flow completion time is more than 3.3 with ECMP, while with FlowBender the ratio reduces to less than 1.3 implying a tighter latency distribution with lower variance.

4.2.3 All-to-All Workloads

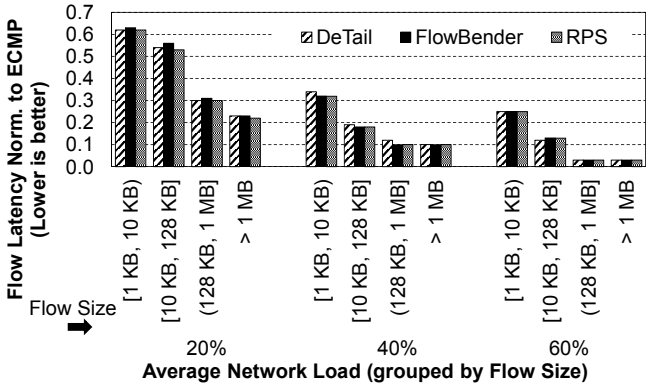

Figure 3: All-to-All workloads: Mean Latency Norm. to ECMP

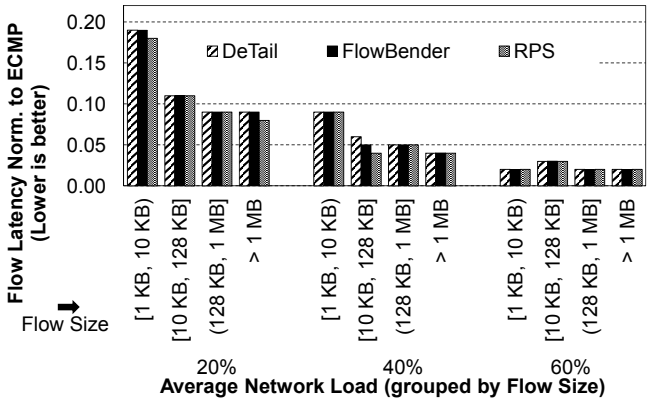

Figure 4: All-to-All workloads: Tail Latency Norm. to ECMP

We compare ECMP, RPS, DeTail, and FlowBender with traffic patterns typical of large-scale online services such as Web search. Our flow size distribution is heavy-tailed and is modeled based on the data from [8]. Every server randomly picks another server in the network to send data to, and flows arrive at the sender as a poisson process with the mean adjusted to produce the desired load. We vary the load (reported relative to the bisectional bandwidth), and compare the different schemes in terms of the mean and the $99^{th} percentile$ latency.

We show the means in Figure 3 and the 99^{th} percentile latencies in Figure 4, with each being binned across flow sizes: (a) less than or equal to 10KB, (b) greater than 10KB but less than or equal to 128KB, (c) greater than 128KB but less than or equal to 1MB, and (d) greater than 1MB. The load is plotted along the X-axis, and we plot the latency of RPS, DeTail, and FlowBender normalized to that of ECMP along the Y-axis. Across all of these loads, we see that all the three schemes i.e., DeTail, FlowBender and RPS *substantially* outperform ECMP. The high level takeaway from our experiments here is that though DeTail, RPS, and FlowBender achieve similar performance in general, FlowBender does not require *any* hardware change at the switches *and* is not strictly predesigned to operate in symmetric topologies, unlike DeTail and RPS.

4.2.4 Out-of-order Packet Delivery

FlowBender incurs *negligble* out-of-order packets relative to what DeTail and RPS incur. In fact, we monitored the number of out-of-order packets in all simulations and found that with FlowBender, the probability of a packet arriving out of order was about 0.006% higher than ECMP's. Compared to FlowBender's performance, DeTail had more than 97.9% the number of out-of-order packet delivery that RPS experienced.

Furthermore, DeTail suffers from the Tree Saturation problem due to its reliance on link-level PFC signals [11, 20, 12], and RPS cannot cope with asymmetric topologies and link failures [23].

