Master Thesis

Static Yet Flexible: Expander Data Center Network Fabrics

Author(s):
Kassing, Simon Arnold

Publication Date:
2017

Permanent Link:

https://doi.org/10.3929/ethz-a-010890129 →

Master Thesis

Static Yet Flexible: Expander Data Center Network Fabrics

Author(s):
Kassing, Simon Arnold

Publication Date:
2017

Permanent Link:

https://doi.org/10.3929/ethz-a-010890129 →

ETH
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Static Yet Flexible: Expander Data Center Network Fabrics

Master Thesis

S.A. Kassing

March 17, 2017

Supervisor: Prof. Dr. A. Singla
Department of Computer Science, ETH Zürich

Abstract

Recent work has indicated that any static data center network is fundamentally limited, due to its inability to move around network capacity. Is this truly the case, is the static network not flexible enough to handle varying (skewed) traffic scenarios through only traffic engineering? Can we only find refuge in dynamic topologies, introducing on-the-fly re-arrangement of network links at a cost? In pursuit of these research goals, three main data center topologies were evaluated: traditional (oversubscribed) fat-trees, expanders and dynamic topologies. Using flow optimality evaluation via a linear program, worst case traffic scenarios were identified and their respective performance measured under perfect traffic engineering. A custom discrete packet simulator was used to evaluate the two static topologies under a wide range of traffic scenarios and compare results to performance claims of recent dynamic topology research. It was found that the modeling of server up-links is crucial to a meaningful comparison. Equivalent performance to that of the most recent dynamic topology was achieved. At 67.5-80% of its cost, the expander was able to rival the fat-tree in all traffic scenarios using at least one of its routing strategies. Three factors had significant impact in this: (a) the high ToR-ratio (per-ToR network up-links divided by per-ToR server up-links), (b) the low average shortest path length, and (c) the property that every cut in the network is traversed by many links. It was found that oversubscription of the fat-tree indeed creates problematic traffic scenarios, where communication between only a small fraction of active servers is severely impaired. The work raises new potential inquiries pertaining to the vast space of routing logic and their parametrization, engineering challenges in the deployment of expanders, exploration and topology-independent characterization of workload in the data center, further verification through use of other simulators or physical deployment, and theoretical understanding of the capabilities of dynamic topologies.

Contents

Chapter 1

Introduction

Data center infrastructure is the backbone of virtually all Web services. With the growing demand for these services, the supporting infrastructure needs to scale correspondingly. Engineering full-bandwidth connectivity between all pairs of servers would be costly for such large facilities. Moreover, there is significant indication only a small fraction of servers require high-bandwidth communication [10, 12] at any given time, making such design seem wasteful. Traditional tree-like topologies such as the fat-tree [1], only offer two choices: (a) a costly full-bandwidth network, or (b) a cheaper oversubscribed network. The oversubscribed network would result in not providing high-bandwidth connectivity even to a small subset of servers [24].

Recent work, in response to these general concerns, has proposed the change to dynamic links, under the premise that static topologies are fundamentally limited due to its network capacity being fixed. The static network is supposedly unable to handle the skewed workloads because it cannot move network capacity around. The proposals present a way to dynamically move around network capacity between traffic demanding nodes at run-time. These schemes depend for example on well established technologies such as 60 GHz wireless technology [12], or on "radical departures" by proposing completely new forms such as [10] using a mirror construction and lasers to dynamically reconfigure links.

The basic premise advocated by these dynamic topologies, that static topologies are inherently limited and unable to address concerns, is primarily based on observations of traditional tree-like static topologies. Static topologies can find their refuge in their only method to address traffic: routing strategies, and are aided by the lower cost of static equipment versus specialized dynamic equipment. Recent work has shown significant promise in the employment of random regular graphs [26] and Xpanders [27], both of which are *expanders* [27]. Topologies in this class show promise primarily due to their low average shortest path length [25] and the property of hav-

ing every cut traversed by many links [27]. Due to these favorable properties, expanders offer an interesting proposition: is it possible to achieve performance rivaling that of traditional tree-like structures and even dynamic topologies?

All these proposals of data center topologies require a thorough analysis to be better understood and compared. The work is structured as follows. First, in chapter 2, the topologies in question are explained elaborately. Second, in chapter 3, a flow evaluation on engineered worst case traffic scenarios is performed using optimization via a linear program, achieving perfect routing. Third, in chapter 4, a discrete packet simulator is used to evaluate the static topologies and compare their results with one-another and performance claims made by the most recent dynamic topology paper. Fourth, in chapter 5, discussion of the work takes place. Fifth, in chapter 6, possible future work is proposed. Finally, in chapter 7, the thesis is concluded.

Chapter 2

Topologies

The three main topologies approaches of this thesis are described in this chapter. First, in section 2.1, the fat-tree topology [1] is described, a traditional tree-like data center architecture. Second, in section 2.2, the class of *expander* graphs are discussed, in particular Jellyfish [26] and Xpander [27]. Third, in section 2.3, dynamic topologies are discussed, focusing on analyzing their potential and limitations.

2.1 Fat-tree

The fat-tree has roots from a paper about supercomputer architecture by [18], but has since been adopted in the context of data center network architecture [1]. The architecture differentiates between its internal nodes, which are solely switches, and its leaf nodes, which are ToRs. The design consists of three layers: the core layer, the middle ('aggregation') layer and the lower ('edge') layer. The lower and middle layer are horizontally separated in pods.

The entire topology is determined by even integer parameter k, naming the resulting topology unambiguously a k-fat-tree. Every switch has a degree of k. There are k pods of which in each k switches reside. In each pod every lower switch is connected to every middle layer switch in its pod. This results in $k/2$ upward network links per lower switch and $k/2$ downward network links per aggregation switch. There are a total of $k^2/4$ core switches. Every middle layer switch in a pod is connected to $k/2$ core switches. A diagram of a fat-tree with $k = 4$ is shown in figure 2.1.

In a fat-tree, there are always k shortest paths between any pair of ToRs. Routing mechanisms such as ECMP (see section 4.1.3) are highly effective in this uniform environment, enabling simple next-hop forward tables kept in each switch to be sufficient to enable good load balancing.

The full fat-tree, which employs a full non-blocking interconnect fabric, becomes quite expensive as it scales with a large k, requiring $5/4 \cdot k^2$ switches, $1/2 \cdot k^3$ bi-directional inter-switch network links and $1/4 \cdot k^3$ bi-directional switch-to-server network links. Typically, an oversubscribed fat-tree is used in practice, with less switches while still supporting the same amount of servers. The calculation to determine the performance of an oversubscribed fat-tree is described in appendix B.

The critique [24] lies in the poor performance in skewed traffic scenarios of these oversubscribed fat-trees, which are commonly deployed to reduce costs. Suppose that a fat-tree is built at β% capacity (e.g. by removing β% of the core switches). Even a traffic load involving only two pods, as such only $2/k$ of the servers are involved, can then merely communicate at β% of the full throughput (as all traffic is directed through core switches). The rest of network capacity, which is "invested" in the connectivity of other pods, is kept idle. An example of this poor performance is demonstrated in figure 2.2 for a $k = 4$ fat-tree.

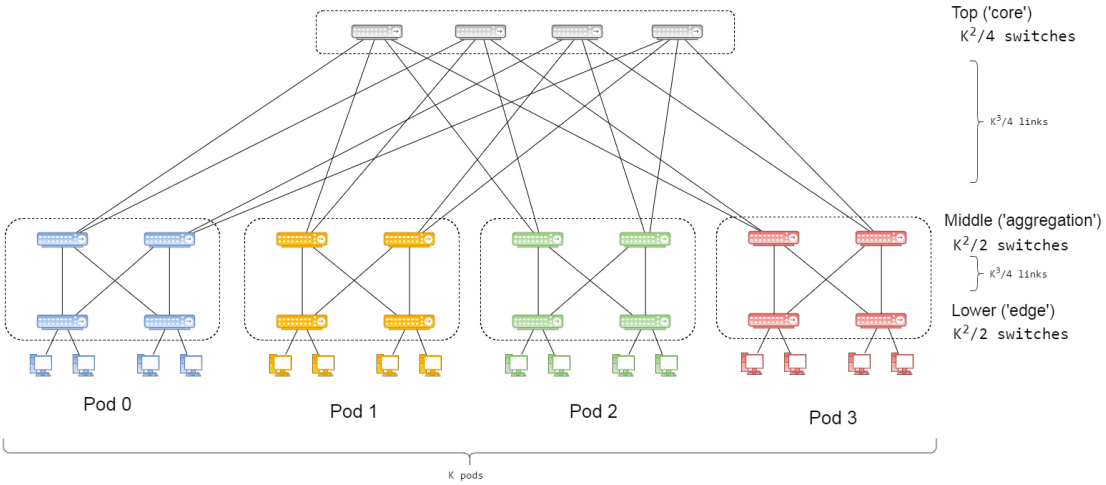


Figure 2.1: Fat-tree topology with $k = 4$

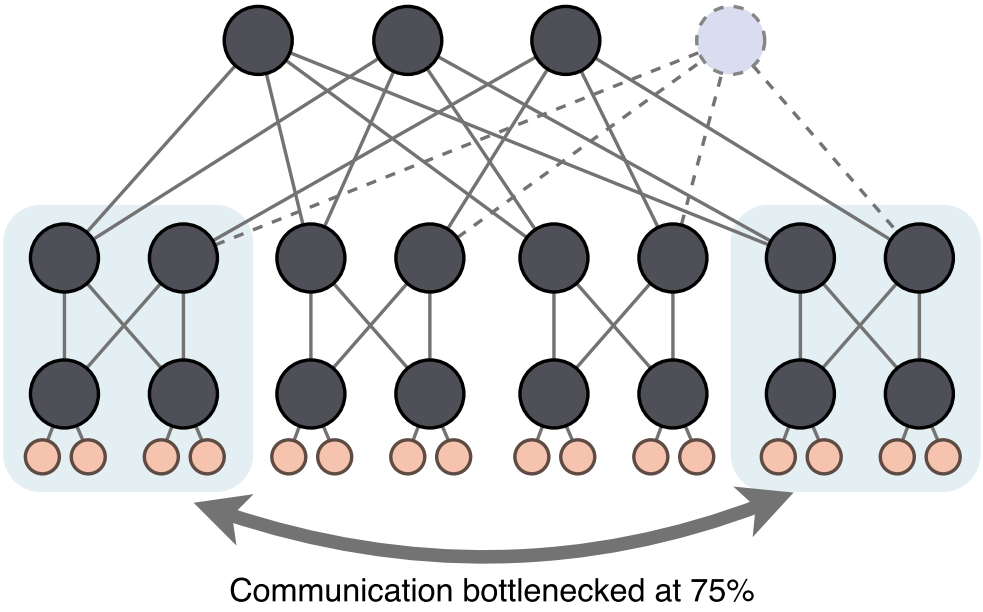


Figure 2.2: A 75%-oversubscribed $k = 4$ fat-tree is only able to provide 75% of throughput when a mere 50% of servers (in two pods) is active.

