2013 IEEE 33rd International Conference on Distributed Computing Systems

Longer is Better: Exploiting Path Diversity in Data Center Networks

Fung Po Tso*, Gregg Hamilton*, Rene Weber†, Colin S. Perkins* and Dimitrios P. Pezaros*
*School of Computing Science, University of Glasgow, G12 8QQ, UK
†Department of Computer Science, Chemnitz University of Technology, 09107 Chemnitz, Germany
Email: posco.tso@glasgow.ac.uk, g.hamilton.3@research.gla.ac.uk, rene.weber@s2011.tu-chemnitz.de,
csp@csperkins.org, dimitrios.pezaros@glasgow.ac.uk

***Abstract*—Data Center (DC) networks exhibit much more centralized characteristics than the legacy Internet, yet they are operated by similar distributed routing and control algorithms that fail to exploit topological redundancy to deliver better and more sustainable performance. Multipath protocols, for example, use node-local and heuristic information to only exploit path diversity between shortest paths. In this paper, we use a measurement-based approach to schedule flows over both shortest and non-shortest paths based on temporal network-wide utilization. We present the *Baatdaat*[1] flow scheduling algorithm which uses spare DC network capacity to mitigate the performance degradation of heavily utilized links. Results show that *Baatdaat* achieves close to optimal Traffic Engineering by reducing network-wide maximum link utilization by up to 18% over Equal-Cost Multi-Path (ECMP) routing, while at the same time improving flow completion time by 41% - 95%.**

***Index Terms*—Traffic Engineering, Multipath Scheduling, Unequal-cost Multipath, Data Center Networks, Software Defined Networking**

I. Introduction

Admittedly, the Internet has known huge success mainly due to the complete decentralization of the infrastructure, and the bandwidth over-provisioning of point-to-point links carrying aggregate traffic over the network core. Today, Cloud computing is emerging as an important paradigm where processing and communication resources are concentrated and hosted over generic Data Center (DC) infrastructures that need to accommodate a wide range of services, from private data processing to public website hosting. Although Cloud computing lends itself to much more centrally orchestrated resource management and provisioning approaches, the underlying DC networks are operated using commodity packet communication mechanisms, such as shortest path distributed routing protocols and traffic engineering based on node-local optimizations. This results in sub-optimal overall network operation, where congestion is exhibited across a significant number of links [1] especially at higher layers in the topology, while others are idling [2]. A number of traffic engineering approaches have been devised to alleviate network hotspots mainly by using redundant shortest paths [3] but, they either fail to take temporal network-wide state into consideration [4] or require applications to explicitly make the network aware of traffic priorities [5]. Such schemes suffer from inherent limitations of pseudo-random algorithms (e.g., collisions of packet header hashes) [6] or in the legacy Internet-like mechanisms they use that, e.g., limit path diversity to usually two distinct shortest paths between source-destination server pairs. In general, they attempt to solve a global optimization problem when they only use distributed node-local state [3][4][6].

In this paper, we introduce *Baatdaat*, a novel flow scheduler that exploits path diversity to reduce congestion and increase the usable capacity of DC topologies. *Baatdaat* is a logically centralized system compatible with OpenFlow [7] that uses direct measurement in short timescales to construct a network-wide view of temporal bandwidth utilization, and to subsequently schedule flows over both shortest and non-shortest paths to further exploit path redundancy and spare capacity in the DC. In contrast to existing traffic engineering approaches, *Baatdaat* does not rely on the ability to predict or schedule flows based on their size [6][5], nor does it assume a known traffic matrix [8] which would be unsuitable for the highly unpredictable DC traffic dynamics. Unlike typical *match-commit* Software-Defined Networking (SDN) approaches, *Baatdaat* uses a combination of centralized and distributed measurement and control components to synthesize global knowledge from node-local performance information and to subsequently enforce network-wide flow scheduling. We employ OpenFlow running on NetFPGA programmable switches [9] that allows packet processing to be performed at line-speed and enable real-time dynamic flow scheduling. Node-local link utilization measurements are aggregated to a central controller which then computes and installs least utilized outgoing paths on each switch. Flows can then be scheduled over non-shortest lightly utilized paths to avoid congestion. Our results show that by avoiding congested shortest paths, we can significantly reduce the overall network utilization while at the same time improve flow completion times, i.e., avoiding congestion pays off for the extra hops taken by a subset of flows. *Baatdaat* does not require any modifications to applications that operate over DC environments, while its combination of centralized and distributed components makes it more scalable than typical OpenFlow deployments over DCs [10].

The main contributions of this work include:

- A hardware-assisted traffic monitor that measures link utilization on the switches at line-speed without impacting their forwarding performance.
- A SDN-based adaptive flow scheduling system that or-

[1] *Baatdaat* is Cantonese for "reachable in all directions".

1063-6927/13 $26.00 © 2013 IEEE
DOI 10.1109/ICDCS.2013.36

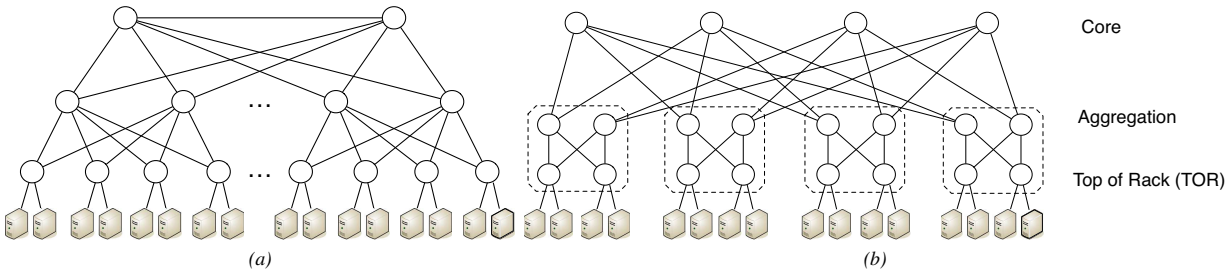

Fig. 1: (a) Canonical tree topology and (b) Fat-tree topology

chestrates and enforces network-wide traffic engineering based on link-local metrics.

- Improved congestion alleviation in DC networks over existing scheduling algorithms by spreading load over shortest and detour paths, while at the same time reducing flow completion time.

Our results show that *Baatdaat* can achieve close to optimal Traffic Engineering by improving over ECMP by up to 18% for different types of load [5] while only deviating by 3% from optimal, on average. At the same time, flow completion is improved by up to 10%.

Although long-term provisioning of DC infrastructures is necessary to accommodate growth in service demand, the improved network utilization offered by *Baatdaat* over short timescales can provide the necessary short-term traffic shaping to avoid resource outages at the onset of sudden traffic dynamics.

