–40% speedup in both mean and 99-th percentile FCT for mice, and about 40% throughput improvement for elephants.**

I. Introduction

Modern data center networks are constructed with Clos topologies such as fat-tree [5] and VL2 [16]. To achieve full bisection bandwidth in a scalable, cost-effective manner, these topologies provide abundant path diversity with many equal-cost paths between a given pair of hosts. Hash-based multipath routing, e.g. ECMP (Equal Cost Multipathing), is widely used to choose a path among all shortest paths based on the hash value of a flow's five-tuples. ECMP is simple to implement and does not require per-flow information at switches.

Yet, ECMP suffers from two major drawbacks when used in data center networks [6], [11], [14], [25]. First, ECMP does not differentiate between mice flows—flows that are latency-sensitive short requests/responses for interactive applications—and elephant flows—flows that transfer bulky traffic for parallel computation jobs [7], [25]. Thus, mice flows often find themselves queued behind elephants in the egress port, suffering from long queueing delays and poor flow completion times (FCT), especially in the tail [7], [25]. Second, ECMP cannot effectively utilize available bandwidth due to hash collision [6], [11], [14]. When the hash values of two (or more) elephants collide, they have to share the same link with reduced throughput even though other paths may be completely free.

The networking community has devoted much attention to ECMP's problems recently. In general there are two approaches to improve ECMP. The first is to identify and route elephants to uncongested paths based on global information [6], [10], [13], [20]. This approach requires centralized optimization algorithms and poses scalability challenges for current OpenFlow switches [13]. More importantly, it does not address the tail latency problem, since elephant routing is performed at the granularity of seconds, due to the overhead of statistics collection and the complexity of optimization. The second, and newer, approach is to use per-packet rather than per-flow routing, where each packet is forwarded to a distinct port among equal-cost paths in a randomized [14] or deterministic [11] way. Per-packet routing clearly leads to better load balancing and improved tail latency, though its performance in asymmetric topologies due to link failures and topology updates is unclear at best, especially for mice flows with packet reordering.

We present a new approach to designing multipath routing systems that provide high utilization, low latency, and short latency tail, called TinyFlow. TinyFlow is based on a simple observation: ECMP is designed to balance load evenly when there are many flows of similar sizes [14]. Thus, we can break elephants down into many mice flows, and distribute them uniformly at random over all paths. Intuitively, TinyFlow represents a per-mice routing approach, and automatically changes the traffic characteristics of the network to be amenable to ECMP. During the lifetime of an elephant, its packets are spread across the network instead of pinned to a pre-determined path, leading to better bandwidth utilization and tail latency. On the other hand, reordering now only happens across mice flows of an elephant flow. The negative impact on tail latency is reduced as each "real" mice flow is always kept on the same path. Note that reordering does not affect throughput of long-lived elephants with TCP SACK and packets traversing equal-cost paths whose loads are well balanced. This is verified by both testbed experiments in previous work [14] and our simulations.

TinyFlow can be implemented in many different ways to transform an elephant into mice. The current TinyFlow design, generally speaking, works on each edge switch of the network to record the active elephants together with their egress ports and byte counts. An elephant's egress port is varied randomly every 10KB, effectively breaking it down into a *sequence*[1]

[1]These mice flows are transmitted sequentially, not concurrently.

978-1-4799-4894-9/14/$31.00 ©2014 IEEE

of mice flows each with 10KB. This design does not require global coordination in the control plane or application/end-host modification, and can be easily implemented over any OpenFlow switches without the complexity of periodically running any centralized algorithm. An elephant (say $>$1MB) is equivalent to a large number of mice flows ($<$100KB). Thus in TinyFlow, when an elephant is split into mice, they are transmitted sequentially rather than concurrently to avoid incast [22], and is different from MPTCP [19].