2.2 Expanders

Valdarsky et al. [27] unveiled the potential that expanders graphs have for data center networks. From the mathematical foundation follows that every cut in the network is traversed by many links. When this is the case for a minimal cut, it means that traffic between two nodes, one on either side, is less bottlenecked by the network. Moreover, this provides high resiliency to network failure: if a single link fails, many paths still exist to fall back on. Interestingly, random graphs (such as Jellyfish [26]) are shown to be "*remarkably good expander graphs*". Valdarsky et al. [27] argue that the inherent unstructured state of random graphs makes them hard to reason about (to achieve guarantees) and to build (due to wiring complexity). They propose their algorithm, Xpander, which deterministically builds an expander graph.

The challenge and opportunity of expander graphs lie in the diversity of paths. Like the fat-tree, there are many paths between two ToRs. The selection of viable routing paths however is not as clearly defined as for a fat-tree. In a fat-tree, only shortest path routing is required and will yield the most viable paths. This is achieved through abundant addition of routing switches providing the tree-like infrastructure. In an expander on the other hand, selection of only shortest paths would omit many of the viable paths present. It is desirable to create a mechanism that is able to identify a set of viable paths of varying length in an expander. Techniques such as K-shortest paths, valiant load balancing, or other K-path selection techniques are proposed to overcome this hurdle.

In the next two sections, the generation of the two respective expander graphs are examined.

2.2.1 Jellyfish

The Jellyfish topology [26] is a random graph. The generation procedure is described in pseudo-code in algorithm 1. It can be run multiple times to filter out 'bad results' such as when it is impossible to fulfill all links because by chance a clique of remaining nodes has formed with free link ports. Furthermore, the spectral gap can be used to filter out random graphs that do not display nice expander properties [27].

2.2.2 Xpander

The Xpander topology [27] is based on the concept of lifting a graph. This is a method of enlarging an already existing expander graph. It is done the following: create k duplicates of every $v \in V$ namely $v_1, ..., v_k$ and then for every $e = (u, v) \in E$ create a k-matching between every the two sets of duplicates $u_{1...k}$ and $v_{1...k}$. Naturally, there are many k-matchings possible. A k-lift of a graph $G = (V, E)$ results in a graph $G = (V', E')$ with $|V'| = k \cdot |V|$ and

$|E'| = k \cdot |E|$. Valdarsky et al. [27] have shown that this random procedure can be deterministically done, moreover allowing incremental expansion via a heuristic that preserves expander properties. The spectral gap is used as measure to determine edge expansion.

Construction of an Xpander

1. Start with complete d-regular graph on $d+1$ vertices (a fully connected graph of $d+1$ vertices).
2. Lift the graph repeatedly until desired lower bound number of vertices is reached.
3. Incrementally expand the graph till number of vertices is reached.

Algorithm 1 Generating d-regular Jellyfish topology

Require: V : *set of n nodes*
$V_{left} \leftarrow \{1, ..., n\}$
$D_{used}[i] \leftarrow 0, \ \forall\, i \in \{1, ..., n\}$
while $|V_{left}| > 0$ **do**
    if link possibility exists in V_{left} **then**
        Add link between two random nodes $v_1, v_2 \in V_{left}$ w/o link
        $D_{used}[v_1]$++
        $D_{used}[v_2]$++
        if $D_{used}[v_1] \geq d$ **then**
            $V_{left} = V_{left} - \{v_1\}$
        end if
        if $D_{used}[v_2] \geq d$ **then**
            $V_{left} = V_{left} - \{v_2\}$
        end if
    else ▷ Clique exists or single node remains
        break
    end if
end while

2.3 Dynamic Topologies

Recent work [10, 12, 13] has shown increasing interest in topologies that can dynamically configure the links between ToRs. The idea behind this is that static topologies are inherently flawed, forcing link capacity to be wasted in skewed traffic scenarios. Dynamic topologies on the other hand, following this reasoning, can move capacity around in an on-line fashion, enabling optimal usage of the available resources.

Of course, if dynamic ports are of equal cost as static ports, this comparison is easily finished: the dynamic topology can at least do everything a static topology can. This simple statement is however not reflective of reality, in which dynamic ports are more expensive. Furthermore, there are additional cost-inducing factors such as maintenance (e.g. due to sensitivity of ports) and spatial requirements (e.g. lay-out of the data center). In this thesis we will simplify this by stating that a dynamic port is a factor of δ more expensive than a static port. Because dynamic ports are establish in an on-line fashion, buffering of flows will be needed for ones that do not have access to a route and need to wait for one to be created. A favorable estimation of buffering requirements in an all-to-all traffic scenario is made in appendix A. It shows that buffering and its effect on flows is far from a trivially solvable problem.

The main challenge in these dynamic topologies is the decision which switches to connect (match) with each other, on top of the routing challenge which static networks face as well. The approaches favor short as possible paths between the active nodes, preventing waste of network capacity by multi-hop routing.

ProjecToR [10] is the most recent in a series of dynamic topology works. The authors claim that there are three desirable properties for reconfigurable networks: *"1) Seamlessness: few limits on how much network capacity can be dynamically added between ToRs; 2) High fan-out: direct communication from a rack to many others; and 3) Agility: low reconfiguration time"*. It motivates these respectively with skewed traffic matrices, capacity consumption of multi-hops and the existence of sources sending lots of data to many destinations. ProjecToR realizes a switching time of $12\mu s$, followed by a transmission period of $120\mu s$. The numbers they provide estimate its dynamic ports to be a δ factor of $\sim$1.5x (low ball) to $\sim$2.0x (high ball) more expensive than a static port.

Chapter 3

Flow Evaluation

To determine the potential of each static network, under perfect network flow routing, a flow model is employed. First, in section 3.1, the need for flow-level evaluation is discussed and the accompanying metrics. Second, in section 3.2, the flow (linear program) model and its implementation as the TopoBench framework is described. Third, in section 3.3, the results of the flow evaluation are discussed.

3.1 Flow Metric Establishment

The goal of flow evaluation is to be able to compare network topologies without having to worry about routing strategies: the optimization of flows guarantees *perfect* traffic engineering. The complication is moved to the ability of translating the performance achieved by this perfect traffic engineering into a (distributed) network and transport protocol, which is attempted for the expander in chapter 4. In the flow evaluation, it is necessary to define a metric that is able to objectively compare the topologies. The challenge is that choosing a single or a set of independently "drawn" traffic scenario to compare topologies with, would favor the topology which is constructed to specifically handle those drawn traffic scenarios. It is thus needed to construct the traffic scenario *based on* each topology. Specifically, it would be useful to identify and apply the worst case traffic scenario for each topology.

Finding these worst-case traffic scenarios is not undemanding. [24] puts it concisely: "*finding the worst-case traffic model is a computationally non-trivial problem the complexity of which is not well understood, we make our best efforts to evaluate statically-wired networks under difficult TMs*". Several heuristic models are commonly used to evaluate network performance, typically aimed at providing a skewed TM that forces flow bottlenecks to form, even under perfect traffic engineering (TE). All traffic flow is modeled under the hose traffic model. First, in section 3.1.1, the idealistic concept of throughput

proportionality is introduced. In the following sections, the worst case traffic is identified for all three classes of topologies under investigation: fat-tree (section 3.1.2), expanders (section 3.1.3), and dynamic topologies (section 3.1.4).

3.1.1 Throughput proportionality

Throughput proportionality (TP) is an idealistic goal for a network topology [24]. It stems from the desired ability of network flexibility. It puts into metric the desire to evaluate to which extent it is possible to put all available network capacity to use using the best possible TE, regardless whether many servers are active, or only few.

Throughput proportionality is defined as follows:

1. Given a network topology G.
2. Identify an as-difficult as possible traffic scenario, with correspondingly named set of traffic matrices TM_{hard} (see following sections for examples). These traffic matrices could be formed from a base traffic matrix $TM_{hard\ base}$. The traffic scenario should be able to schedule different server fractions with traffic demand, and difficulty should decrease with the server fraction active for this adaptable traffic scenario.
3. Under the assumption of perfect TE, calculate the maximum α for which the traffic matrix $\alpha \cdot TM_{hard}$ is satisfiable under the hose traffic model for an active fraction of 1.0 (typically $TM_{hard\ base}$). This is done via a linear program (see section 3.2).
4. The α is then used in the following idealistic scenario: a network is throughput proportional when for any hose-traffic-abiding traffic matrix involving any fraction $x \leq \alpha$ of servers it achieves a full throughput of 1 for each of the servers involved. In the interval of $[\alpha, 1]$ it should achieve a throughput per server of α / x.
5. To evaluate whether a network is throughput proportional, a plot should be made laying the throughput per server against the fraction x of servers with traffic demand. This could for example be done by for $1 - x$ servers setting the row/column entries to zero in a base $TM_{hard\ base}$, producing a traffic matrix $\in TM_{hard}$. The plot should then be compared to the in step 4 defined TP function.

3.1.2 Fat-tree: Inter-pod All-to-All

A fat-tree is a completely balanced topology, forcing every inter-pod communication to go upwards to the core layer over k possible shortest paths before being sent downwards over a single possible shortest path. As such,

for a full fat-tree, which is fully non-blocking, there is no traffic matrix which is worst-case for the concept of proportional throughput per server. However, this changes for the cost-reducing oversubscribed fat-tree. Under the assumption that β% core switches are removed to oversubscribe a full fat-tree, only a throughput of $(100 - \beta)$% is possible for a full pod-to-pod communication of servers. This is reflected in the calculation in appendix B, which forces the best possible β% oversubscription to support the number of servers given the limited number of available switches.