The remainder of the paper is structured as follows. Section II discusses the design and implementation of *Baatdaat*, and its different hardware and software components. Section III presents in detail the algorithmic operation of *Baatdaat* scheduling. Section IV evaluates the performance of *Baatdaat* over testbed and simulated environments. Section V discusses related work and Section VI concludes the paper.

II. System Design & Architecture

In this section we present the design of *Baatdaat*, with particular emphasis on our implementation of link utilization modules on the NetFPGA platform. We focus the discussion on typical DC topologies, such as canonical [11] and fat-tree [12] as shown in Fig. 1.

A. Design Requirements

A system that performs non-shortest-path flow scheduling in DCs based on real-time link status feedback has to meet the following design requirements:

- R1: Each switch should be able to monitor link utilization of its associated links at line rate.
- R2: Flows should be allowed to opportunistically take detours (i.e., longer paths) when it is beneficial to do so.
- R3: Global flow scheduling should easily scale to the size of a DC network.

While current switches do not support real-time link measurements, R1 requires a new hardware-based module to conduct line-rate measurement while not impacting the device's packet forwarding performance. We have implemented such a component alongside NetFPGA's [9] OpenFlow implementation. We will show in later sections that our implementation is highly efficient and maintains line-rate throughput.

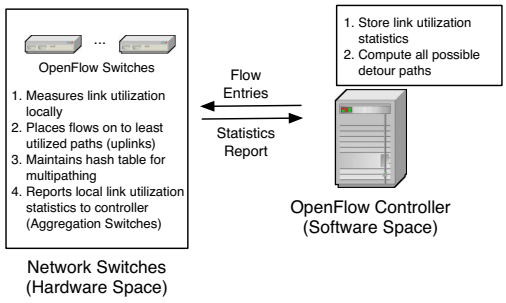

Fig. 2: System architecture.

R2 is highly desirable given the measurement studies revealing that DC networks generally remain underutilized, albeit with a small fraction of congested links [2]. Using non-shortest paths to route traffic can improve network-wide utilization, so long as it doesn't impact individual flow performance.

Scalability, R3, is crucial for centralized approaches since performance of the controller has a direct impact on the network as a whole. In order not to turn the controller into the bottleneck, we have adopted a mix of distributed and centralized scheduling approaches for *Baatdaat*. First, all switches individually monitor their link utilization and independently schedule flows onto the links, while the controller only determines detour paths. This way, the controller's workload can be significantly reduced.

B. System Architecture

The *Baatdaat* architecture consists of OpenFlow switches in DC-compatible topology, and a single OpenFlow controller to collect link utilization statistics among aggregation switches (in the multilayered tree structure, links seen by ToRs are also exposed to aggregation switches) and to determine flow detour paths, as shown in Fig. 2. The single controller can create a performance bottleneck and a single point of failure in the network. In *Baatdaat*, the use of multiple controllers is permitted. For example, one controller can be deployed for each pod in a fat tree or its equivalent counterpart in the canonical tree. Multiple controllers can work independently requiring no synchronization between them, making the control plane

scalable and robust to failures. Each controller can interact with switches within a pod and compute pod-local paths, if detours are restricted at the pod-level (in both canonical tree and fat-tree). As we will see in section IV, this is the most beneficial mode of operation for *Baatdaat*, i.e., when detours are only allowed within each pod.

C. Benefits of Hardware Implementation

In order to understand the impact of continuously gathering throughput information on each link, we have implemented link throughput measurement on each of three OpenFlow platforms: NetFPGA hardware; the Open vSwitch kernel module [13]; and the Stanford reference user-space OpenFlow implementation [7].

Each measurement module periodically polls for the total number of bytes transferred and received on each network interface. The method of polling is specific to each implementation. The NetFPGA implementation polls the hardware; Open vSwitch reads the statistics from each `net_device` structure; and OpenFlow parses data from `/proc/net/dev`. After each successive reading, the throughput over the polling interval between the current and previous reading is calculated and communicated to the central OpenFlow controller. The current polling interval is 1ms.

Experiments were conducted to evaluate the performance of the switches with our link utilization measurement capabilities implemented. The experimental testbed setup was two end-host systems connected via a switch. The two end-hosts were AMD Athlon 64 X2 5600+ systems, each with 3.5GB RAM and a 82571EB Gigabit Ethernet NIC installed. All experiments used a NetFPGA box as the switch, consisting of an AMD Phenom II X4 965 with 3.5GB RAM, a 4x1G NetFPGA card, and a 2x1GB Intel 82571EB Gigabit Ethernet NIC. The NetFPGA card was used to test the hardware implementation, and the Intel NIC was used to test the kernel and user-space implementations, while keeping the base system the same. The NetFPGA reference switch was used as an experimental baseline for comparison against the cost of our measurement-enabled implementations.

Our first experiment was to have one end-host ping the other. 10,000 back-to-back pings were taken for each platform and the average taken, with the results presented in Table I. As our system is distributed in nature and pushes data collection tasks to the switches, we also wanted to evaluate the strain placed on each system, and to understand any degradation in the forwarding performance of our implementations. We ran throughput tests using Iperf with TCP and a variety of UDP datagram sizes and took readings of CPU utilization on the switch for the duration of each throughput test. The TCP and UDP throughput tests were each run for 60 seconds. The CPU utilization results are presented in Table II. For UDP, the average CPU usage across all packet sizes is presented.

As Table I shows, the NetFPGA implementation shows a lower ping time than the software implementations, revealing the clear benefits of offloading the work to dedicated on-board hardware. The TCP throughput achieved by both the NetFPGA reference switch implementation and our measurement-enabled NetFPGA OpenFlow switch was 940 Mb/s, showing

TABLE I: Ping times across different switch implementations.

Switch Type	NetFPGA Ref. Switch	NetFPGA	Open vSwitch Kernel	OpenFlow User-Space
Min. Response (ms)	0.051	0.055	0.143	0.157
Avg. Response (ms)	0.062	0.072	0.234	0.289

TABLE II: Average CPU usage across different switch implementations.

Switch Type	NetFPGA Ref. Switch	NetFPGA	Open vSwitch Kernel	OpenFlow User-Space
TCP	0.01%	0.04%	6.57%	25.8%
UDP	0.00%	0.03%	13.74%	27.86%

that our implementation does not impact forwarding performance noticeably. Similarly, both the reference implementation and our own achieved a throughput of 956 Mb/s. The UDP throughput results also support the claim that our measurement-enabled hardware implementation has no noticeable impact on the forwarding performance of our switch, when compared to the reference switch implementation.