We evaluate TinyFlow with packet-level simulations in NS-3. Our simulations use a data center traffic trace that mimics a web search workload [7] with a 16-pod 1,024-host fat-tree. It is shown that TinyFlow achieves $\sim$18% and $\sim$40% speedup in both mean and 99-th percentile FCT for mice flows, and $\sim$40% in mean and tail FCT for elephants compared to ECMP. These preliminary results confirm that TinyFlow is a lightweight and effective multipath routing system.

The remainder of this paper is organized as follows. Sec. II summarizes related work. Sec. III discussses the motivation, design, and alternative implementations of TinyFlow. Sec. IV presents trace-driven NS-3 simulations results, and finally Sec. V concludes the paper.

II. Related Work

Motivated by the drawbacks of ECMP, many new multipath routing designs have been proposed. We briefly review the most relevant prior work here, including both per-flow and per-packet approaches. We also cover other related solutions working on other layers of the data center network to improve FCT.

Per-flow multipath. Most work here adopts a centralized approach enabled by software defined networks (SDN) [15]. Hedera [6] is the first that identifies ECMP's hash collision problem in data centers. Al-Fares *et al.* propose a dynamic routing system that detects elephants and finds a good end-to-end path with optimization algorithms, and show that aggregated throughput can be improved by more than 100%. MicroTE [10] leverages the short-term traffic predictability and develops a fine-grained routing system using linear programming. Both systems are OpenFlow based, and require global information and decision-making. TinyFlow shares the philosophy to strategically route elephants, but does so in a purely local and distributed manner by randomly varying the egress ports and transforming them into mice flows. LocalFlow [20] is a switch-local solution that routes elephants based on multi-commodity flow algorithms. Yet, similar to other work, it is not designed to reduce the tail latency of mice flows.

Some other work focuses on specific aspects of the system design. Mahout [12] improves the scalability of elephant detection with a host-based mechanism that monitors the host socket buffers to detect elephants. DevoFlow [13] proposes a modified OpenFlow model to improve the scalability of the controller for centralized flow management. These proposals are complementary to TinyFlow.

Per-packet multipath. Another thread of work departs from the traditional per-flow paradigm, and advocates per-packet multipath routing instead [11], [14]. Per-packet routing significantly improves load balancing. However it introduces packet reordering which interacts negatively with TCP, because TCP interprets out-of-sequence packets as a congestion signal and unnecessarily reduces the window size. Dixit *et al.* [14] examine the impact of packet reordering with random packet spraying in a fat-tree network using testbed implementations. They find that because of the symmetry of the topology, and modern TCP's DupACK threshold adjustment, DSACK option, and other built-in mechanisms, packet reordering can be tolerated quite well and does not affect throughput of long-lived elephants. As part of the system, DeTail [25] includes an adaptive network layer that forwards a packet to an egress port whose queue occupancy is below a certain threshold. Cao *et al.* [11] focus on latency, and design a deterministic per-packet round-robin routing system that focuses on reducing the queue length.

Despite the progress, it is still unclear how per-packet multipath routing affects the tail latency of mice flows, especially in asymmetric topologies which occur frequently in reality due to temporary failures, maintenance, etc. Mice flows are very sensitive to reordering and a single event of retransmission can cause the flow to hit the tail. When some links are down, congestion levels can differ significantly on equal-cost paths, leading to long latency tail as well as reduced throughput [14].

Beyond the network layer. At the transport layer, MPTCP [19] is a multipath TCP that splits a flow into a few subflows, each routed by ECMP through different paths concurrently in order to increase throughput and robustness. DCTCP [7] and HULL [8] use fine-grained congestion control and phantom queues to reduce the queue length and improve latency. The latency improvement comes at a small cost of throughput though. Other work, including D^3 [23], PDQ [17], and pFabric [9], proposes different queueing and scheduling disciplines to prioritize mice flows for better latency. On the same ground, our previous work RepFlow [24] works at the application layer by replicating each mice flow to reap multipath diversity and achieve better latency on average and at the tail. All these proposals work beyond the network layer, and they can be used together with an efficient multipath routing system, such as TinyFlow, to improve performance in a data center network.