3.1.3 Expanders: Maximum Weight / Minimum Weight / Random Pairs

Expanders, as a subset of well-behaving random graphs [27], achieve a near-optimal node throughput for any static network under uniform traffic matrices of $r/\langle D \rangle$ due to its near-optimal short average node-to-node distance [25]. In this case r is the network degree, and $\langle D \rangle$ the average path length. This follows from the fact that the r-regular network has a total network capacity of $n \cdot r$.

The most challenging traffic scenario, under perfect traffic engineering, would thus be one that forces as high as possible path lengths [24]. These long paths would then force usage of network capacity. Moreover, large rack-to-rack will reduce the number of load-balancing opportunities [24]. A maximum-weight matching is used to select pairs, where the weights are the shortest path distances between the nodes. In a uniform weight network, the maximum-weight matching will result in pairing nodes with each other. Truncating the initial full set to traffic fraction x (to obtain set TM_{hard}) is done by (a) going from top and selecting nodes until have reached x, and (b) doing a maximum matching amongst those nodes.

Under less than perfect traffic engineering, not all possible path diversity in the graph is utilized. Thus, a traffic scenario in which it is difficult to find the path diversity required to prevent traffic being bottlenecked is very difficult. As a way to show this, a minimum-weight matching is used to select pairs. This will result in mostly pairs which are neighbors, until this is no longer feasible in the graph. Another possible difficult scenario is in which a single bottleneck is identified over which two pairs go, thus forcing both flows to share the link. Random permutation pairs can be a solution to, by chance, show such an adverse traffic scenario.

3.1.4 Dynamic Topologies: Undefined

For dynamic topologies it is difficult to define what exactly is the most challenging traffic scenario due to a lack of theoretical foundation on its capabil-

ities and limitations. There are two models for a dynamic topology that are useful to assess its capabilities:

- **Zero ("perfect"):** under perfect traffic engineering and perfect on-line matching, with zero switching time. It is possible to achieve, for a r-regular graph, a complete node throughput of r by directly coupling itself to the destination. Once a flow is over, it can instantaneously switch to the next using its out-degree of r. There is no traffic worst case traffic scenario. This *perfect* model can be used as a strict upper bound on dynamic topologies;
- **Non-zero non-buffered ("restricted"):** under perfect traffic engineering but with non-zero switching time without allowing buffering of flows [24]. This means that it achieves perfect utilization of the network, as such it is the perfect upper bound described in [25] making effective use of $1/\langle D \rangle$ of the network capacity of $n \cdot r$ in an all-to-all traffic scenario. Under other traffic scenarios, it is not yet known what the best-possible static topology would be; for an all-to-all-α-fraction traffic scenario an upper bound calculated via [25] with $n = \alpha \cdot n_{original}$ is the best-known effort. This can be used as an best-known estimate loose lower bound on dynamic topologies.

3.2 TopoBench: Static Network Flow Evaluator

In order to perform the flow evaluation, a continuation project was created building upon the work created by Jyothi et al. [15]: the TopoBench framework. The continuation included additional documentation of the code, significant architectural changes with modularization as its primary objective, and extension of it with additional traffic modes and a path evaluator.

The TopoBench framework is based on a flow model. The solution of this model forces that all flows in the traffic matrix TM are treated linearly equally. It does this by finding the maximum feasible proportionality factor $\alpha \cdot TM$. The solution to the linear program is found using the Gurobi solver [1]. The underlying linear program (see section 3.2.1 and 3.2.2) requires the following two input variables:

- **Topology:** a directed graph $G = (V, E)$ where each directed edge $e \in E$ has a link capacity defined;
- **Switch-level traffic matrix:** a traffic matrix $TM : V \times V \rightarrow \mathbb{R}$, which maps a directed pair of switches to its respective traffic.

Topology and Traffic Generation

To simplify usage, instead of forcing each time the supply of a full graph definition, the framework supports the topology variable by having built-in parameterizable graphs of which key attributes such as number of switches and servers, and the network degree can be supplied. The key topologies (Jellyfish, Xpander and fat-tree) introduced in chapter 2 are implemented.

Similarly, it enables easy definition of the traffic matrix by defining parameters, such as for random permutation pairs, all-to-all, minimum weight pairs and maximum weight pairs.

A universal seed is to control all randomness in both generation. This forces each run to be fully reproducible given the configuration file.

Extension: Path Evaluator

An extension was added to MCF Fair Condensed linear program (see section 3.2.2), which allows to restrict which edges are allowed to transfer which flows. In essence, it excludes certain edge-flows from participating in the constraints, and thus forcing them to not be able to be used to achieve the α proportional throughput. The extension allows users to specify one of the following path evaluators:

[1] http://www.gurobi.com/ (retrieved 31-01-2017)

- k**-slack:** an edge (u, v) for a flow $s \to t$ is only allowed to participate if the length of path $s \to$ *...shortest path...* $\to u \to v \to$ *...shortest path...* $\to t$ is lower than or equal to the length of path $s \to$ *...shortest path...* $\to t$ plus k;
- **Neighbor:** an edge (u, v) for a flow $s \to t$ is only allowed if it is on a shortest path from one of the neighbors of s to t;
- k**-shortest-paths:** an edge (u, v) for a flow $s \to t$ is only allowed if it is on one of the k-shortest-paths;
- k**-valiant-load-balancing:** an edge (u, v) for a flow $s \to t$ is only allowed if it is a shortest path from source s to one of the k valiant nodes u or from u to target t.

3.2.1 Simple Linear Program

The simple linear program does not track how much of each flow goes over an edge, but rather the aggregate. This makes it particularly suitable for dense switch-level traffic matrices (e.g. all-to-all), in which case keeping track of contributions of individual flows would cause an explosion in the number of variables. The linear program is defined as follows:

Variables and constants

α	Proportionality factor in $\alpha \cdot TM$.
$TM_{s,t}$	The constant value of the switch level matrix at index (s,t).
c_{link}	Link capacity, typically 1 by convention.
$f_{s,t}$	The amount of flow going from source s to sink t.
$l_{i,j,k}$	The amount of flow originating from node i going over edge (i,j) towards destination node k.

Objective

Maximize α subject to the following constraints.

Flow proportionality constraints; every flow is reduced equally such that there is proportionality towards the entire TM:

$$\forall (s,t) \in V \times V : f_{s,t} \leq \alpha \cdot TM_{s,t}$$

Link capacity constraints; every link is only capable of handling a certain amount of flow:

$$\forall (i,j) \in E : \sum_{k=1}^{n} l_{i,j,k} \leq c_{link}$$

Flow conservation constraints for distinct pairs; the amount of flow going from s to t plus the amount of flow coming into s from all other nodes towards t should be equal to the amount of flow going out of s:

$$\forall (s,t) \in V \times V \wedge s \neq t : f_{s,t} + \sum_{(j,s) \in E} l_{j,s,t} = \sum_{(s,i) \in E} l_{s,i,t}$$

Flow conservation constraints for individual nodes; the amount of flow that destined to go into destination t must be equal to the amount of flow coming from its edges towards it:

$$\forall s \in V : \sum_{(s,t) \in V \times V} f_{s,t} = \sum_{(j,t) \in E} l_{j,t,t}$$

3.2.2 MCF Fair Condensed Linear Program

The MCFFC linear program tracks individual flows. Instead of having the link capacity variable in the simple linear program, summations of flow contributions are used to define link-level constraints. This makes it particularly suited for sparse switch-level traffic matrices (e.g. pairings), in which not as many of the switch pairs participate. The linear program is defined as follows:

Variables and constants

α	Proportionality factor in $\alpha \cdot TM$.
$TM_{s,t}$	The constant value of the switch level matrix at index (s,t).
c_{link}	Link capacity, typically 1 by convention.
C_f	Number of flows in the TM (e.g. the number of non-zero entries).
$f_{fid,i,j}$	The amount of flow with identifier $0 \leq fid < C_f$ going over edge (i,j).

Objective

Maximize α subject to the following constraints.

Flow fairness constraints; for every flow (s,t), the sum of the flow leaving s must be greater than or equal to $\alpha \cdot TM_{s,t}$:

$$\forall (s,t) \in V \times V : \sum_{(s,j) \in E} f_{fid_{s,t},\, s,\, j} \geq \alpha \cdot TM_{s,t}$$

Link capacity constraints; every link is only capable of handling a certain amount of total summed flow:

$$\forall (i,j) \in E : \sum_{fid=0}^{C_f} f_{fid,i,j} \leq c_{link}$$

Flow conservation constraints for intermediary nodes; all flow fid entering the intermediary node must also leave it:

$$\forall (s,t) \in V \times V : \forall u \in V \wedge s \neq u \wedge t \neq u : \sum_{(j,u) \in E} f_{fid_{s,t},j,u} = \sum_{(u,j) \in E} f_{fid_{s,t},u,j}$$

Flow conservation constraints for source nodes; no flow that comes into s originates from a flow of itself (thus forcing it to be deposited at the sink):

$$\forall (s,t) \in V \times V : \sum_{(j,s) \in E} f_{fid_{s,t},j,s} = 0$$

3.3 Traffic Flow Results

3.3.1 Fat-tree Flow Results

For the experiment a $k = 16$ fat-tree is used, which has 320 switches with a degree of 16 and 1024 servers at the bottom. Per section 3.1.2, an all-to-all fraction inter-pod is used. The results are shown in figure 3.1. The increments on the x-axis are pods being added to the all-to-all fraction, as it is a worst-case scenario. As expected, in a full fat-tree a per-server throughput of 1.0 is achieved due to the full non-blocking fabric that it provides. In general it is true that any fat-tree oversubscribed by $\beta\%$ can achieve a $(100 - \beta)\%$ throughput per server. This is demonstrated by oversubscribing the $k = 16$ fat-tree by removing 25% and 50% of its 64 core switches, resulting in the predicted server throughput of respectively 0.75 and 0.5 in figure 3.1. As there are only (routing-wise useful) shortest paths between (pod) nodes in a fat-tree, there is no different outcome using either all paths or only shortest paths.