4.2.5 Partition Aggregate Workloads

In this experiment, we evaluate FlowBender's ability in reducing tail latency under synchronized Partition-Aggregate traffic patterns typical of datacenter applications like Web search. We use the same heavy-tailed flow size distribution here that is used in the previous all-to-all experiment. We initiate Partition-Aggregate jobs as a Poisson arrival process with mean such that the total utilization of the bisectional bandwidth is equal to 40%. Each Partition-Aggregate job corresponds to a 1MB transaction broken evenly across n workers that are spread randomly in the fabric with all workers responding simultaneously together with their data but essentially finishing at different times. In Figure 5, we vary n between 4 and 32 along the X-axis and plot the average job completion time normalized to ECMP's i.e., the average of flow completion times of flows that finish last in each job, along the Y-axis. As expected, FlowBender performs better with smaller fan-in degrees (larger flow sizes) than with larger fan-in degrees (smaller flow sizes). FlowBender's performance, like that of RPS and DeTail, suffers with a larger fan-in degree due to synchronized packet arrivals to the destination ToR with the receiver's last hop being *the* bottleneck i.e., multipathing does not help in principle. Overall, across different fan-in degrees, FlowBender achieves a reduction in the average job completion time by a factor of *four* in the best case (fan-in degree of 4), and by a factor of two in the worst case (fan-in degree of 32), and closely matches the performance of other expensive schemes such as DeTail and RPS.

4.2.6 Sensitivity Analysis

In this section, we perform sensitivity analysis of FlowBender, by varying two of its primary knobs: N, the number of RTTs a sender must be congested for before switching paths, and T, the congestion threshold.

In Figure 6, we vary N along the X-axis and show the mean latency across different flow sizes normalized to the default setting i.e., $N = 1$. As expected, as N increases, FlowBender's performance suffers as its response slows down with higher values of N. Nevertheless, the performance variation to N is *marginal*, and FlowBender exhibits robust performance across a broad range of values.

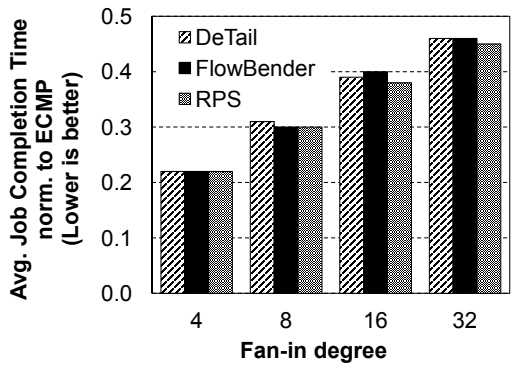

Figure 5: Partition Aggregate Workloads: Avg. Job Completion Time

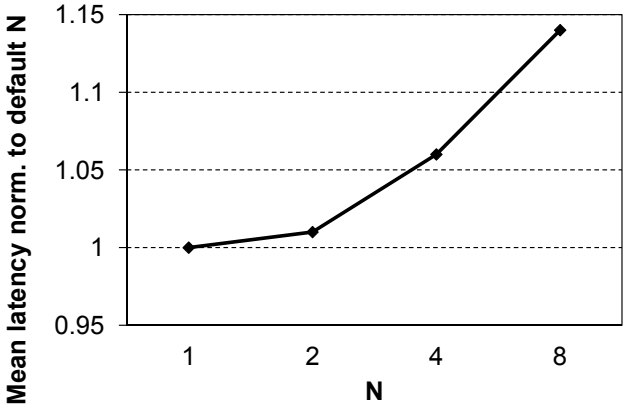

Figure 6: Sensitivity to N

In Figure 7, we vary T, the congestion threshold, along X-axis, and show the mean latency across different flow sizes normalized to the its default setting i.e., $T = 5\%$. For small values of T i.e., $T = 1\%$, FlowBender suffers marginal performance degradation as it responds to bursty spikes in the fraction of marked packets. We attain the best performance for $T = 5\%$. Increasing T beyond 5% slows down FlowBender's response, causing marginal performance degradation. Overall, similar to N, we see that FlowBender's performance is relatively robust across a wide range of T values.