III. Motivation and Design

We present the motivation and design of TinyFlow in this section.

A. Motivation

Production data center networks use symmetric topologies such as Clos-based fat-tree or VL2. For such networks, ECMP is designed to balance load perfectly when there are many flows with similar sizes. This is intuitive to understand: hashing a large number of flows leads to a uniform distribution of hash values, and a uniform distribution of traffic over the paths given flow sizes are similar. However it is widely known that data center traffic violates this assumption [7], [25], and causes various performance problems.

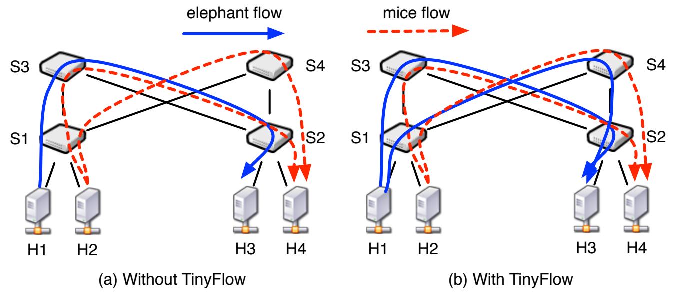


Fig. 1: TinyFlow alleviates head-of-line blocking for mice flows and improves latency.

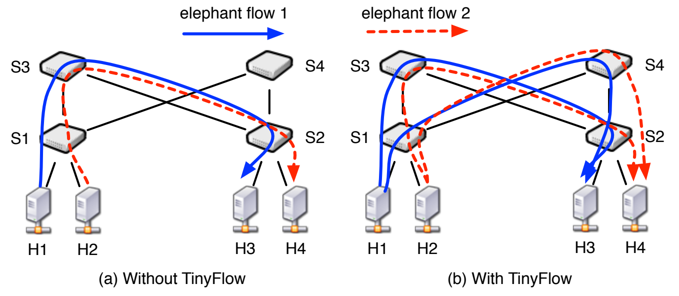


Fig. 2: TinyFlow improves throughput of elephant flows as well as bandwidth utilization.

Researchers have focused on redesigning ECMP to better cope with data center traffic as mentioned already. Another approach, working in the opposite direction, is to "redesign" the traffic and transform its characteristics to be amenable to ECMP's original design, which has not been discussed thus far. This motivates TinyFlow, whose objective is to break elephants down into a stream of many mice.

TinyFlow brings two benefits. First, it alleviates head-of-line blocking due to elephants hogging up the egress port buffer in switches, and improves FCT for mice flows. To understand this, consider the following toy example shown in Fig. 1. Say H1 sends an elephant flow to H3, which takes the path S1-S3-S2 as shown in Fig. 1(a). H2 sends a stream of mice flows to H4. By ECMP, these mice flows have equal probability to traverse the two paths S1-S3-S2 and S1-S4-S2, and roughly half of them will experience head-of-line blocking because of the colocating elephant. With TinyFlow, as shown in Fig. 1(b), the elephants are broken into many mice flows, evenly spread across both paths. The uniform load distribution significantly reduces tail latency of mice flows.

For the purpose of illustration, assume the link capacity is μ bps, the elephant flow has a throughput of λ bps, and the traffic from mice flows is negligible. This is a reasonable simplification as mice flows account for less than 5% of total bytes based on previous measurements [7], [16], [25]. Further assume one path can be modeled as a M/D/1 queue, whose expected queue length as a function of the load $\rho = \lambda/\mu$ is $Q(\rho) = \frac{(2-\rho)\rho}{2(1-\rho)}$. Then the average queue length seen by mice flows is $0.5Q(\rho)$, and half of them see an expected queue length of $Q(\rho)$ without TinyFlow. With TinyFlow, both paths are identical and the arrival rate becomes 0.5λ. The average queue length is $Q(0.5\rho)$, which is less than $0.5Q(\rho)$ due to concavity of $Q(\cdot)$. This holds for many queueing models as long as the expected queue length is a concave function of load. Therefore, TinyFlow improves both average and tail latency by evenly distributing traffic from elephants.