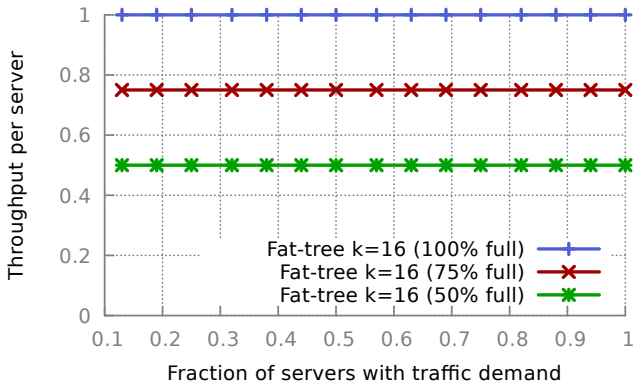


Figure 3.1: Flow results for Fat-tree topology ($k = 16$); reproduce using *TopoBench/scripts/thesis/thesis_fat_tree_{0, 25, 50}.sh*

3.3.2 Expanders Flow Results

For the experiment two sets of $n = 128$ Jellyfish graphs with a degree of 8 are used; each node has 8 servers underneath. The first set is without any filtering, whereas for the second set, candidate random regular graphs (Jellyfish) have been filtered by spectral gap. According to Valdarsky et al. [27], this can serve as a computationally efficient way of evaluating how good of an expander a random graph is. Overall evidence seems to suggest that filtered appears to perform similar for the all-to-all fraction and better for the pairings, though further more statistically thorough work is required for more rigorous conclusions. The results are averaged over the 10 random seeds, and for each the standard deviation as fraction of the mean, henceforth named σ_m, is calculated to quantify variance. As this is about expanders, only the filtered results are used for in-text statistics.

All-to-All Fraction

The first experiment is using an all-to-all fraction of communicating nodes, meaning that within the fraction all nodes communicate with each other. The results are shown in figure 3.2. For all-paths, variance metric σ_m decreases from 2.2-3.1% for low-fraction to 0.1-0.2% at high-fraction; for shortest-paths, from 17-22% to 0.1-0.2%. The expander topology performs well in both all-paths and shortest-paths, going towards the level of throughput proportionality for all-paths. Moreover, shortest-paths perform well, especially as load comes higher. This is because random (regular) graphs (or expanders for that matter) approximate closely the bound of lowest average path length described in [25]. In a situation where there is high network load, short paths are preferred as they consume less network capacity; this is evident as there is approximate convergence at the highest fractions.

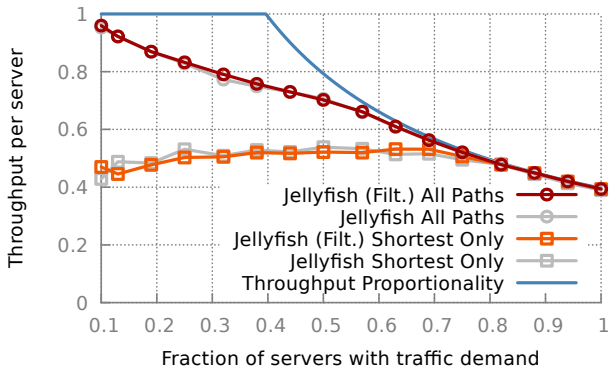
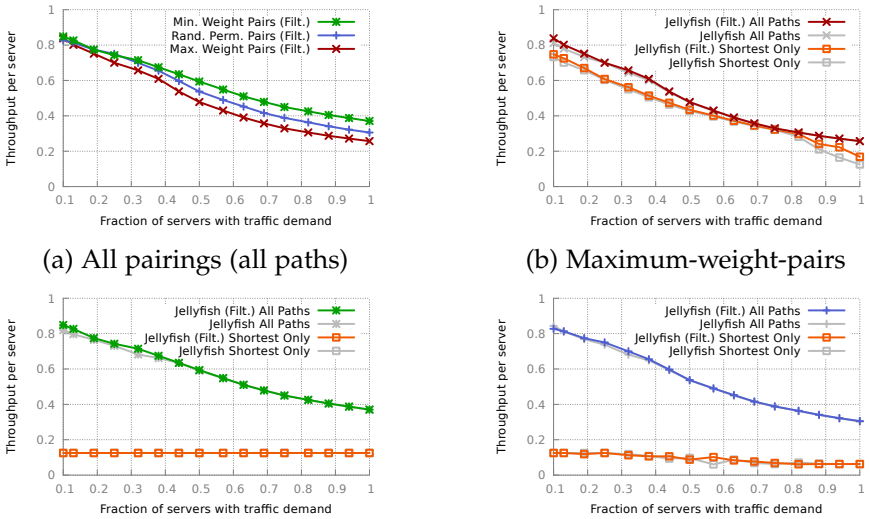


Figure 3.2: Flow results for $n = 128$ Jellyfish topology ($d = 8$) in an all-to-all-fraction; reproduce using *TopoBench/scripts/thesis/thesis_jellyfish{_filtered,}_all_to_all_fraction.sh*



(a) All pairings (all paths)

(b) Maximum-weight-pairs

(c) Minimum-weight-pairs

(d) Random-permutation-pairs

Figure 3.3: Pair flow results for Jellyfish topology ($n = 128, d = 8$) ; reproduce using *TopoBench/scripts/thesis/thesis_jellyfish_{filtered, }_{mawp, miwp, rpp}.sh*

Maximum-Weight-Pairs

The second experiment is using maximum-weight-pairs, which is the best-known most-difficult traffic scenario for expanders (see section 3.1.3), as is shown in figure 3.3a achieving the lowest throughput per server for any fraction amongst all tested pairings. The sole maximum weight pair results are shown in figure 3.3b. For all-paths, the variance metric σ_m decreases from 8% at fraction 0.1 to 0.5% at the full fraction 1.0; for shortest-paths, σ_m decreases from 3-5% at low fractions to 1% at fraction 0.75 before increasing to 30-40% at the tail. The pairs are chosen as distant from each other as possible, forcing their flows to have a higher chance of needing to share links.

Despite the restriction that only shortest-paths are allowed to be taken by flow between the pairs, it performs considerably well. This is due to the construction of maximum weight pairs: by choosing pairs of nodes very distant from each other, it results in a high likelihood that all neighbors of a source node are close (one closer) to the target node. Thus, there are many paths a flow can take. The continued decline towards the end of the tail is due to the more-and-more forced share of link capacity as all nodes are becoming involved. At the tail, less distant nodes are added as well, which have less probability of the previously discussed "*neighbor effect*". The tail variance for shortest-paths is due to a combination of congestion and the addition of nodes with less "*neighbor effect*".

Minimum-Weight-Pairs

The third experiment is using minimum-weight-pairs, as it intuitively forces path diversity to be present to achieve good flow performance. The results are shown in figure 3.3c. For all-paths, the variance metric σ_m goes from 3.3-4% at lower fractions to 0.4-1% at high fractions; for shortest-paths, the variance metric σ_m is 0% for all fractions. When the flow optimization can use all flows, it achieves better performance than that of maximum-weight-pairs. The story changes for when only being able to use shortest paths; the maximum directed node pair throughput is 1.0 as there are no double links.

Random-Permutation-Pairs

The fourth experiment is using random-permutation-pairs, which is expected to be an even worse case for shortest path routing. The results are shown in figure 3.3d. For all-paths, the variance metric σ_m goes from 3.6% at the lowest fraction to 0.5-1% at higher fractions; for shortest-paths the variance ranges from 30-40%, being lower in the lowest fractions and the highest fractions. As it does not select extremely close nodes, nor extremely far nodes, the performance of all-paths is between minimum weight pairs and maximum weight pairs (as shown in figure 3.3a). When being only able to shortest-paths, it performs even poorer than minimum-weight-pairs. This is because, by chance, it can for example force two pairs with only a single path to share a link, thus achieving a mere throughput of 0.5. The probability of this (or worse) occurring increases with an increasing fraction of traffic pairs.

3.3.3 Combined Equal-cost Flow Results

The final experiment is to compare the three topologies (fat-tree, expander, and dynamic) when generating each at equal-cost. Following [24], a 578 node topology with 49 ports is used supporting a total of 13872 servers. The following topologies are equal-cost for this scenario:

- Fat-tree (using calculation in appendix B):
  357 ToRs at bottom layer, 147 at the aggregation layer, and 74 at the core layer; achieves a throughput of 0.2604 ($\beta = 74\%$ oversubscribed).
- Expander (Jellyfish; filtered):
  $n = 578$, $d_{network} = 25$, $d_{server} = 24$, 10 random seeds
- *Perfect* dynamic topology (different port costs):
  $n = 578$, $d_{network} = \{1/2, 2/3, 1\} \cdot 25$, $d_{server} = 24$; achieves a throughput of respectively $1/2 \cdot 25/24 = 0.521$, $2/3 \cdot 25/24 = 0.694$, and $25/24 = 1.042$.
- *Restricted* dynamic topology (using calculation in appendix C):
  $n =$active fraction of 578, $d_{network} = 25$, $d_{server} = 24$

The results are shown in figure 3.4. The worst traffic scenarios for each are used. The equal-cost fat-tree ($\beta = 74\%$) offers the expected constant throughput of $(1 - \beta)$. The expander ($\sigma_m \leq 1\%$ for all fractions) offers significantly higher throughput as its fat-tree counterpart at equal-cost. The perfect dynamic topology adaptations (TA) can online create direct connections between nodes communicating and does not suffer from any buffering restrictions thus making full use of its network ports. The restricted topology adaptation, which does not permit any buffering, has dynamically settled on the most optimal static network. In the lower active fractions, the expander shows potential to exceed its dynamic topology counterparts.

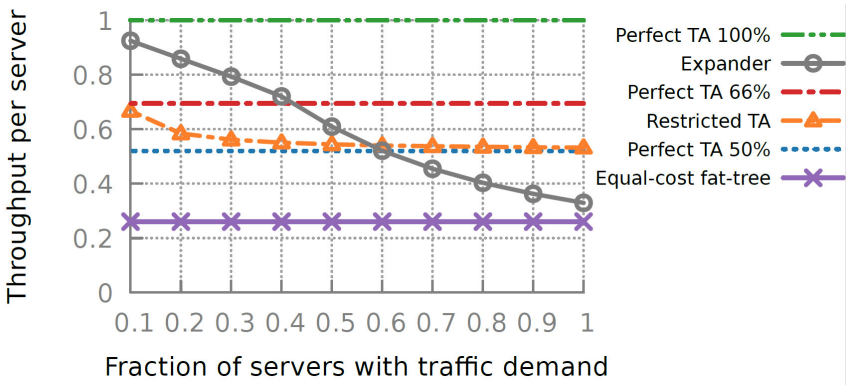


Figure 3.4: Equal-cost fat-tree and expanders, and dynamic topology bounds. Reproduce using *TopoBench/scripts/thesis_jellyfish_filtered_578.sh*.