While our hardware implementation appeared to display similar throughput capabilities to the reference implementation hence demonstrating that our throughput measurements have negligible effect on the forwarding performance, this is not the full picture. Table II reveals that the reference NetFPGA switch implementation only puts a load of 0.01% on the CPU, while our measurement-enabled NetFPGA OpenFlow switch imposes a load of only 0.04% in the worst case, showing the benefits of hardware acceleration. By comparison, Open vSwitch and user-space OpenFlow forwarding require 13.74% and 27.86% of CPU time in the worst case, respectively.

These results allow us to draw two conclusions about our measurement-enabled OpenFlow implementations. First, we do not negatively impact the line-rate forwarding capabilities with our hardware NetFPGA implementation, achieving the same throughput rates as the reference NIC implementation. Second, the NetFPGA implementation benefits significantly from offloading all packet processing to the NetFPGA, resulting in lower ping times, and lower (almost non-existent) CPU utilization when compared to Open vSwitch or user-space OpenFlow. These results clearly reveal the benefits of pushing data collection logic to OpenFlow switches containing programmable hardware in order to significantly reduce CPU load and network response times.

D. Measurement Module Design

In OperFlow's operational paradigm, forwarding decisions are made by a logically centralized controller, which can in turn add and remove forwarding entries in OpenFlow switches. This form of Software-Defined Network (SDN) abstracts complexity from hardware to controller software, allowing forwarding behavior to be programmed into a central controller that communicates with switches via in-band control messages or via directly-connected out-of-band links.

Each OpenFlow switch matches incoming flows using exact-match or wildcard filters on specific protocol fields. If a

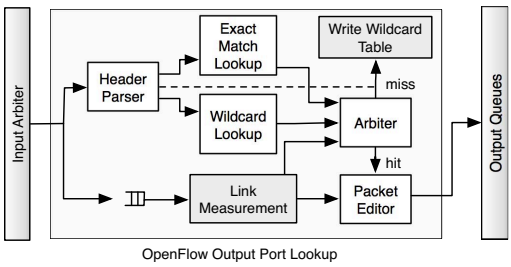

Fig. 3: Design of the link utilization module within the OpenFlow switch pipeline.

match is not found for an arriving packet, the packet is sent to the controller which registers the new flow and decides on the action(s) that should be applied to all subsequent packets matching the same filter. The action is then pushed to the switch and cached as an entry in the switch's flow table. Subsequent packets belonging to the same flow are then directly forwarded without the need for redirection to the controller.

We have implemented a hardware-based link measurement module in-line with NetFPGA's OpenFlow switch implementation [14] as illustrated in Fig. 3. As soon as a packet arrives at the input port, the link utilization module adds the size of the packet to the overall bytes arriving on the specific link. Similarly, after the packet is processed by the output port lookup, its size will be added to the total for the outgoing link. After a specified interval link utilization will be calculated by counting the number of bytes sent and received on a link and subtracting these from the stored byte counts from the previous calculation.

E. Hardware Multipathing

OpenFlow v1.0 does not support multipathing (OpenFlow v1.1 has added support for ECMP, however the reference hardware implementation is not yet available). We hence needed to enable hardware multipathing to take advantage of path redundancy of the physical infrastructure.

The lack of native support for multipathing means switches are not able to do line-rate multipathing locally as forwarding decisions are still dictated by the controller even if there is multipath support. For example, as the *exact match table* takes precedence over the *wildcard table* in the normal mode of operation, whenever there is a match miss in the *exact match table*, the packet will be matched against the *wildcard table*. If both tables return a miss, the flow will be treated as a new flow and will be forwarded to the controller. However, in *Baatdaat* we want some multipathing decisions to be made locally. The other modification required by *Baatdaat* is to enable a forwarding table such that each switch knows (when) there are more than one outgoing ports for a flow. Hence, in *Baatdaat*, the *wildcard table* is also a forwarding table and the shortest equal cost paths are always the default entries. Detour paths, if determined by the controller, will also be added to the wildcard table. However, there is an extra field in each table entry storing the link utilization. Every new flow (no match in the *exact match table*) will be matched against the wildcard table with their destination IP initially, and the one with least utilization among the set of paths will be used. Eventually, an exact match entry for the flow will be created and all subsequent packets will be matched against this.

Switches supporting multipathing need to recognize each flow individually, which is typically achieved by hashing one or more tuples in the packet's headers, as is the case in ECMP. The reference implementation in [9] already comes with a flow hashing module, the *header parser*, which we can exploit to enable native flow identification at line rate.

Next, since multipathing can deteriorate the performance of stateful protocols such as TCP due to out-of-order packet delivery, *Baatdaat* ensures that all packets belonging to the same flow are always scheduled over the same path even when the link utilization changes. Hence, maintenance of a flow table is required. To realize this on the NetFPGA, we hooked in an anchor between the header parser module and the arbiter module such that whenever both *exact match lookup* and *wildcard lookup* assert their *match_miss* signal –implying a newly arriving flow–, a new flow entry is created and added to the *exact match table* (i.e., in NetFPGA's SRAM). Therefore, successive packets will always have an *exact match hit* and follow the same path. The exact match table is virtually a hash table storing all active flows in the network. *Baatdaat* uses this table to prevent flow oscillation due to change of link utilization since scheduled flows will always follow the same path.

III. *Baatdaat* SCHEDULING

In this section we discuss the operation of the *Baatdaat* scheduling loop and address how the controller makes use of link utilization information to schedule flows on non-shortest-path routes.

Intuitively, the scheduling loop should be short enough to capture all flow inter-arrivals. In this case, centralized schedulers that decide which path to pin a flow to may be hard-pressed to keep up, due to too much overhead in processing control data. While it is reported that flow inter-arrivals per switch port can be as long as 10-15 ms for Cloud DCs, we set our measurement loop to run every 1 ms by default with an aim to capture instantaneous traffic bursts. The link-utilization measurement results are stored locally in each switch as an array of size equivalent to the number of ports in the switch. *Baatdaat* is a flow-based scheduler that uses 5-tuple header hashing to guarantee packets belonging to the same flow traverse the same path, so that packet re-ordering is avoided. Hence, when a new flow joins, the switch first queries the number of outgoing ports, such as output ports for multiple shortest paths, and then places the flow onto the least utilized link by comparing measured link utilizations. When all outgoing links are reported to have the same utilization, such as when all links have 0% utilization right after initialization, the flow is randomly scheduled using commodity ECMP.

Aggregation switches, however, need to report statistics to the controller, which will determine whether or not the new flow should take a detour. The report frequency is independent to the measurement interval and can be set to 100 ms.

Considering the latency in the control path, where processing delay can be up to 220 ms in the controller, it is unnecessary to report the statistics as frequently as link measurement.