The second benefit of TinyFlow is that it alleviates the ill effect of hash collision among elephants and improve their throughput. This can also be illustrated by a toy example in Fig. 2. The setup is similar to Fig. 1, where H1 sends an elephant flow to H3 via S1-S3-S2. Now H2 needs to send another elephant to H4. When a hash collision happens between the two elephants, they obtain a throughput of 500 Mbps only if the network is 1 Gbps. The bandwidth utilization is only 50% as the other path is completed wasted. With TinyFlow, both elephants are evenly spread out, each with a throughput of 1 Gbps which represents a 100% bandwidth utilization.

B. Design

On the high level, TinyFlow's design can be summarized in one sentence: it identifies and routes elephant flows as many independent mice flows, by dynamically varying the egress port of a detected elephant. TinyFlow only needs to run at the edge switches of a Clos network; the rest of the network uses unmodified ECMP. This is because each host is connected to a unique edge switch, and varying the egress port at the edge switch leads to a unique non-overlapping path to the upstream. TinyFlow is comprised of two components: elephant detection and dynamic random rerouting. We present the detailed design in the following.

Elephant detection can be done in several ways: maintaining and polling per-flow statistics, packet sampling, and end-host based monitoring. Per-flow statistics as used in Hedera [6] has high accuracy at the cost of poor scalability in commodity switches. End-host based monitoring such as Mahout [12] overcomes the scalability issue, though it has not been widely adopted. In our current design we use packet sampling, since it is widely used in practice with mature switch support such as sFlow [3]. This also allows us to understand the practical benefits of the system. In TinyFlow edge switches sample packets on the downstream ports connected to hosts and keep flow records for a small time period. If a flow's occurrence exceeds a pre-defined threshold it is detected as an elephant. TinyFlow is not bound to a particular elephant detection method, and certainly stands to gain with future improvements in detection accuracy.

At the heart of TinyFlow is the dynamic random rerouting mechanism based on OpenFlow, which effectively transforms an elephant into a stream of mice and works together with ECMP. Specifically, all flows are routed using ECMP by default. Once an elephant is detected, the edge switch adds a new entry for it to the flow table and monitors its byte count. When the byte count exceeds a threshold, the switch chooses a different egress port out of the equal-cost paths uniformly at random for the elephant, modifies the flow table entry, and resets the byte count. This design is distributed where each edge switch makes independent rerouting decisions. Thus the controller does not need to be invoked at all, and the overhead of aggregating information and running centralized optimization is eliminated.

Some aspects of the dynamic rerouting design bear further discussions. First, the system can be potentially improved by using end-to-end congestion information to aid path selection. Such information can be carried over feedback messages between switches, such as the congestion notification packets defined in IEEE 802.1Qau [1]. Indeed some work has explored this direction with promising results [21]. Second, TinyFlow may also be improved with just local congestion information, i.e. port buffer occupancy. This represents another dimension of the design space with little previous work [25]. We believe, as verified by large-scale simulations also, that TinyFlow's simple design balances the load of the entire network and to a large degree remedies ECMP's problems. We do not explore these directions in this preliminary study, with the hope that they will be addressed in follow-up studies.

C. Alternative Implementations

TinyFlow lends itself to many implementation choices. Here we discuss some alternative designs briefly.

It is possible to implement TinyFlow purely at the application layer, by having the application transmit an elephant as a stream of mice flows. To ensure the resulting mice are transmitted independently by ECMP, their five-tuples (most likely port numbers) must be modified. This approach does not require switch modifications. Yet it does require changes to existing application code, and it may not be extensible to implement adaptive rerouting as it lacks direct control on path selection.