3.4 Flow Evaluation Conclusion

The traffic flow results have created important insight into the potential throughput that the topologies can achieve in (best-known) worst case traffic scenarios. Fat trees are able to support as much throughput as their non-blocking switch fabric allows. If the non-blocking switch fabric is oversubscribed by β%, then the fat-tree is able to support $(100 - \beta)$% of throughput. The fat-tree, by construction, achieves optimal path diversity using shortest paths only. Expanders are not guaranteed to be non-blocking under all traffic scenarios, but can generally achieve a good throughput due to their path diversity as [27] found that every cut in an expander is traversed by many links. In certain cases, all-to-all-fraction and maximum-weight pairs, even using only shortest paths achieves good throughput, whereas in other cases, minimum-weight pairs and random-permutation pairs, poor performance is achieved using only shortest paths. The combined (equal-cost) flow results show that the expander can exceed most of its dynamic topology counterparts, and exceeds the performance of equal-cost fat-trees under worst-case scenarios. In light of these results, three observations are made.

First, all flow results were under optimal traffic engineering. In fat-trees, optimal traffic engineering only needs to concern itself with shortest paths. In expanders however, for various traffic scenarios, there is a need for routing along many different length paths to achieve the throughput potential that has been shown in the flow evaluations. In some cases, using only shortest paths suffices, in other cases it will not. Can we create a routing strategy that is able to exploit the potential in the expander network structure?

Second, limited (theoretical) models exist to sufficiently assess the extent of potential performance by dynamic topologies. The models presented, *perfect* and *restricted*, both show the ability to outperform fat-trees and static topologies. Of course by definition, under equal equipment cost and capability, a dynamic network can achieve everything a static network can. The modeled topologies were compared that either do not account for buffering (perfect) or are forced to use only direct connections in its most efficient manner (restricted). Even though ports are more expensive, is it possible to create a matching (and routing) strategy that makes significantly more use of these ports, better than is possible for any routing scheme in static networks using mere static ports?

Third, this chapter concerned itself with worst-case traffic scenarios, which provides theoretical bounds for flow results. How does this flow model translate to actual physical data center networks: how can we engineer the flow throughput potential into an effective routing scheme? Knowing the potential in worst-cases, more specifically: which traffic scenarios occur in data centers and how does the routing scheme need to react to realize the potential?

It is needed to take a step back from the high abstraction of flow evaluations to answer these questions. In the following chapter, a discrete packet simulator is used to evaluate routing strategies for static topologies. Performance gains for static topologies are compared to that of existing dynamic topology research claims.

Chapter 4

Discrete Packet Simulation

Static topologies are evaluated from a packet-level perspective in this chapter. It is done to overcome the trade-offs made by flow evaluation, which favored model simplicity at the cost of the ability to express more representational models. Specifically, the flow model abstracts over the complexity of traffic engineering. To construct this more representational model, existing network techniques such as TCP and ECMP are introduced and discussed in section 4.1. Second, in section 4.2, the implemented network simulator NetBench is introduced which is driven by discrete events and incorporates implementations based on the network techniques to simulate network scenarios at packet-level. Third, in section 4.3, experiments run with the network simulator are featured. Fourth, in section 4.4, the expander routing deployment is analyzed from an engineering perspective. Finally, in section 4.5, a high-level conclusion is made based on the results.

4.1 Network Techniques

This section shapes an impression of the landscape of transport protocols and routing in the field of data center networking. First, the main established network techniques TCP (section 4.1.1), DCTCP (section 4.1.2) and ECMP (section 4.1.3) are introduced in this section. Second, in section 4.1.4, a selection of novel ideas are introduced to highlight important developments beyond established network techniques.

4.1.1 TCP: Transmission Control Protocol

Transmission Control Protocol (TCP) is a widely deployed transport protocol. Several RFCs have proposed variants of its logic, most notably the original by Postel et al. [22], and Allman et al. [6] which specifies algorithms for slow start, congestion avoidance, fast retransmit and fast recovery. Based

on these RFCs, an interpretation is presented in the following paragraphs which is used throughout this thesis and is implemented in the simulator.

TCP assumes that the underlying network provides a single path for each TCP socket established. TCP guarantees in-order lossless delivery of packets, forcing the destination to wait (and re-order packets) until this is guaranteed before allowing them to flow to the application layer. There are two phases: (a) the *slow start* phase, in which it roughly linearly increases its congestion window size *cwnd* with the acknowledgments it receives until the slow start threshold *ssthresh* is reached, after which it goes into (b) the *congestion avoidance phase*, in which it increases its congestion window size with approximately one packet per round-trip-time (RTT).

In the simulator TCP implementation, the receiver is simple and cumulatively acknowledges every packet it receives and has an infinitely large receiver window. A flow is finished once the sender has received an acknowledgment for every segment of the flow. The sender sends out at most the *cwnd*. Space in the *cwnd* is only opened up upon reception of the acknowledgment for the left-most segment in it, emulating in-order delivery to an imaginary application layer. The TCP implementation employs *selective acknowledgment*: if the sender receives acknowledgments of not the left-most segment (out-of-order packets), it saves which segments have been acknowledged. A retransmission time-out is set for each data packet. Once the missing left-most segment is acknowledged, the *cwnd* space is freed (by moving $SND.UNA$) until the last non-acknowledged segment. As a final recovery measure, the sender maintains a timer for every packet which starts when it is sent out. If the timer exceeds the Retransmission Timeout (RTO), it considers it a loss. A single loss will half the congestion window size *cwnd*. TCP parameters are set based on a combination of RFC recommendations and experimental observations, namely $SMSS = 1380\ bytes$, $IW = 3 \cdot SMSS$, $RTO = max\{1ms,\ 2 \cdot RTT + 4 \cdot VAR\}$, $LW = 1 \cdot SMSS$, and $ssthresh = 30 \cdot SMSS$.

A disadvantage in data centers is that TCP forces queue buffers on the path to be consistently build-up to its maximum capacity, causing packet losses to occur once the queue limit is reached, which in their turn makes TCP reduce its rate exponentially. This leads to the common tooth-saw pattern in queue length and latency observed in TCP measurements. DCTCP (see section 4.1.2) focuses on resolving the latter problem, enabling *"low latency for short flows, high burst tolerance and high utilization for long flows"* [3].

4.1.2 DCTCP: Data Center TCP

Data Center TCP (DCTCP) [3][1] is an improvement upon TCP for the data center setting. It addresses the issue of when to adjust the sending rate *cwnd* based on the buffering happening at output queues of switches on a TCP path. Regular TCP continues to increase its sending window until packets are lost due to exceeding buffer limits, forcing it to exponentially reduce the window again. This behavior results in persistent large buffers on the route, thus inducing unnecessary latency due to queue buildup. DCTCP uses the ECN (Explicit Congestion Notification) flag on acknowledgment packets (either normal or delayed, for multiple packets) to provide feedback on buffer state on the path, thus allowing the end-hosts to reduce their window before the buffer becomes full. The sender maintains, for a given time-window, the fraction F of packets that have the ECN-Echo marker to determine the congestion of the path: $alpha \leftarrow (1-g) \cdot \alpha_{previous} + g \cdot F$ (g is a weight parameter for past vs. current rates). It decreases its congestion window *cwnd* proportionally for every marked packet it receives: $cwnd \leftarrow cwnd \cdot (1 - \alpha/2)$. In the implementation, g is set to $1/16$ and K to 20 packets.

There are three main requirements that DCTCP fulfills [3]. First, it provides burst tolerance as queues are less full and thus are better able to handle a sudden increase. Second, it achieves lower latency for short flows. In data centers there are typically a few large flows and a lot of transient short flows. TCP handles this poorly, leading to that if a large flow shares an edge with a short transient flow, the short flow will have poor latency and flow completion time. It also remedies the incast scenario to a certain extent, where many small transient flows simultaneously want to send to a single destination. DCTCP prevents the inevitable back-off caused by a sudden loss of packets due to queue congestion at the target destination. Third, it achieves the same throughput as TCP.

4.1.3 ECMP: Equal-Cost Multi-Path

ECMP (Equal-Cost Multi-Path) routing is a multi-path forwarding network protocol. It is analyzed in RFC2992 [14], on which the interpretation of ECMP in this section is based. For each destination, the forwarding table stores K next hop neighbors which must be on a shortest path to the destination. It calculates a hash[2] based on the flow identifying packet headers, mapping it inside one of the K regions to determine the next-hop neighbor. This means that even though there are multiple paths possible and known

[1] Implementation adapted from ns-2 patch proposed by Alizadeh et al. at http://simula.stanford.edu/~alizade/Site/DCTCP.html (retrieved 24-01-2017)

[2] RFC2992 [14] assumes CRC16, though in actual deployment other (proprietary) hashing functions can be used. Robert Jenkins' 32-bit integer hash function (http://burtleburtle.net/bob/hash/integer.html; retrieved 09-02-2017) is used in the simulator.

to the router, each single flow (data stream) is sent along a single predetermined path by hashes. However, upon detection of failure of a link, ECMP does enable the usage of another path as the hash-space is re-mapped by the switch using the link. Though not stated in the RFC, a switch-unique seed is also added to the hash such that it makes use of all possible paths.

A problematic scenario for ECMP is for a (src, dst)-pair in a topology in which there are multiple paths to the destination, but only very few (or even just a single) are shortest paths. ECMP can then only use a fraction of the available network capacity. ECMP works particularly well in tree-like topologies, where all paths between servers are of equal length.

Even in favorable topologies, Alizadeh et al. [2] (authors of CONGA) note two main problems with ECMP:

- "*Because ECMP randomly hashes flows to paths, hash collisions can cause imbalance in network utilization especially when there a few large flows*";
- "*ECMP uses a purely local decision to split traffic among equal cost paths without knowledge of potential downstream congestion on each path. Thus ECMP fares poorly with asymmetry caused by link failures that occur frequently and are disruptive in datacenters [11, 19]*".

4.1.4 Recent Developments

Flowlets. [2] states flowlet switching as a concept first introduced by [16]. Flowlets work as follows: if the gap between sending subsequent packets is larger than the maximum difference of time of all possible paths, the next gap-less sequential group of packets is called a new flowlet. The flowlet gap should be set at a value that balances the cost of potential reordering at the destination and the decision logic of sending a flowlet along another path than the previous flowlet to better distribute load across the network.