To make the scheduling manageable, we impose two constraints on any flow taking a detour: 1) the flow is downlink traffic of the aggregation switches. This means detours only happen between the aggregation and the Top-of-Rack (ToR) layers of the DC topology. In DCs, cross-aggregation-switch traffic is more common than cross-core-switch due to the traffic locality nature of DC networks. Multi-rooted tree architectures, e.g., fat-tree, provide large amounts of interconnects between the aggregation-ToR layers. In particular, these are mesh interconnects in a fat tree topology. This offers significant path diversity if detour is allowed; 2) the flow can only take a detour of 2 hops longer than the shortest path to prevent oscillation and flooding-like effects in the network. We will show later in Sec. IV that a detour of 2 extra hops is an ideal choice.

While each switch keeps link utilization values locally in the form of an array, the controller maintains a *link utilization matrix*, M. Only aggregation switches update the OpenFlow controller with the utilization of their associated links. Assuming M is a $m \times n$ matrix, m denotes the number of aggregation switches in a pod, while n is the number of downlink ports on an aggregation switch. For ease of presentation, we assume switches and their ports are sequentially numbered. For example, M_{ij} is the link utilization of port *j* of switch *i*.

By allowing detours, path diversity in the fat tree network increases by $k/2 \times (k/2-1) \times (k/2-2)$ compared to the original shortest path scheme which provides a path diversity of $k/2$. The path utilization of detour paths is defined as max(*detour links*)$\times c$, where c is a weighting factor to reflect the fact that it is a longer path. In our simulation experiments we found that $c = 1.5$ works well as it gives good detour opportunity. We have to set $c > 1$ in order to bias longer paths, but setting $c \geq 2$ will see a significant decrease in detour opportunities.

The next question is, *how does the controller determine detour paths?* Clearly, as detour paths are limited to being no more than two hops longer than shortest paths and only happen in the aggregation-TOR layer, starting from any aggregation switch, any detour of 2 more hops will lead to a path of 4 hops. Hence, the scheduling algorithm starts by constructing an acyclic tree of depth 3, as shown in Fig. 4, with k switches as vertices and links among them as edges. This can be easily achieved by parsing M (i.e., adjacent matrix) as it contains all required link and port information in a pod.

Depth-first or breadth-first search can then be applied to identify and compute path utilization along the path. The OpenFlow controller, after determining a detour path for a new flow, installs the OpenFlow entry to all affected switches.

The time complexity of this search is $O((k/2-1)^2)$ for a fat-tree topology although the search tree has a depth of 3. For a k-ary fat tree, there are $k/2$ aggregation and ToR switches in a pod, respectively. In the example given in Fig. 4 we can see that the sequence for a detour path is: *(ToR switch* $\xrightarrow{link1}$ *aggregation switch* $\xrightarrow{link2}$ *ToR* $\xrightarrow{link3}$ *aggregation switch* $\xrightarrow{link4}$ *ToR)*. As *link 1* is independently chosen by the ToR switch itself, *link 2 – link 3 – link 4* is the actual detour determined by the controller. Therefore, starting from an aggregation switch, due to the densely interconnected nature of each pod, it has $k/2$ links to ToR switches including the source node. But the source ToR switch has to be excluded to prevent loops, so the algorithm only needs to search through (and to compute link utilization for) $k/2-1$ links at the first iteration (depth 1) for *link 1*. Similarly, there are $k/2-1$ iterations each at depth 2 for *link 3*. For *link 4*, due to the mesh-like interconnect of aggregation–ToR switches, the next hop must contain the destination node, and the algorithm does not need to search any further hence stopping at this level. Therefore, the overall complexity is $O((k/2-1)^2)$.

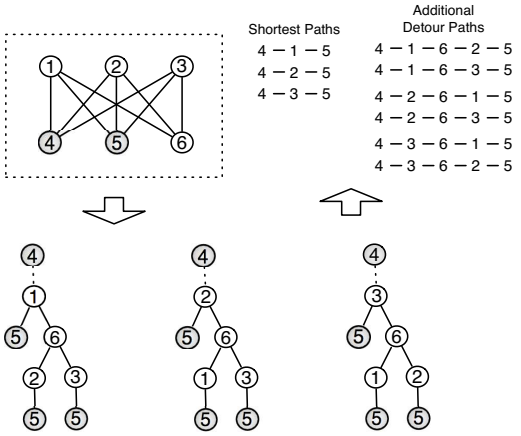

Fig. 4: Example of a calculated detour in a pod, assuming a flow is to be forwarded from node 4 to node 5.

These rules and algorithm should also apply to the canonical tree DC topology without modification. Assuming that, in such a topology, every set of m switches connect to two aggregation switches. Hence, the path diversity in between aggregation and ToR layers becomes $2\times(m-1)$ and complexity for path searching is $O(m)$.

IV. Performance Evaluation

In this section, we evaluate the performance of *Baatdaat* with respect to reduced network-wide maximum link utilization and improved load-balancing due to path diversity. In order to test the properties of our system at scale, we have used the ns-3 network simulator for network-wide properties, and our hardware-assisted testbed implementation to evaluate the footprint of the time-critical processing elements.

A. Network-wide Experimental Results

We have simulated a $k = 8$ fat-tree topology (128 servers grouped into 8 pods with 8 switches each) with 1Gb/s interconnect links in ns-3 with the OpenFlow module enabled. Simulated flows consist of uniformly chosen 4 KB, 8 KB, and 100 KB flows, to include the range of latency-sensitive flows common in DC networks, the majority of which are according to research findings, small and complete in one or two RTTs [15], [16]. As for the large flows which exceed 10%