4.3 Testbed Implementation

Our real implementation (testbed) has 15 ToR switches with 12 to 16 servers each. The servers are connected to the ToRs via 10*Gbps* links, and the ToRs are interconnected via 4 aggregation switches with one 10*Gbps* link to each of the 4 switches. In other words, each server has 4 distinct paths to reach any other server on the other ToR. Servers are running with Linux 3.0, including the aforementioned FlowBender changes (less than 50 lines of code added to the kernel) and the DCTCP implementation, and have their RTO set to $10msec$. We use standard ECMP-capable switches with a shared buffer space of $2MB$. The switches are configured with the CE marking threshold set to $90KB$.

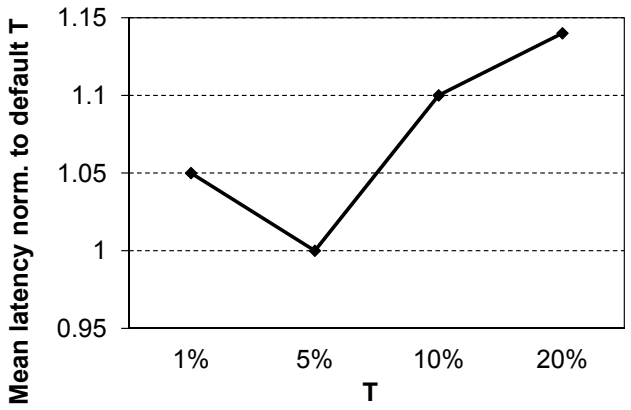

Figure 7: Sensitivity to T

4.3.1 All-to-All Traffic

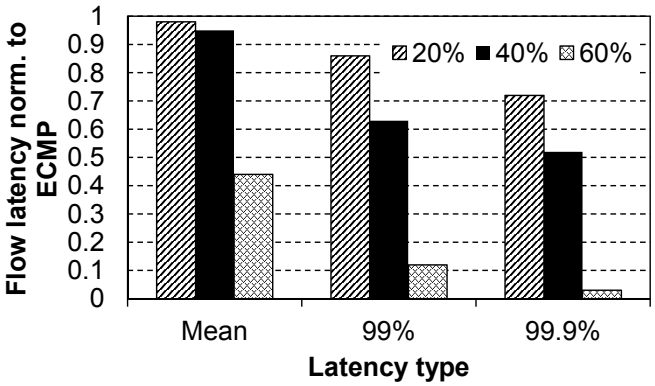

Figure 8: FlowBender's Latency Reduction at 20, 40, and 60% load (Bisectional)

In this experiment servers on one ToR initiate 1 MB flows randomly to any other server in the network with exponential inter-arrival times at a rate that cumulatively amounts to 20%, 40%, or 60% average utilization across the bisectional links. We initiate a total of 1.2 million flows, and wait for all flows to finish. We use the default TCP re-ordering threshold of 3, and monitor the out-of-order delivery numbers to ensure that FlowBender does not introduce *undesired CPU (processing) overhead*. To reconfirm that FlowBender does not lead to any abnormal packet re-ordering activity, we re-ran our experiments with a TCP re-ordering threshold of 30, and we didn't see any noticeable difference in performance.

In Figure 8, we show the *mean*, the 99^{th} percentile, and the 99.9^{th} percentile latencies along the X-axis, and FlowBender's completion time normalized to that of ECMP along the Y-axis. We use default parameter settings for FlowBender i.e., $N = 1$ and $T = 5\%$. FlowBender improves the 99^{th} and 99.9^{th} percentiles by $15 - 26\%$ and $34 - 45\%$ respectively, in comparison to ECMP. At 60% load, FlowBender's flows finish more than *twice* as fast as ECMP on average, and $87 - 96\%$ faster at the tail end.

A real implementation has a number of performance-affecting system-level details (e.g., application-level delays, kernel la-

tencies, offload inefficiencies, CPU power-saving modes, etc.) which are typically absent in simulations. Accordingly we see that the testbed results are not quantitatively equal to that of simulations. However the qualitative results are similar and support the claim that FlowBender offers drastic improvement over static schemes like ECMP.