Congestion-sensitive proposals. Beyond the established network techniques TCP, DCTCP and ECMP, new techniques have been proposed particularly focused on answering the question of how to handle more sophisticated congestion signs such as ECN markers (as DCTCP) instead of relying on loss. CONGA [2] attempts to overcome shortcomings of ECMP through keeping a running estimate of congestion of each path by piggybacking congestion information to the source. It allows a new flowlet to switch ECMP path to the least congest ECMP path known to the source. CLOVE [17] performs a path discovery phase, in which it select the most distinct paths. During run-time it maintains a running estimate of the congestion of each path via ECN marked packets. Upon a flowlet gap, it weighs the available paths by their history of congestion and performs weighed round-robin to determine to which path to switch. RackCC [29] employs the interesting notion of storing congestion information at the rack, and assigning and adjusting rate to

constituent flows on-line. MPTCP (e.g. as defined by [28]) allows a single flow to be separated into multiple sub-flows.

4.2 NetBench: Discrete Packet Simulator

To assess routing in static topologies, a packet simulator using discrete events was implemented named *NetBench*. It was created with especially the design principles of maintainability, extensibility, and understandability in mind. The three main modules are discussed in the following sections. First, the infrastructure features are described in section 4.2.1. Second, the routing initialization is described in section 4.2.2. Third, the traffic generation model is explained in section 4.2.3.

4.2.1 Infrastructure

The infrastructure of a simulation refers to the modeled physical components of the network. There are five main choices that need to be made: the link, the output port, the network device, the flowlet intermediary, and the transport layer. There is no model for input port queuing. The interplay between these five components is visualized in figure 4.1.

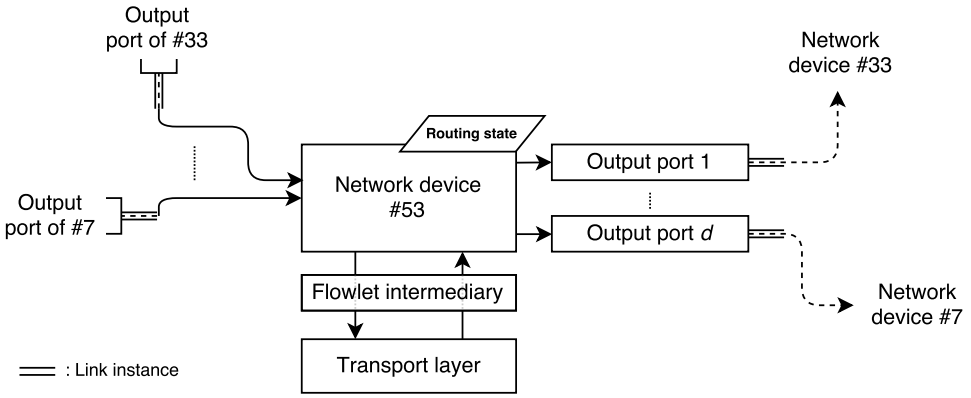


Figure 4.1: NetBench infrastructure example. Network device #53 has two bi-directional links to #33 and #7.

Link

The link instance quantifies the capabilities of the physical link, which the output port adheres to. The modeled capabilities are: (a) delay (ns), (b) bandwidth (bit/ns), and (c) drop rate (probability of a packet not arriving, e.g. through interference). In all experiments, links are used with a bandwidth of 10 bit/ns (10 Gbit/s) and a delay of 20ns without any chance of dropping a packet due to link failure. A bi-directional cable of a data center is modeled as two links with respective output port in opposite directions.

Output port

The network simulation only models output ports of network devices. An output port receives a packet from the network device and does its best effort according to its protocol and restrictions to transmit it to the *target network device*. A single implementation is used throughout the experiments: an *ECN tail drop output port*. The tail drop queue accepts packet into its FIFO queue until the maximum buffer queue size is reached, at which point it simply drops packets. It marks packets with the ECN flag that arrive when the current buffer queue size exceeds the ECN threshold. Pseudo-code of the algorithm it uses is described in appendix D.1.

Network device

The network device resembles the concept of a node in a graph. It always has a collection of output ports. A network device with a transport layer attached to it is a server, whereas a network device without is only a switch. A network device directly receives incoming packets from the output ports of which it is the target network device. When receiving 'foreign' packets, it decides based on the destination which of its output ports should queue it, or if it should pass it on to the underlying transport layer typically if that the packet's destination. It can also receive packets from the underlying transport layer, which it can either directly forward or for example encapsulate to encode routing-specific state.

There are four base network devices that are used in tests and the experiments:

- *Forwarder-Switch:* received packets are forwarded to a single next-hop neighbor;
- *ECMP-Switch:* a hash based on (source, destination, flow_id, flowlet_id) is calculated that decides to which of the next-hop neighbors it should forward the packet;
- *Valiant-Switch:* upon reception of a packet from the transport layer, a valiant encapsulation is created. A hash based on (source, destination, flow_id, flowlet_id) is calculated, which determines to which of the available valiant nodes the encapsulation is sent. Routing of the encapsulation is done the same as for an ECMP-switch. Once an encapsulation has passed the valiant node, it is removed and is directed to its actual destination.
- *Source-Routing-Switch:* upon reception of a packet from the transport layer, a source routing encapsulation is created. A hash based on (source, destination, flow_id, flowlet_id) is calculated, which determines which of the k paths to the destination are taken. The chosen path is

stored in the encapsulation, and is forwarded by the switches exactly as the path describes.

Besides these four base network devices, specific variants of the network devices have been created to test out slightly different tactics.

- *Random-Valiant-Switch:* it is possible to specify a range of network device indices out of which to choose the valiant node.
- *ECMP-then-Valiant-Hybrid-Switch:* until threshold Q (typically set to $100KB$) is reached, the flow is sent over direct ECMP shortest paths. Afterwards, the flow is sent over a valiant load balancing path;

Flowlet intermediary

Besides the default routing logic of network devices, there is a *flowlet intermediaries* for each, which does a pass over every packet that goes into and comes out of the transport layer. The conceptual idea of a flowlet is explained in section 4.1.4.

There are three flowlet intermediaries:

- *Identity:* flowlet identifier is never modified and stays at 0;
- *Uniform:* once a flowlet gap is detected, it increases the flowlet identifier by one and thus the next flowlet will take a different path if the switch logic incorporates the flowlet identifier in its routing decision;
- *DCTCP-detect-uniform:* only changes the flowlet identifier of a flow if a flowlet gap is detected and a draw from a Bernoulli-distribution with $p = DCTCP\text{-}\alpha\text{-}fraction$ comes out positive. This might be useful following for two reasons: (a) it would be adverse to change the path of the next flowlet when there is no congestion, and (b) in e.g. a scenario of two conflicting flows in which only one flow should move, it lessens the probability of both flows changing path.

Transport layer

The transport layer maintains the sockets for each of the flows that are started at the network device and for which it is the destination. There are two possible transport layers: TCP (see section 4.1.1) and DCTCP (see section 4.1.2).

4.2.2 Routing

The routing initializes the routing state stored on network devices. The network devices use their internal logic and routing state to determine to which output port (or to the underlying transport layer) it should send a

packet. Network devices are based on abstract classes that certain routing initialization schemes require, e.g. ECMP-Switch is the base class needed for ECMP routing initialization.

The possible routing initialization strategies are as follows:

- *Single-forward:* first calculates all shortest path lengths using Floyd-Warshall, and then for every (src, dst)-pair selects, of all possible outgoing edges of *src* on a shortest path towards the *dst*, the next-hop to the node with the lowest node index (as a heuristic);
- *ECMP:* first calculates all shortest path lengths using Floyd-Warshall, and then for every (src, dst)-pair saves all the outgoing edges of *src* on a shortest path towards the *dst* as a set of possible next-hops;
- *K-paths:* for every (src, dst)-pair it reads from file (up to) K arbitrary paths between the two and saves them as routing state;
- *K-shortest-paths:* performs Yen's *K*-shortest-paths algorithm to find the *K* paths for every (src, dst)-pair and saves them as routing state. It also saves them in a file such that it can be used via *K*-paths next time.

4.2.3 Traffic

The traffic of the simulator is planned using flow start events. A flow start event has four variables: (a) a time, (b) a source, (c) a destination, and (d) the flow size. A Poisson traffic schedule is used to generate these flow start events before the run of the simulation. Three components are needed to generate a Poisson traffic schedule for a certain experiment duration: (a) λ, the number of flow starts per second; (b) $p_{flow\,size}$, the flow size distribution; and (c) p_{pair}, a communication pair probability distribution. This strategy of flow planning is employed as well by [5] and [10]. The procedure is described in algorithm 2. The choice for the components are explained in the following sections.

Algorithm 2 Generating Poisson traffic schedule

Require: $\lambda\ (starts/s),\ p_{flow\,size},\ p_{pair},\ duration\ (ns)$
1: $time = 0$
2: **while** $time \leq duration$ **do**
3: Indep. draw $pair$ from p_{pair}
4: Indep. draw $flowsize$ from $p_{flow\,size}$
5: Create flow start event $(time,\ pair,\ flowsize)$
6: Indep. draw $intertime$ from λ-Poisson distribution
7: $time = time + intertime$
8: **end while**

Flow start frequency λ

The start frequency determines how many flows are started per time unit. Once the flow size and pair distribution are fixed, it is used to adjust the corresponding load on the network. Skewness of the flow size distribution and (server) pair distribution especially have impact what value of λ creates bottlenecks in the network. [10] indicates that the maximum load should be when the busiest link has an average utilization of 80%, at which [10] indicates 5% of its flows do not finish in its custom simulator.

Flow size distribution $p_{flow\,size}$

The goal of the flow size distribution is to emulate the actual sizes of flows occurring in the target emulated network. Each flow start event independently draws its flow size from this distribution. The range of possible flow size distributions of the experiments is discussed in section 4.3.

Pair distribution p_{pair}

The goal of the pair distribution is to emulate the skewness of communicating nodes or servers in the target emulated network. Each flow start event independently draws its (src, dst)-pair from this distribution. The range of possible flow size distributions of the experiments is discussed in section 4.3.