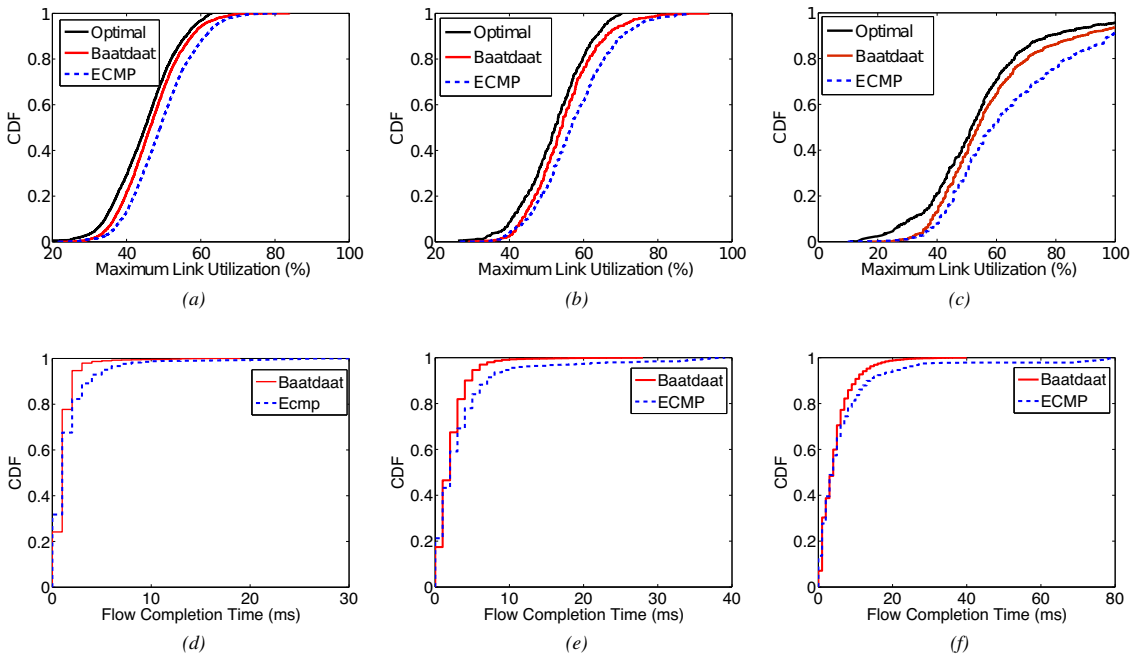

Fig. 5: CDF of maximum link utilization (MLU) and flow completion time for 4 KB, 8 KB and 100 KB flows respectively.

of a server's link capacity (as used in Hedera [6]), they should only be scheduled onto the least utilized shortest path since they can otherwise lead to congestion at multiple links. In this paper we stay focused on the small flows which are mostly found in the production data mining and Web service Cloud DCs [15]. The traffic is generated by sending these uniform flows to 100 servers at different racks and therefore create a high and unbalanced network load.

Optimal values used for benchmarking are computed based on the Penalizing Exponential Flow spliTting (PEFT) algorithm [8]. We first logged the traffic matrices (TMs) of every switching device in a simulated network, and then these TMs are used to compute optimal link weight for traffic splitting.

Fig. 5 shows the measured Maximum Link Utilization (MLU) for 4 KB, 8 KB and 100 KB flows for ECMP and *Baatdaat* approaches, respectively. The results demonstrate that *Baatdaat* consistently outperforms ECMP for all types of flows by up to 18%, and only deviates by 3% from optimal on average.

Looking more closely into the CDFs, the improvement for 4 KB flows is more uniform between the 30% – 70% of the MLU region. For 8 KB flows, the improvement is more significant around the 60% MLU region, and the improvement for 100 KB flows is more visible around the 70% MLU region. In these regions, *Baatdaat* only deviates 3% - 5% from the optimal, which demonstrates that the detour approach efficiently mitigates the increased congestion by offloading some flows onto less congested, albeit longer paths. This is an important feature since it offers an additional 10-18% headroom to DC providers using short-term traffic engineering to avoid resource outages due to temporal increases in traffic.

While DC RTTs can be as low as 250μs [15], can a slight detour of two more hops degrade individual flow completion time? We show the flow completion times for 4 KB, 8 KB and 100 KB flows for ECMP and *Baatdaat*, in Figures 5d, 5e, and 5f, respectively. While most traffic flows complete within 1 ms for 4 KB, 3 ms for 8 KB and 10 ms for 100 KB flows, we can also see that *Baatdaat* can significantly improve flow completion time, particularly between the 60th - 100th percentile, by as much as 41% to 95%. Obviously, this result demonstrates that allowing detours not only does not deteriorate but rather improves flow completion time. ECMP is slightly better than *Baatdaat* in the 30th percentile, due to small flows being more latency sensitive, and detour paths marginally increasing ($\leq$ 1 ms) latency. However, we can see in the 70th - 100th percentile, at least 10% flows complete faster in *Baatdaat* than they do in ECMP. Allowing larger number of flows to complete faster is an important feature for data center networks because many Web applications that run over these topologies require strict deadlines [15].

B. Impact of Measurement Intervals

Baatdaat uses temporal network load to schedule network flows, hence the link utilization measurement interval becomes crucial. In this experiment, we increased the measurement interval from 1 ms to 10 ms and 100 ms respectively, to study how it will affect *Baatdaat*'s performance.

From Fig. 6, it is apparent that the measurement frequency does not have an equal impact on every type of flow for MLU and flow completion time. The impact decreases with increments in the flow size. MLU for 4 KB flows drops dramatically when we increased the measurement interval

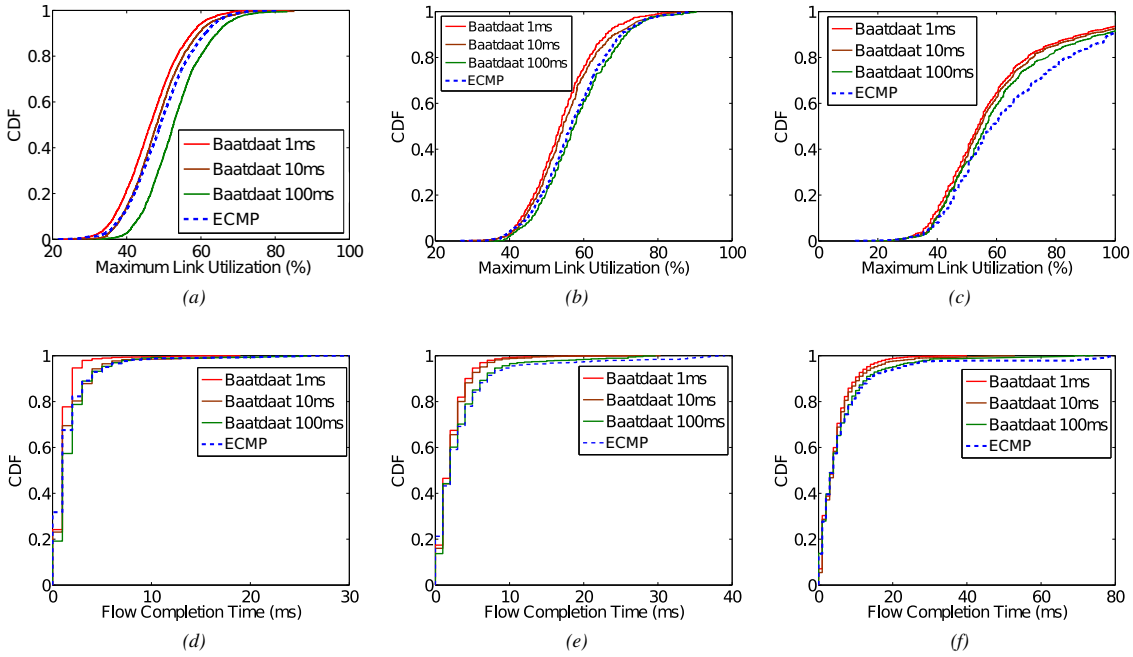

Fig. 6: CDF of maximum link utilization (MLU) and flow completion time for 4 KB, 8 KB and 100 KB flows with different measurement intervals respectively.

to 10 ms, as shown in Fig. 6a, with lower MLU region overlapping with ECMP and higher region exhibiting about 5% gain. Once we further increased the interval to 100 ms, the MLU curve shift further right giving a worse result than ECMP. In 8 KB and 100 KB flow scenarios, the 10 ms interval still gives comparable result to the 1 ms case. With a 100 ms interval, the CDF of the MLU curve tends to overlap with that of ECMP in the 8 KB flow scenario, while the 100 KB scenario achieves a significant performance gain of up to 10%. Similar behavior is also reflected in the flow completion time experiments.