4.3.2 Decongesting HotSpots

We now evaluate FlowBender's ability to decongest flows by re-routing them around hotspots. We simplify our traffic matrix in this case and initiate an all-to-all random shuffle of $1MB$ TCP flows from one ToR to another (all in the same direction). The aggregate TCP traffic generates $14Gbps$ from the sending ToR (arbitrarily spread across the 4 $10Gbps$ links). We also initiate 1 UDP flow between the same pair of ToRs, in the same direction as the TCP traffic, and rate limit it to $6Gbps$. The purpose of the UDP flow is to create a static (asymmetric) hot spot along one of the four paths given that this flow will not be re-routed or load balanced by FlowBender. We denote the path which this UDP flow hashes on by U.

Note that the aggregate TCP and UDP traffic on the four routes between the sending and the receiving ToRs amounts to 20Gbps. Hence, in an ideal setting, one would wish that the $14Gbps$ would have been equally split across the three paths other than U given that $14/3Gbps$ is still less than UDP's $6Gbps$ that was already routed on U. With ECMP, on one hand, U was unsurprisingly getting around quarter of the TCP traffic ($14/4 = 3.5Gbps$) obliviously mapped to it, thus ending up with around $9.5Gbps$ on average in total, driving that link practically unstable. FlowBender, on the other hand, succeeded in load balancing the traffic to a great extent with only around $1.5Gbps$ of the $14Gbps$ going on U. This experiment confirms FlowBender's ability to adaptively re-route around congestive hotspots in the network, and to respond to congestion created by non-TCP traffic.

The above experiment is also interesting from a Weighted Cost Multipathing (WCMP) perspective (as opposed to ECMP) in the context of asymmetric topologies, where in order to reach a certain destination group, the viable ports at a switch are configured with different forwarding weights so as not to prematurely oversubscribe those paths with lower capacities. One of the challenges with WCMP is to be able to reflect the different weights of the forwarding ports accurately, which is highly dependent on how many entries the forwarding table can accommodate as per current ECMP implementations (i.e. larger tables can represent the different weights with higher granularity). The significance of this experiment is in how even if the forwarding weights suffer some inaccuracy because of the forwarding table being constrained to have few entries only (as is the case with our testbed and most of the commodity switches), FlowBender is able to dynamically re-adjust the traffic on the different available paths such that those with lower capacities are not severely congested (i.e. more robustness to forwarding weight misconfigurations or chip limitations).

4.3.3 Topological Dependencies

Our simulation and testbed results above are based on a topology with 8 and 4 different paths between any pair of pods or ToRs respectively. A question that commonly arises here is: how helpful would FlowBender be when the path diversity between any pair of pods increases? In other words, what role does FlowBender play if the port density of each switch is, say, doubled, together with the number of servers per ToR, while keeping the over-subscriptions ratios the same (i.e. the path diversity quadruples)? The extent to which FlowBender helps clearly depends on the number of the available paths P, but it also depends on the number of those larger flows L that we're trying to spread out on those paths. More precisely, and particularly true in the limits, the performance improvement depends on the ratio $R = L/P$ of the two numbers as we show next, which is typically expected to remain constant given that as the bisectional capacity (i.e. P) is scaled up, the load, and hence, L would be also scaled up proportionally to maintain the same utilization.

Considering the micro benchmark basic validation results discussed earlier, where FlowBender is shown to do a good job in evenly spreading the flows across the different paths, FlowBender's performance improvement amounts to how bad ECMP's flow distribution performance was in the first place. Given the oblivious nature of ECMP, the distribution of the number of large flows per each route is, in steady state, a very straightforward binomial distribution with a mean R and a variance $R(1 - 1/P)$, which is therefore not that different for a reasonably large P. For example, varying P from 8 to 32 would increase the variance by less than 11% only and hence would have a negligible effect in practice. In fact, we reran our All-to-All experiments with a different fan-out degree, and the performance improvement due to FlowBender was almost the same.

5. FURTHER OPTIMIZATIONS AND FUTURE WORK

Our paper presents one of the most basic versions of FlowBender. As discussed earlier, we have intentionally resisted any temptations to optimize FlowBender in the interest of demonstrating how effective this simple idea is, and our evaluation results confirm our position for all the practical cases that we have discussed. Looking forward, however, FlowBender might be desired to operate in more challenging environments, potentially outside datacenters, and could benefit from some of the suggestions summarized below.