4.3 Experiments

In this section the setup of the experiments and the results of the experiments are discussed. First, in section 4.3.1, the flow size distributions are described. Second, in section 4.3.2, the traffic pair probability distribution used are explained. Third, in section 4.3.3, the used routing options are detailed. Fourth, in section 4.3.5, the ToR-ratio metric is defined. Fifth, in section 4.3.6, a set-up similar to ProjecToR [10] is created. Sixth, in sections 4.3.7 and 4.3.8, a respectively low and medium node all-to-all-fraction traffic scenario is evaluated. Seventh, in section 4.3.9 and 4.3.10, traffic scenarios based on Pareto skewed pair probabilities are examined. Eighth, in section 4.3.11, server all-to-all fractions are evaluated.

4.3.1 Flow size distributions

Two main flow size distributions are used:

- **Alizadeh Web Search (ALW):** A discrete flow size distribution which is introduced in [5] as Web Search, cited as the more challenging one of the two distributions in that paper. The upper bound of the discrete CDF is taken. The expectation flow size is approximately 2.4MB, the minimum 10KB and the maximum 30MB. A similar flow size distribution is also used by ProjecToR [10]. ALW is considered a moderately skewed flow size distribution with a high average.
- **Pareto (σ, μ) (PAR):** A Pareto distribution which is also used in [4]. The average flow size μ is set to 100KB, and the shape σ to 1.05. To prevent extremely large perpetual flows from forming, the independent draw is capped at 1GB which would take $870\mu s$ at the full true link rate of 9.2 Gbps to complete. PAR is considered a heavily skewed flow size distribution with a low average.

The following notation is used to indicate that flow arrival parameter λ is associated with a particular flow size distribution XYZ to create a workload: XYZ-λ. For example an Alizadeh Web Search (ALW) flow size distribution coupled to $\lambda = 90,000$ flow arrivals per second is noted as ALW-90K.

4.3.2 Pair probability distributions

Four main pair probability distributions are used:

- **ProjecToR:** An empirical 128 node-level directed communication pair probability released by Ghobadi et al. [10]. It is heavily skewed, in which only a few nodes are very likely to participate. The node-level probabilities are split into $s \times s$ server-level probabilities if applicable.

- **Node all-to-all-fraction:** A certain fraction of nodes are chosen to communicate in an all-to-all. For a fat-tree, the fraction is chosen left-to-right, forcing new pods to be added with increasing fraction. For an expander, the fraction is chosen randomly as the structure of an expander with high probability resembles a random regular graph [27]. The node-level probabilities are split into $s \times s$ server-level probabilities if applicable (communication between servers on the same node does not happen).
- **Node Pareto (σ, x_m):** A Pareto distribution with a specific shape σ and scale x_m is used. The pair probability distribution is generated as follows: (a) each pair draws its "mass" from the Pareto distribution, and (b) each pair's "mass" is converted into its probability by normalizing by the total drawn mass.
- **Server all-to-all-fraction:** A certain fraction of randomly picked servers are chosen to communicate in an all-to-all.

4.3.3 Routing options

Many different architectural (including routing) options were implemented in the simulator (see section 4.2). Not all implemented features were included in the results presented in this work. These features are provided to aid future work in studying their effects. Three main routing options are employed. A flowlet-timeout of $50\mu s$ is used, i.e. packet sequences separated by more than $50\mu s$ are considered separate flowlets. The *uniform* flowlet intermediary increments the flowlet identifier every time such a gap occurs. A flowlet's path is decided based on a hash including the flowlet identifier. Thus each flowlet is hashed independently to a path. The routing options are defined as follows:

- **ECMP:** Utilizes only shortest path routing. A change in flowlet identifier will result in a potentially different hash at each of the switches which can lead to a different ECMP path. ECMP is applied to every fat-tree in the experiments;
- **VLB:** Utilizes only valiant load balancing paths, uses ECMP from source to valiant and from valiant to destination. The valiant node chosen is dependent on a hash including the flowlet identifier. Thus, a change in flowlet identifier will result in a potentially different valiant node and affects the two paths the same as ECMP;
- **HYBRID:** First does ECMP routing until the amount of flow data sent by the sender exceeds threshold Q (set to 100 KB if not specified otherwise), after which it will switch to VLB for the remainder of the flow. Effect of change of flowlet identifier depends on which of the two states it is in.

4.3.4 Configurations

The following configurations are selectively compared throughout the experiments[3]:

- Full fat-tree ($k = 16$, 100%) using ECMP. It supports a total of 1024 servers;
- Core-oversubscribed fat-tree ($k = 16$, 50%) using ECMP. At the core layer, 50% of the nodes are uniformly removed. It supports a total of 1024 servers;
- Xpander ($n = 256, d = 12, s = 4$) using either ECMP, VLB, or HYBRID. This is 80% of the full fat-tree's equipment and supports 1024 servers;
- Xpander ($n = 216, d = 11, s = 5$) using either ECMP, VLB, or HYBRID. This is 67.5% of the full fat-tree's equipment and supports 1080 servers (56 more).

4.3.5 Definition: ToR-ratio

The configuration that performs worse is the one which has the most flows sharing links. To predict which configuration performs worse in low fraction of participating nodes, it is especially important to look at the ratio of amount of servers per ToR and the out-degree, coined the *ToR-ratio*. The higher the ToR-ratio, (a) the less likely at a source node a flow collision occurs when flows originating from the servers are spread oblivious to the outgoing network links, and (b) the less likely at a destination node a flow collision occurs when flows coming from the supporting fabric traverse the incoming links of the destination node. A visualisation of the ToR-ratio is depicted in figure 4.2. The ToR-ratio for the configurations is as follows. First, for all fat-tree configurations, the ToR-ratio is 1:1. Second, for the Xpander ($n = 256$, $d = 12$, $s = 4$) the ToR-ratio is 3:1. Third, for the Xpander ($n = 216$, $d = 11$, $s = 5$) the ToR-ratio is 2.2:1. The ToR-ratio is an imperfect metric, as it does not account properly for continuing probability of flow conflict in the remainder of the supporting network fabric (beyond what is considered ingress/egress).

[3]These exclude the specific Xpander ($n = 128, d = 16$) configuration used in the ProjecToR comparison.

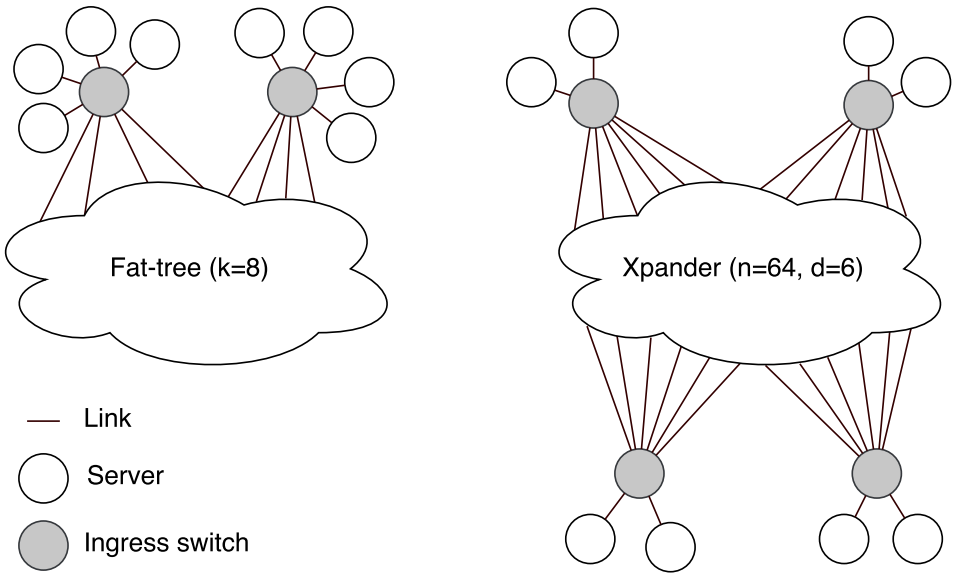
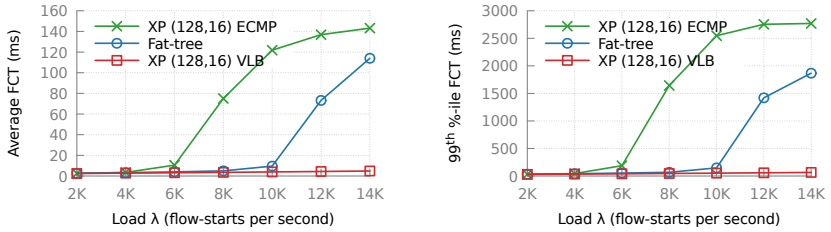
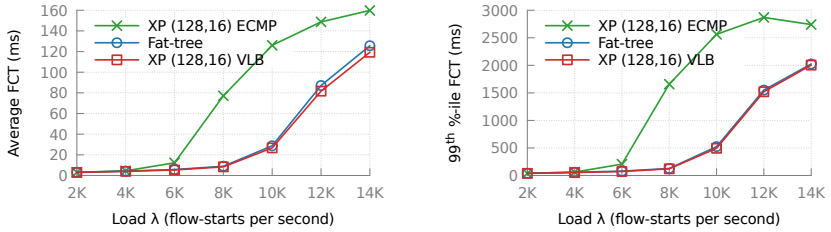


Figure 4.2: ToR-ratio difference example between $k = 8$ fat-tree (1:1) and an Xpander (3:1) which has 80% of the same network equipment.



(a) Avg. FCT w/o server bottlenecks (b) 99th FCT w/o server bottlenecks



(c) Avg. FCT w/server bottlenecks (d) 99th FCT w/server bottlenecks

Figure 4.3: Packet-level results for the ProjecToR workload

4.3.6 ProjecToR experiments

Main findings. When not modeling server up-links, the expander topology is able to achieve similar performance gains (7-96% achieved vs. 30-95%) as the state-of-the-art dynamic ProjecToR relative to the fat-tree without even accounting for the higher cost of dynamic ports. Upon adding server up-links, which is crucial for fair inter-topology comparison, the performance gains are reduced to a less significant 5-6%.

Motivation

ProjecToR [10] is the most recent in a line of dynamic network research. In their evaluation a comparison is made between a full-bisection k-fat-tree and a ProjecToR topology ($n = 128, d = 16$). They provision k dynamic ports for each of the ToRs in the ProjecToR interconnect, along with $k/2$ ports to connect to servers. This results in both topologies supporting an equal amount of servers. There is one major issue that is overlooked in this comparison: this set-up would inherently disadvantage the fat-tree. The nodes in the bottom layer of a fat-tree only possess eight up-links to the non-blocking fabric above (see section 2.1). This means that each ToR in the fat-tree can send a mere 50% of what the ProjecToR can send into the supporting network fabric.