In this experiment, we can see that traffic flows with smaller flow size are more vulnerable to sparse measurement intervals. In fact, this is related to the flow arrival interval. When the measurement interval in *Baatdaat* is increased, it becomes less responsive to transient changes in network load, so more incoming small flows may be agnostically placed on to the same path, and cause higher link utilization.

C. Impact of Longer Paths

We advocate using less congested albeit longer paths to alleviate congestion on the highly utilized links. However, one question that needs to be answered is *how much longer is the ideal for DC networks?* Due to the symmetric design of DC networks, any longer path between two communicating servers is a multiple of 2 hops longer than shortest ones. We have already shown earlier that a detour of two extra hops can indeed remedy congested link and improve flow completion time. Can a detour of four extra hops achieve better results given the even larger number of diverse paths it can allow?

In this experiment, we have set the detour path to four extra hops. The algorithm presented in Sec III also applies but with two more search depths. Fig. 7 shows our experimental results. The CDF of MLU curves in all scenarios constantly exhibit around 3-5% improvement over ECMP with only the lower region (i.e., MLU $\leq$ 20) being slightly worse than ECMP. This is due to detours becoming very unlikely as it is more difficult to opportunistically find a suitable path which satisfies the requirement *utilization of candidate path* $= max($*detour links*$)\times c$. However, when a detour does happen, the penalty it imposes on multiple links along the path is too significant to compensate the local gain. Flow completion time, as shown in Fig. 7d-7f, does not show significant improvement over ECMP.

On the other hand, when one considers longer paths, the complexity of the path searching algorithm needs to be accounted for as well. Not only do longer detour paths not give any obvious performance gain, but the path finding complexity also increases quickly as the path length grows. Thus, we suggest that only detours of two extra hops should be considered.

D. Testbed Experimental Results

We have implemented *Baatdaat* and evaluated it on a testbed consisting of six OpenFlow switches, an OpenFlow controller and 4 server hosts, as shown in Fig. 8.

The testbed is formed to evaluate *Badtdaat* performance on canonical tree topologies (as opposed to the fat-tree topology used in the simulation). Two of the switches are aggregation switches and the remaining are ToR switches, each connecting to one host machine. The two aggregation switches communicate with the controller through out-of-band connections

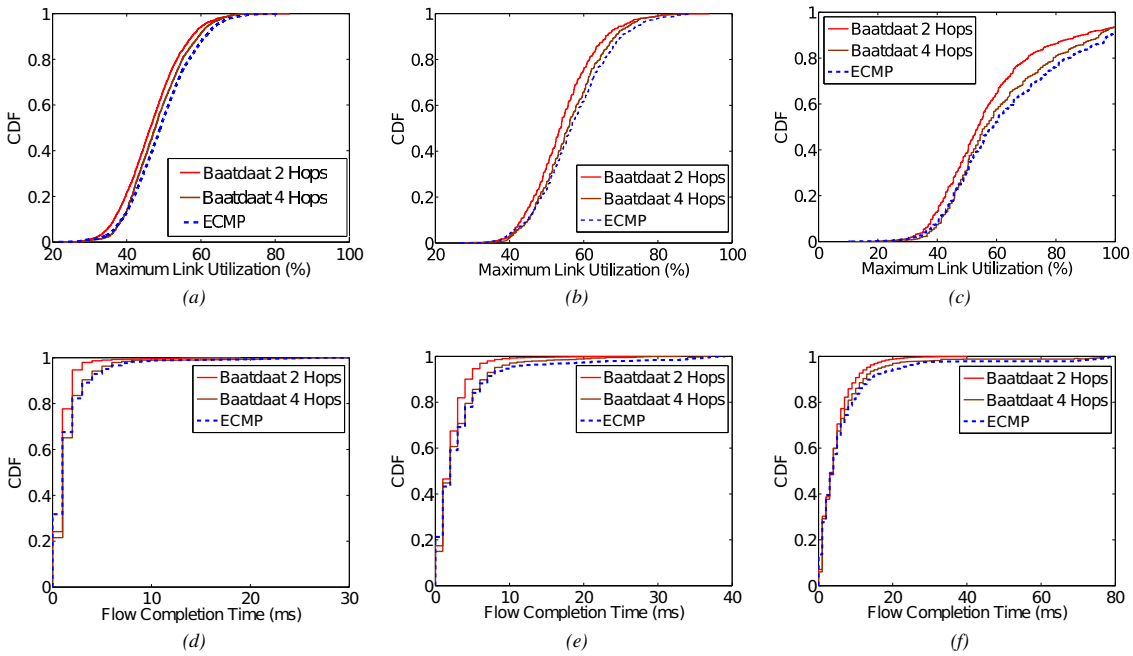

Fig. 7: CDF of maximum link utilization (MLU) and flow completion time for 4 KB, 8 KB and 100 KB flows with different path lengths respectively.

TABLE III: Baatdaat/ECMP MLU ratio at 25th, 50th, 75th and 100th percentile for fat-tree and canonical tree topologies

Flow Type	Topology	25th	50th	75th	100th
4 KB	Fat-tree	0.9463	0.9433	0.9456	0.9807
	Canonical	0.9258	0.9470	0.9334	0.9324
8 KB	Fat-tree	0.9557	0.9468	0.9408	0.9906
	Canonical	0.9675	0.9485	0.9469	0.9875
100 KB	Fat-tree	0.9333	0.9123	0.8383	0.9407
	Canonical	0.9749	0.9676	0.9706	0.9972

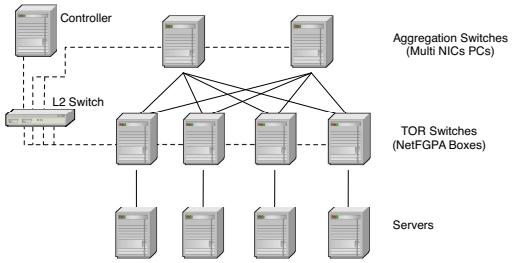

Fig. 8: Testbed topology

(shown as dash lines in Fig. 8). The out-of-band connections are only for control data among the switches and the controller, and do not carry any data traffic. To accurately capture DC network characteristics, we also need to produce a certain level of network oversubscription. To achieve this we throttled the link capacity among switches from 1 Gb/s to 100 Mb/s and have link capacity between switches and hosts , yielding an oversubscription ratio of $1:5$.