5.1 Stability

FlowBender does not monitor the load on all available paths before taking a rerouting decision. Instead, it randomly chooses a new path when it detects that the current path is congested. Therefore, the new path it reroutes to may be also congested. If that happens to be the case because say of an incast episode or because the network is highly congested in general, FlowBender will trigger yet another path change, and the rerouting process may repeat. In some pathological cases, such a design could continuously thrash from one path to the other if no further measures for preventing this are taken. Because every rerouting instance carries the potential of out-of-order delivery of packets to the receiver, such thrashing is undesirable.[5] Accordingly, FlowBender can be extended to limit the number of path

[5]Even though such artifacts are more likely to occur in an incast situation, the extent to which they occurred in our simulations was negligible as is evident from the improvement ratios discussed earlier. We do bring up the feature suggested herein, however, so as to add more confidence around FlowBender's ability to handle some of those quite

changes that could occur when the network is severely congested. More specifically, FlowBender can be constrained to switch paths for a maximum of S consecutive times before it goes into a *locked* state. In the locked state, it will pick one path out of the last S paths that had the lowest value of F, the fraction of ECN-marked ACKs, and will lock in to that path for the next U RTTs. At the end of the locked phase, FlowBender resumes business as usual, tracking F and switching paths if it exceeds T for N consecutive RTTs. Note that if we were to choose S and U as 5 and 10, respectively, with an N of 2 (which gives almost the same performance as the default $N = 1$ configuration), then we would be limiting the rerouting events to a maximum of 5 times in every $5 \times 2 + 10 = 20$ RTTs, thus significantly limiting the number of out-of-order packets that could be triggered by FlowBender.

5.2 Gradual Rerouting

Instead of shifting the traffic of a congested flow all at once from one route to another, the rerouting process can be attempted gradually. Of course, this is assuming the transport layer is adjusted to tolerate out-of-order packe delivery. For example, we can start by initially transmitting 10% of the traffic of a congested flow on a new path[6] while monitoring whether this has any effect on alleviating congestion. If congestion has been already mitigated by partially migrating to the new path, then this could be an indication that more traffic should be routed on the new path (e.g. go up to transmitting 20% of the traffic on the new path and so on). Otherwise, it could be that the desire to send on a different, uncongested path was not really met (potentially because of partially or fully overlapping with the existing path at one or more congested links), so the flow would attempt to send on a different new path instead (i.e. abandon the intermediate one).

The above discussion has focused on the case of load balancing across 2 subflows, which might be sufficient for most practical purposes, but the same approach can be extended to load balancing across more than two paths simultaneously.

5.3 Selective Rehashing

In our current implementation, we change the value of V for the flexible hashing input field obliviously once congestion occurs. Changing the value of V, however, does not always mean that the route will change as this really depends on the hashing function. That said, given sufficient knowledge about the hashing functions in the fabric, one can avoid this artifact by having each flow precompute a number of potential values for V that would result in hashing to different paths. The process for precomputing such values might be very challenging and time consuming to perform for general hashing functions, but if the network operators choose their hashing functions in a way such that flipping one of the hashing inputs bits would result in a different hashing output (e.g. XOR hash functions), then this process would become much easier to perform. Alternatively, flows can correlate the different RTT estimates or values for F corresponding to different V's in order to infer, with a high probability, how these V's map to different paths and avoid choosing a redundant value for rerouting.

5.4 Proactive Probing and Route Caching

We have demonstrated the reactive version of FlowBender, where a flow is rerouted only once congestion has been detected. Of course, one might argue that FlowBender is already quite prompt in rerouting when congestion arises, given its low congestion detection threshold T, but the load balancing performance could be still further improved by allowing a flow to proactively probe across the different paths. By probing we mean that a flow can periodically send a few of its packets with different V values, keeping track of their sequence numbers ranges, and check once they have been acked which of them have experienced congestion. One of those V's corresponding to probe packets which did not seem to be congested at all could be proactively selected as the basic V once F has exceeded a threshold T' smaller than T. Alternatively, a flow may attempt to keep track of some of those *better* V's to hash upon once congestion occurs, instead of choosing V obliviously, or might simply blacklist those highly congested V's and avoid revisiting them until sufficient time has elapsed.