It is also possible to have a different switch implementation of TinyFlow. For example one may instruct the OpenFlow switch to directly modify the five-tuples of a detected elephant, so that ECMP will hash it to a different egress port. This approach still lacks direct control on path selection. Further it imposes additional per-packet processing overhead which may degrade the latency performance.

Compared to these alternatives, our current design strikes a good balance between practicality and performance. TinyFlow only requires primitive OpenFlow operations, such as byte count and modifying egress port, that has been supported since the inception of OpenFlow standard [4]. It also allows direct control over path selection with extensibility, and existing applications can directly run on top of it. The tradeoff we make is that packet sampling is not as accurate as an application layer implementation that perfectly identifies elephants.

IV. Experimental Evaluation

We now evaluate TinyFlow using packet-level simulations in the NS-3 simulator.

A. Methodology

Topology: We use a 16-pod fat-tree as the network topology [5], which is commonly used in data centers. The fabric consists of 16 pods, each containing an edge layer and an aggregation layer with 8 switches each. Each edge switch connects to 8 hosts. The network has 1,024 hosts and 64 core switches. There are 64 equal-cost paths between any pair of hosts at different pods. Each switch is a 16-port 1Gbps switch, and the network has full bisection bandwidth. The end-to-end round-trip time is $\sim 32\mu$s.

Benchmark workloads: We use empirical workloads to reflect traffic patterns in production data centers. We consider a flow size distribution from a data center mostly running web search [7], which is heavy-tailed with a mix of mice and elephants. In this workload, over 95% of the bytes are from 30% of the flows larger than 1MB. Flows are generated between random pairs of hosts following a Poisson process with load varying from 0.1 to 0.8 to thoroughly evaluate TinyFlow's performance in different traffic conditions. We simulate 0.5s worth of traffic at each run, and ten runs for each load. The entire simulation takes more than 900 machine-hours.

B. Schemes Evaluated

ECMP: The default multipath routing scheme. ECMP hashes the five-tuples, i.e. src and dst IP, src and dst port number, and protocol number.

TinyFlow: Our design as explained in Sec. III. We implement TinyFlow switches as a new `point-to-point-net-device` in NS-3, and adds dynamic rerouting support at the network layer. Packet sampling is done every 100 KB, an elephant is detected once two samples are found in 500 μs, and a detected elephant is rerouted every 10 KB.

For both ECMP and TinyFlow, TCP NewReno is used without any tuning as the transport protocol. Our code is based on NS-3.10 which contains a full implementation of TCP NewReno as of RFC 2582 [2], without SACK. Note SACK may further improve throughput with the presence of packet reordering. Table I summarizes some key parameters.

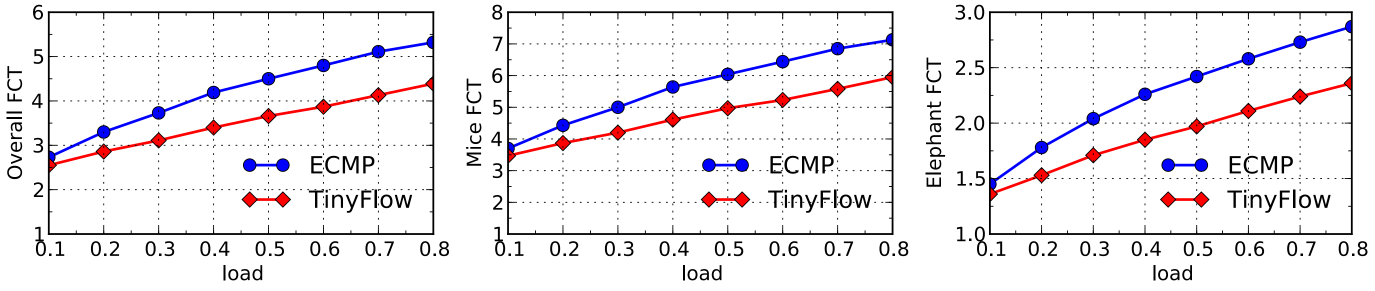
Parameter	Value
Fat-tree topology	$k = 16$
Link speed	1 Gbps
Switch queue type	DropTail
Buffer size per port	25 packets
Packet size	1500 B
Sampling interval	100 KB
Elephant detection threshold	2 samples in 500 μs
Reroute interval	10 KB