Table III tabulates our testbed results of Baatdaat v.s. ECMP MLU ratio at 25th, 50th, 75th and 100th percentile. It can be seen that *Baatdaat* has consistent gain of up to 7% over ECMP, over the canonical tree testbed. The performance gain, however, is less than that over a fat-tree topology due to the reduced path diversity that can be exploited. Nevertheless, the testbed results show that *Baatdaat* is a topology-neutral scheduling algorithm that works over both fat-tree and canonical tree topologies.

E. Hardware Implementation Complexity

We have analyzed the hardware synthesis reports and made a comparison between the original and our modified OpenFlow implementation on NetFPGA. The result is shown in Table IV.

On the NetFPGA a Xilinx Virtex 2 Pro 50 chip is used which consists of 5,904 CLBs (Configurable Logic Blocks). A CLB consists of four *Slices*, and a Slice consists of two LUTs (Look Up Tables) and two Flip Flops. Hence, the chip has 47,232 (5,904×4×2) Flip Flops and 47,232 (5,904×4×2) LUTs in total. In *Baatdaat*'s version of OpenFlow presented in Sec II, we have not only implemented a new link utilization measurement on top of the reference implementation, but also modified the flow processing pipeline to provide for multipathing support. Additionally, a divider generator is used to calculate the link utilization, resulting in consuming additional logic. However, the extra number of flip flops as well as the LUTs are still kept under 20%, hence the additional complexity is minimal. Moreover, evaluation results show that our implementation does not degrade the NetFPGA's throughput performance.

TABLE IV: NetFPGA implementation complexity

	Component	Ref Design	Baatdaat	Difference
Slice Logic Utilization	Slice Flip Flops	43%	61%	18%
	Slice LUTs	72%	91%	19%
	Slices LUTs as logic	56%	75%	19%
Memory Utilization	Slice LUTs as Dual Port RAMs	7%	7%	0%
	Slice LUTs as 32x1 RAMS	0.5%	0.5%	0%
	Slice LUTS as 16x1 RAMs	0.3%	0.3%	0%
	RAMB16s	65%	65%	0%
I/O Utilization	Number of bonded IOBs	51%	51%	0%
Specific Feature Utilization	Number of BUFGMUXs	50%	50%	0%
	Number of DCMs	75%	75%	0%

TABLE V: Controller Performance

$k =$	4	8	16	24	32	48
Single Iteration (ms)	0.005	0.012	0.021	0.041	0.063	0.122
Number of Aggregation Switch	8	32	128	288	512	1152
Single Iteration (ms) for All Aggregation Switches	1.53	6.47	23.2	53.2	95.8	220.3

F. Controller Efficiency

Next, we test the *Baatdaat* controller's efficiency. As presented in Sec II-A, only aggregation switches connect to the controller. The controller determines detour paths for the aggregation switches by executing the *Baatdaat* scheduling algorithm presented in Sec III, rather than dealing with each flow individually. So a simple question like "how many flows the controller can handle at a time?" cannot sufficiently capture *Baatdaat's* scheduling characteristics. The more important question here is "How many aggregation switches can the controller support at a time with minimal computational latency?".

The controller runs on a machine with an Intel i5 2.3Ghz CPU and 4GB RAM. Another identical machine generated connections and transmitted link utilization packets to the controller. We first ran the test on the controller to find the baseline time the controller takes to search through all possibilities in one iteration, for one aggregation switch and various *k-port* switches. We then run tests by connecting all aggregation switches to the controller and for various *k-port* fat-trees. The test result is tabulated in Table V. Clearly, we see that the algorithm is very efficient as it only varies from 5 μs to 122 μs for single iteration (finished executing path searching once) which is linear with a growing number of k from $k = 4$ to $k = 48$. And with such large k values, the number of network-wide aggregation switches grows from 32 to 1,152. Again, the time to finish one iteration for all aggregation switches grows linearly from 1.53 ms to 220.3 ms. Obviously, the test result reflects that even with a large fat-tree network ($k = 48$), one controller can seamlessly serve all aggregation switches with remarkably fast response times.

G. Baatdaat Overhead

As a load-adaptive scheduler, *Baatdaat* comes with certain level of control overhead, and extra hardware component cost. We have shown that the hardware measurement module can be easily implemented as part of the NetFPGA's OpenFlow switch module with only a little increment in the logic component used. We believe this would be an inexpensive module to build in otherwise commodity OpenFlow switches. In addition, *Baatdaat* requires a flow table to store path information for the active flows, and a forwarding table for multipathing. Nevertheless, as mentioned above, *Baatdaat* can exploit Openflow's existing exact match (i.e., same piece of memory with the exact match table) and wirldcard tables for storing flow and forwarding information, respectively, and thus incurring no extra memory cost. For example, NEC PF5240 Openflow switch can store up to 160,000 12-tuple flow entries. Furthermore, the control message overhead due to link utilization being reported to the controller can be configured at an appropriate temporal interval. Assuming a frequent 1 ms interval, the control message overhead would be attributed to $8 \times 48 = 384$ Bytes (assuming a 48 port switch) which is equivalent to only 384,000 Bytes per second per aggregate switch reporting to the controller. This is a marginal cost, compared to the potential revenue return *Baatdaat* can bring to operators through increasing DC's available capacity and improving flow completion times.

V. Related Work

Traffic Engineering and routing techniques have been widely used for Internet topologies [17][18][19] that typically operate on predictable aggregate traffic matrices.

Within DC networks, typically ECMP [3] is used to exploit path diversity and distribute load among redundant shortest paths. However, ECMP suffers from hash collisions, which can result in unbalanced flow allocations across paths. VL2 [4] is a virtual layer 2 infrastructure which uses Valiant Load Balancing (VLB) to randomize packet forwarding. Monsoon [20] is a mesh-like DC architecture that also incorporates VLB to allow for traffic balancing over a large layer 2 mesh for any traffic patterns. However, VLB exhibits similar limitations to those of ECMP [6]. Hedera complements ECMP by identifying large flows exceeding 100 Mb/s at network edge switches and then scheduling these flows over a redundant path with suitable capacity [6]. However, this approach becomes limited as network utilization increases and flows fair-share the bottleneck bandwidth. On the contrary, our work attempts to place flows based on minimum link utilization and independent of flow size. DeTail [15] attempts to reduce flow completion time in DC networks for web site load times by using flow priorities

and switch port buffer occupancies to determine next hop behavior. However, it requires major changes in several layers of the networking stack and applications must be modified to communicate flow priorities for latency-sensitive traffic scheduling. *Baatdaat* is much simpler to deploy, requiring no in-depth knowledge of individual application flows, yet still significantly reducing flow completion time in congested networks. MicroTE [5] uses short-term traffic patterns and partial predictability to achieve its goals. However, with highly unpredictable DC traffic, it becomes equivalent to, or worse than, ECMP, and requires significant changes to end hosts as traffic monitoring duties are pushed to servers.