5.5 Rerouting and Flow Control

Rerouting, in the load balancing sense, and flow control, in the congestion control sense, are two ways for handling congestion in multipath environments. When both mechanisms rely on the same congestion signal (e.g. ECN driving both, FlowBender and DCTCP), it becomes useful to decide on which of the two mechanisms should be triggered based on the nature of the congestion event. For example, if congestion is occuring at the receiving ToR (e.g. an incast event), then it is unlikely that rerouting the individual congested responses will help (though rerouting still did not have a noticeable negative impact on the overall performance per our simulation results). Alternatively, if a flow is mainly bottlenecked at a core switch while other core switches are lightly loaded, then slowing down the rate of this flow would be rather harmful when it could have been rerouted instead. In our evaluation for FlowBender, we did not attempt any optimizations along these lines given that DCTCP would not have kicked in aggressively yet once the very low threshold T would have been just exceeded. In the future, however, FlowBender and other congestion control and load balancing mechanisms could significantly benefit from more informative congestion control signals that can somehow distinguish whether congestion is occuring at the edge of the fabric and/or somewhere else.

6. CONCLUSION

In this paper, we proposed a new load balancing mechanism called FlowBender. The main motivation for introducing FlowBender is to overcome the limitations of oblivious hashing schemes such as ECMP, prominent in today's datacenters, without suffering from high packet re-ordering or requiring custom hardware changes and complicated host mechanisms that could offset any potential benefits. FlowBender is a host-based, congestion-driven scheme that distributively re-routes individual flows around congested hotspots only once congestion is experienced, end-to-end, or when link failures occur. FlowBender relies on ECN and ECMP

unlikely pathological scenarios that our evaluations did not cover.

[6]This can be done in a periodic manner to avoid CPU-intensive computations for generating random numbers (e.g. every 10th pkt is sent on a new path).

switch support, which is typical in today's datacenters, and implements the load-balancing logic at the end-hosts via a very straightforward kernel patch. In contrast to centralized flow scheduling schemes, FlowBender recovers from link failures after an RTO occurs and reroutes congested flows at the RTT granularity, several orders of magnitude faster than state of the art routing management schemes.

Our ns-3 [4] simulations, with workloads representative of datacenter applications, show that FlowBender *substantially* reduces the mean and tail latency compared to ECMP, while achieving performance very similar to that of other expensive or undesirable schemes like DeTail and RPS. In particular, our tail latencies are reduced by more than 5x to 20x relative to ECMP's, and our partition-aggregate jobs are about 2x to 4x faster on average. We have also evaluated FlowBender using a real implentation and found that it cuts the flow completion tail latencies by around 40% relative to ECMP's for large flows.

7. REFERENCES

[1] 802.1Qbb - priority-based flow control. `http://www.ieee802.org/1/pages/802.1bb.html`.

[2] Avoiding network polarization and increasing visibility in cloud networks using broadcom smart hash technology. `http://www.broadcom.com/collateral/wp/StrataXGS_SmartSwitch-WP200-R.pdf`.

[3] Cisco cli command reference. `http://www.cisco.com/en/US/docs/wireless/asr_901/Command/Reference/Cmdref_asr901.html`.

[4] NS-3 network simulator. `http://www.nsnam.org/`.

[5] M. Al-Fares, A. Loukissas, and A. Vahdat. A scalable, commodity data center network architecture. In *Proceedings of the ACM SIGCOMM 2008 conference on Data communication*, SIGCOMM '08, pages 63–74, New York, NY, USA, 2008. ACM.

[6] M. Al-Fares, S. Radhakrishnan, B. Raghavan, N. Huang, and A. Vahdat. Hedera: dynamic flow scheduling for data center networks. In *Proceedings of the 7th USENIX conference on Networked systems design and implementation*, NSDI'10, pages 19–19, Berkeley, CA, USA, 2010. USENIX Association.

[7] M. Alizadeh, A. Greenberg, D. A. Maltz, J. Padhye, P. Patel, B. Prabhakar, S. Sengupta, and M. Sridharan. Data center tcp (dctcp). In *Proceedings of the ACM SIGCOMM 2010 conference*, SIGCOMM '10, pages 63–74, New York, NY, USA, 2010. ACM.