TABLE I: Key parameters in our simulation.

We use (normalized) flow completion time, FCT in short, as the performance metric throughout the evaluation. FCT is defined as the flow's completion time normalized by its best possible completion time without contention.

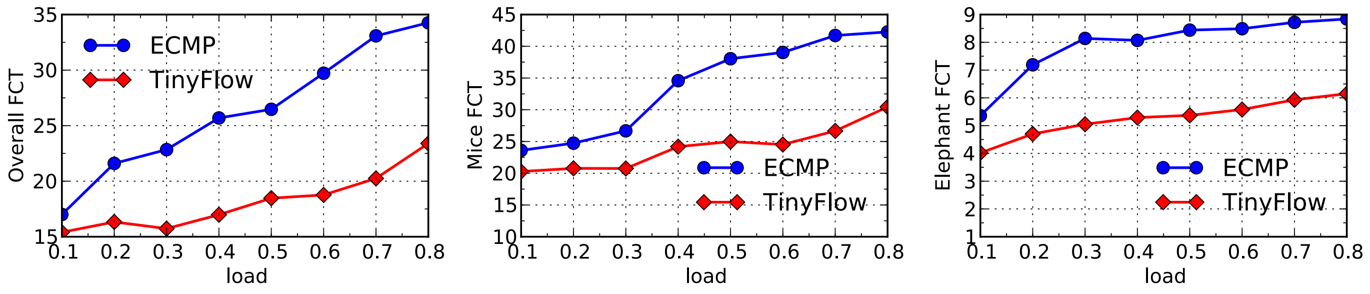
C. Overall Performance

We first look at the overall FCT performance. From Fig. 3a, TinyFlow reduces the mean FCT for all flows by up to 25% compared to ECMP. When the load is low (0.1 or 0.2), the



(a) All flows. (b) Mice flows $(0, 100]$ KB. (c) Elephant flows $(100, \infty)$ KB.

Fig. 3: Mean FCT in a 16-pod fat-tree with the web search workload [7].



(a) All flows. (b) Mice flows $(0, 100]$ KB. (c) Elephant flows $(100, \infty)$ KB.

Fig. 4: 99-th percentile FCT in a 16-pod fat-tree with the web search workload [7].

improvement is small as most of the paths are uncongested and ECMP functions well most of the time. TinyFlow's improvement is more salient in terms of tail FCT. As shown in Fig. 4a, it improves the 99-th percentile FCT by around 40% consistently in almost all loads.

The results demonstrate the advantage of breaking elephants down into mice in load balancing the network, especially when the load is mild to high (>0.3). The FCT reduction is more substantial in relatively high loads because with a mix of elephants and mice ECMP results in severe unbalance of the load and high FCT.

D. *Performance of Mice Flows*

Next we focus on mice flows—flows that are less than 100 KB. Fig. 3b and Fig. 4b show the mean and 99-th percentile FCT for mice flows, respectively. The first observation is that mice flows in general has slower FCT: both the mean and tail FCT are longer than that of all flows. This is in line with the recent trend in our community that focuses on improving FCT of mice flows [18]. One can also observe that the discrepancy between mean and tail FCT improvements of TinyFlow is more salient now. Mice flows' mean FCT reduction is only 14%–18%, while the tail FCT reduction is 30%–38%. Mice flows' FCT depends largely on queueing delay in the path. Thus, the long tail FCT evidently shows that mice flows suffer more from head-of-line blocking due to elephants that arrive from time to time. TinyFlow can effectively improve the tail FCT and application level performance.