Other flow scheduling schemes for DCs, such as DCell [21] and SPAIN [22], use servers to determine packet routing or relaying, which requires modification to end hosts. As well as using the servers to choose packet paths, SPAIN also uses pre-computed traffic traces that exploit redundancies in topologies, meaning that, unlike *Baatdaat*, it cannot act upon changes in network utilization. Modifications to end-hosts are required, unlike *Baatdaat* which only requires modifications to a limited number of switches and the central controller, making it an almost unfeasible task in DCs with tens of thousands of servers to update and maintain. DevoFlow [23] aims to replace the OpenFlow model by reducing control-plane communication, moving the responsibility for handling most flows to switches and gathering statistics to identify large flows that are then handled and scheduled via the controller. Through simulations they claim to be able to efficiently load-balance data center traffic. However, we have shown that we are able to gather statistics at the level of per-link, rather than per-flow, and continuously share these with the OpenFlow controller to achieve load balancing with minimal performance impact at the controller.

VI. Conclusion

In this paper we have presented *Baatdaat*, a novel flow scheduling system for DC networks. *Baatdaat* uses a modified NetFPGA implementation of OpenFlow to directly measure the temporal, network-wide utilization of the infrastructure and to subsequently schedule flows over redundant lightly-utilized shortest and non-shortest paths to further exploit topological redundancy. Unlike existing flow scheduling algorithms, *Baatdaat* takes into consideration the current state of the network to dynamically adapt flow scheduling using a combination of centralized and distributed logic as well as hardware acceleration to avoid performance bottlenecks.

Simulation and testbed results have shown that *Baatdaat* is topology-neutral and can substantially improve network-wide utilization while also reducing flow completion times, demonstrating that avoiding congestion pays off for the slightly longer paths taken by a subset of flows. *Baatdaat* reduces network-wide maximum link utilization by up to 18% over ECMP and only deviates by 3% from optimal on average, while it can reduce flow completion times by up to 10%. Results on the NetFPGA platform demonstrate that traffic monitoring and real-time flow-level scheduling can be seamlessly incorporated into networks with virtually no impact on network throughput.

References

[1] S. Kandula, S. Sengupta, A. Greenberg, P. Patel, and R. Chaiken, "The nature of datacenter traffic: Measurements & analysis," *Internet Measurement Conference (IMC)*, 2009.

[2] T. Benson, A. Akella, and D. Maltz, "Network traffic characteristics of data centers in the wild," *Internet Measurement Conference (IMC)*, 2010.

[3] C. Hopps, "Analysis of an equal-cost multi-path algorithm," *RFC 2992, IETF*, 2000.

[4] A. Greenberg, J. Hamilton, N. Jain, S. Kandula, C. Kim, P. Lahiri, D. A. Maltz, P. Patel, and S. Sengupta, "VL2: A scalable and flexible data center network," *ACM SIGCOMM*, 2009.

[5] T. Benson, A. Anand, A. Akella, and M. Zhang, "MicroTE: Fine grained traffic engineering for data centers," *ACM CoNEXT*, 2011.

[6] M. Al-Fares, S. Radhakrishnan, B. Raghavan, N. Huang, and A. Vahdat, "Hedera: Dynamic flow scheduling for data center networks," *The 7th USENIX Symposium on Networked Systems Design and Implementation (NSDI '10)*, 2010.

[7] N. McKeown, T. Anderson, H. Balakrishnan, G. Parulkar, L. Peterson, J. Rexford, S. Shenker, and J. Turner, "OpenFlow: enabling innovation in campus networks," *SIGCOMM Comput. Commun. Rev.*, vol. 38, no. 2, 2008.

[8] D. Xu, M. Chiang, and J. Rexford, "Link-state routing with hop-by-hop forwarding can achieve optimal traffic engineering," *IEEE/ACM Transactions on Networking*, April 2011.

[9] NetFPGA, "http://www.netfpga.org/."

[10] A. Tavakoli, M. Casado, T. Koponen, and S. Shenker, "Applying nox to the datacenter," *ACM HotNets*, 2009.

[11] Cisco Systems, "Data center: Load balancing data center services solutions reference nework design march," 2004.

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

[13] "Open vSwitch: An open virtual switch," *http://openvswitch.org/*.

[14] J. Naous, D. Erickson, G. A. Covington, G. Appenzeller, and N. McKeown, "Implementing an OpenFlow switch on the NetFPGA platform," *ACM/IEEE Symp. on Architectures for Networking and Communications Systems (ANCS)*, 2008.

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

[16] S. Kandula, S. Sengupta, A. Greenberg, P. Patel, and R. Chaiken, "The nature of data center traffic: measurements & analysis," in *Proc. ACM SIGCOMM Internet Measurement Conference (IMC'09)*, 2009, pp. 202–208.

[17] A. Elwalid, C. Jin, S. Low, and I. Widjaja, "Mate: Mpls adaptive traffic engineering," *IEEE INFOCOM*, 2001.

[18] H. Wang, H. Xie, L. Qiu, R. Yang, Y. Zhang, and A. Greenberg, "Cope: Traffic engineering in dynamic networks," *ACM SIGCOMM*, 2006.

[19] S. Kandula, D. Katabi, B. Davie, and A. Charny, "Walking the tightrope: Responsive yet stable traffic engineering," *ACM SIGCOMM*, 2005.

[20] A. Greenberg, P. Lahiri, D. A. Maltz, P. Patel, and S. Sengupta, "Towards a next generation data center architecture: scalability and commoditization," *Proc. ACM workshop on Programmable routers for extensible services of tomorrow (PRESTO)*, 2008.

[21] C. Guo, H. Wu, K. Tan, L. Shi, Y. Zhang, and S. Lu, "DCell: A scalable and fault-tolerant network structure for data centers," *ACM SIGCOMM*, 2008.

[22] J. Mudigonda, P. Yalagandula, M. Al-Fares, and J. C. Mogul, "SPAIN: COTS data-center ethernet for multipathing over arbitrary topologies," *NSDI*, 2010.

[23] A. Curtis, J. Mogul, J. Tourrilhes, P. Yalagandula, P. Sharma, and B. S., "Devoflow: Scaling flow management for high-performance networks," *ACM SIGCOMM*, 2011.