[8] T. Benson, A. Akella, and D. A. Maltz. Network traffic characteristics of data centers in the wild. In *Proceedings of the 10th ACM SIGCOMM conference on Internet measurement*, IMC '10, pages 267–280, New York, NY, USA, 2010. ACM.

[9] A. R. Curtis, W. Kim, and P. Yalagandula. Mahout: Low-overhead datacenter traffic management using end-host-based elephant detection. In *INFOCOM*, pages 1629–1637. IEEE.

[10] A. R. Curtis, J. C. Mogul, J. Tourrilhes, P. Yalagandula, P. Sharma, and S. Banerjee. Devoflow: scaling flow management for high-performance networks. In *Proceedings of the ACM SIGCOMM 2011 conference*, SIGCOMM '11, pages 254–265, New York, NY, USA, 2011. ACM.

[11] W. J. Dally. Virtual-channel flow control. *IEEE Trans. Parallel Distrib. Syst.*, 3(2):194–205, Mar. 1992.

[12] D. M. Dias and M. Kumar. Preventing congestion in multistage networks in the presence of hotspots. In *ICPP (1)'89*, pages 9–13, 1989.

[13] A. Dixit, P. Prakash, Y. Hu, and R. Kompella. On the impact of packet spraying in data center networks. In *INFOCOM, 2013 Proceedings IEEE*, pages 2130–2138, 2013.

[14] J. Kim, W. J. Dally, and D. Abts. Adaptive routing in high-radix clos network. In *Proceedings of the 2006 ACM/IEEE conference on Supercomputing*, SC '06, New York, NY, USA, 2006. ACM.

[15] C. E. Leiserson. Fat-trees: universal networks for hardware-efficient supercomputing. *IEEE Trans. Comput.*, 34(10):892–901, Oct. 1985.

[16] J. Mudigonda, P. Yalagandula, M. Al-Fares, and J. C. Mogul. Spain: Cots data-center ethernet for multipathing over arbitrary topologies. In *Proceedings of the 7th USENIX Conference on Networked Systems Design and Implementation*, NSDI'10, 2010.

[17] C. Raiciu, S. Barre, C. Pluntke, A. Greenhalgh, D. Wischik, and M. Handley. Improving datacenter performance and robustness with multipath tcp. In *Proceedings of the ACM SIGCOMM 2011 conference*, SIGCOMM '11, pages 266–277, New York, NY, USA, 2011. ACM.

[18] C. Raiciu, C. Paasch, S. BarrÈ, A. Ford, M. Honda, F. Duchene, O. Bonaventure, and M. Handley. How hard can it be? designing and implementing a deployable multipath tcp. In *USENIX Symposium of Networked Systems Design and Implementation (NSDI'12), San Jose (CA)*, 2012.

[19] J. H. Saltzer, D. P. Reed, and D. D. Clark. End-to-end arguments in system design. *ACM Trans. Comput. Syst.*, pages 277–288, 1984.

[20] J. R. Santos, Y. Turner, and G. (john Janakiraman. End-to-end congestion control for infiniband. In *In proceedings of Infocom03*, 2003.

[21] B. Vamanan, J. Hasan, and T. Vijaykumar. Deadline-aware datacenter tcp (d2tcp). In *Proceedings of the ACM SIGCOMM 2012 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication*, SIGCOMM '12, 2012.

[22] D. Wischik, C. Raiciu, A. Greenhalgh, and M. Handley. Design, implementation and evaluation of congestion control for multipath tcp. In *Proceedings of the 8th USENIX conference on Networked systems design and implementation*, NSDI'11, pages 8–8, Berkeley, CA, USA, 2011. USENIX Association.

[23] D. Zats, T. Das, P. Mohan, D. Borthakur, and R. Katz. Detail: reducing the flow completion time tail in datacenter networks. In *Proceedings of the ACM SIGCOMM 2012 conference on Applications, technologies, architectures, and protocols for computer communication*, SIGCOMM '12, pages 139–150, New York, NY, USA, 2012. ACM.