E. *Performance of Elephant Flows*

Here we look at the performance of elephant flows, whose size is larger than 100 KB. Their FCT depends on throughput and therefore congestion with other co-locating elephants, instead of queueing delay which could vary significantly. Thus one can clearly see that their performance, especially the 99-th percentile FCT in Fig. 4c, is much better than mice flows in general. Nevertheless, TinyFlow still improves the mean FCT by around 25%, and tail FCT by around 40%. This confirms that TinyFlow can alleviate throughput degradation caused by hash collisions among elephants.

To better understand the effect of packet reordering, Table II shows the amount of additional traffic caused by TCP retransmissions as a result of out-of-order arrivals. We can see that there is a fair amount of retransmitted packets. Note that the results are obtained with un-tuned TCP. If we adjust the DupACK threshold for retransmission and enable SACK (which is not implemented in NS-3), this overhead can be reduced and performance of both elephants and mice can be further improved.

0.1	0.2	0.3	0.4	0.5	0.6	0.7	0.8
9.19%	12.59%	14.06%	14.49%	14.53%	17.08%	16.48%	16.61%

TABLE II: Retransmission overhead in TinyFlow with packet reordering.

To summarize, TinyFlow achieves much better FCT for all flows in data center networks. Our performance evaluation serves as a conservative estimate of its benefits. There is room for improvements especially in elephant detection accuracy, adaptive rerouting, and handling packet reordering.

V. Concluding Remarks

We presented the design and evaluation of TinyFlow, a simple approach that breaks elephant flows down into mice flows to better load balance the data center network. It is an effective multipath routing system that can be readily implemented on OpenFlow switches to support existing applications and transport layer protocols. TinyFlow represents a new approach that aims to redesign the traffic characteristics to better suit the protocols rather than redesigning the protocols to work with the traffic. We believe such a thinking applies beyond the network layer and potentially can lead to promising new ideas. Our work is only the first step in this direction. The design space needs more thorough exploration. Future work is needed to improve various aspects of the system, some of which are already identified, and better understand the benefits of TinyFlow.

References

[1] IEEE 802.1Qau — Congestion Notification. http://www.ieee802.org/1/pages/802.1au.html.

[2] The NewReno Modification to TCP's Fast Recovery Algorithm RFC 2582. http://datatracker.ietf.org/doc/rfc2582/.

[3] sFlow.org. http://www.sflow.org/.

[4] OpenFlow switch specification v.1.3.3. https://www.opennetworking.org/, September 27 2013.

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

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

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

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

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

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

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

[12] A. R. Curtis, W. Kim, and P. Yalagandula. Mahout: Low-overhead datacenter traffic management using end-host-based elephant detection. In *Proc. IEEE INFOCOM*, 2011.

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

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

[15] N. Feamster, J. Rexford, and E. Zegura. The road to SDN: An intellectual history of programmable networks. *ACM Queue*, 11(12):20:20–20:40, December 2013.

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

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

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

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

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

[21] A. S.-W. Tam, K. Xi, and H. J. Chao. Leveraging performance of multiroot data center networks by reactive reroute. In *Proc. IEEE Symposium on High Performance Interconnects*, 2010.

[22] V. Vasudevan, A. Phanishayee, H. Shah, E. Krevat, D. G. Andersen, G. R. Ganger, G. A. Gibson, and B. Mueller. Safe and effective fine-grained TCP retransmissions for datacenter communication. In *Proc. ACM SIGCOMM*, 2009.

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

[24] H. Xu and B. Li. RepFlow: Minimizing flow completion times with replicated flows in data centers. In *Proc. IEEE INFOCOM*, 2014.

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