Set-up

To demonstrate the issue, a similar set-up is created. The (skewed) communication probabilities of ProjecToR [10] and the Web Search flow size distribution of Alizadeh et al. [5] (ALW) are used. As in their simulations, a $k = 16$ fat-tree is used as the benchmark. As the contender for the ProjecToR, a Xpander topology of $n = 128$ and $d = 16$ is used. Note that this does not account for additional cost of dynamic ports. Simulations are run with a flow arrival ranging from $\lambda = 2$K to $\lambda = 14$K flow starts per second. Each simulation has the same identical set of flows of size 112K, which means that the simulation time ranges from 56s to 8s.

Results

Due to the skewness of the pair distribution, only very few nodes have a chance of being involved in traffic. For both with and without servers, ECMP in the Xpander topology performs poorly as it can only use very few of the available paths. This was observed earlier in the flow simulations for random permutation pairs (see section 3.3.2). Without any server up-links (see fig. 4.3a and 4.3b), the Xpander topology greatly outperforms the fat-tree topology when using valiant load balancing. This is because without server up-links, the fat-tree is bottlenecked by the out-degree of its bottom layer nodes whereas the Xpander is not.

As range of comparison, an equivalent to what is done in the ProjecToR comparison is chosen. ProjecToR ranges the load based on the utilization of the most congested link in the fat-tree, namely until it has reached 80%. In their simulator, this corresponds to over 5% of the flows not finishing. In our simulator, at 80% there are not yet notable consistently non-finishing flows. Only at around 99-100% highest port utilization (λ =14K) do 4.1% of the flows not finish. As these notions are coupled differently in our case, they are addressed separately to cover all bases:

- **Utilization until 80%:** A utilization of 79% of the most congested is reached at $\lambda = 8$K. This is thus the range of load $\lambda = [2\text{K}, 4\text{K}, 6\text{K}, 8\text{K}]$, which give a respective improvement of [7.1%, 14.2%, 21.4%, 29.3%] in average FCT.
- **Load when flows don't finish:** In our experiments, only at load $\lambda >$ 10K did flows consistently start to not finish in the fat-tree (at which point highest utilization reached 99-100%). At $\lambda = 10$K, the average FCT of the Xpander is 58.8% lower than that of the fat-tree.
- **Until 5% unfinished flows:** Due to TCP retransmission time-outs, beyond 10K load the flow completion times start to degrade steeply for the fat-tree. When 4.1% of the flows don't finish, at $\lambda = 14$K, the average FCT of the Xpander is 95.7% lower than that of the fat-tree.

The ProjecToR topology claims an improvement across all loads of 30-95% compared to the fat-tree. Thus, given the above comparison, the Xpander can achieve similar performance. When server up-links are added (fig. 4.3c and 4.3d), for both the fat-tree and Xpander, the bottlenecks are the server up-links. Xpander VLB still outperforms the fat-tree ECMP with a consistent margin of approximately 5-6% across loads $\lambda = [2\text{K-}14\text{K}]$. It is important to note that there is no compensation for the use of static ports instead of more expensive dynamic ports. This puts the static topology at a major disadvantage, yet is able to perform similarly.

4.3.7 Node All-to-all-fraction Experiments: Low Fraction (2.5-5%)

Main findings. The expander (at 67.5-80% cost of the full fat-tree) using VLB/HYBRID rivals the performance of the full fat-tree at low fractions of active nodes. This is caused by (a) the expander having higher ToR-ratio than the fat-tree, and (b) the expander through VLB/HYBRID being able to attain sufficient path diversity to avoid creating bottlenecks in the supporting network fabric. This path diversity is available because every cut is traversed by many links [27]. Expanders with ECMP performs significantly worse, because between the few active nodes only shortest path routing makes highly skewed use of links in the supporting network fabric. This forces the creation of bottlenecks. ECMP starts to perform better as the fraction increases. At increasing load, VLB starts to outperform HYBRID for low fractions. Decreasing the size of an expander (to decrease cost) significantly reduces the performance of shortest path routing (e.g. ECMP) for low active fractions, as the fewer active nodes reduce the much needed path diversity.

Traffic scenario motivation

Assume a scenario in which (1) each node has $d_{servers}$ server up-links and $d_{network}$ network links to the supporting network fabric, (2) the supporting network fabric applies perfect routing, and (3) each node does perfect flow spreading. In this scenario the following dichotomy would exist: (a) if $d_{network} \geq d_{servers}$ then only the server up-links are the bottleneck, and (b) if $d_{network} < d_{servers}$, the network links are the bottleneck. In actual scenarios however, it is not possible to have perfect network information. Thus, perfect flow spreading is not achievable nor can the underlying network fabric be perfectly used. It is interesting to investigate the effect of these actual conditions on the topologies under examination.

To approximate the idealized scenario, it is necessary to have experiments in which only a small fraction of nodes are active and communicate only between each other. For a non-discriminatory supporting network fabric, the small fraction of servers would result in the supporting network fabric to be underutilized as a whole on average. This under-utilization, under a routing approach that attains sufficient path diversity, would mean a small probability of flows sharing links in the supporting network fabric (coined *flow conflicts*). As the supporting network fabric is under-utilized, the highest probability of flow sharing is when a flow exits its source node or enters the destination node. Thus, the ToR-ratio (see section 4.3.5) will be highly indicative of performance.

Recent literature has indicated that only few ToR switches are hot in a data center. Ghobadi et al. [10] state that "*46-99% of rack pairs exchange no traffic at all, while only 0.04-0.3% of them account for 80% of the total traffic.*". Halperin

et al. [12] in a similar line of thought state that "*in every dataset [in their possession], over 60% of the matrices have fewer than 10% of their links hot at any time.*", indicating that there is a prevalence of hot-spots in data centers. Thus, performance achieved in skewed node-level distribution scenarios is indicative of the performance in an operational data center.

Workloads

A low all-to-all-fraction node traffic matrix is used, ranging from 2.5% (3 nodes FT, 6 nodes XP) to 5% (6 nodes FT, 12 nodes XP). A range of ALW and PAR workloads has been run for each of the fractions: 2.5%-ALW-{2K-20K}, 5%-ALW-{2K-29K}, 2.5%-PAR-{100K-500K}, 5%-PAR-{100K-500K} and 4%-ALW-{2K-29K}. Unless specified otherwise in figure labels, the 80%-cost Xpander is used.

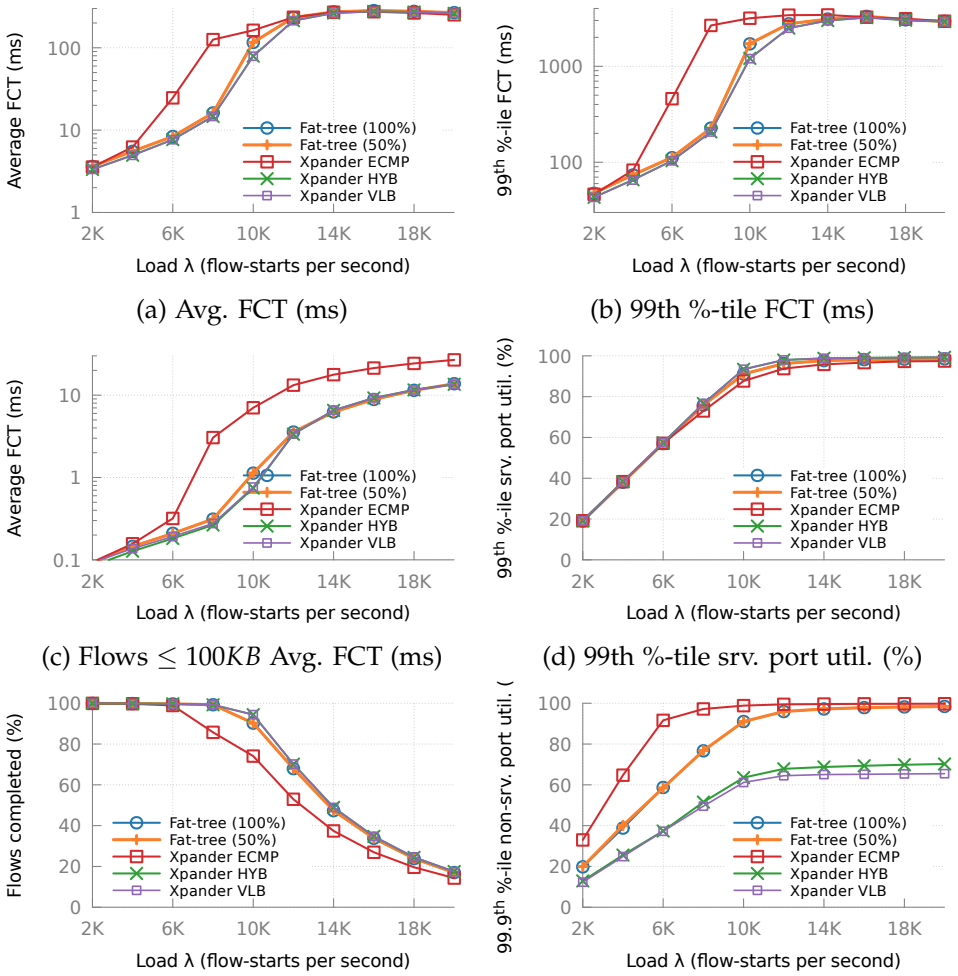
Expander analysis

In a low fraction, only few nodes are active. For an expander, this means that when the communicating nodes are chosen at random, there is a high probability there are only few shortest paths between node pairs (see section 3.3.2). Natural flow spreading through the traffic matrix of a low fraction all-to-all through the network is also not favorable, as each node only has few destinations. This means that to achieve good performance in an expander, routing approaches have to use path diversity beyond what shortest path routing can allow.

For 2.5%-ALW-{2K-20K}, ECMP quickly congests the shortest paths between the 6 active nodes which results in poor performance which is significantly worse in average FCT than HYBRID and VLB as is seen in figure 4.4a. Unlike the other configurations, ECMP starts to perform poorly before the server ports themselves become congested, as is seen at 6K/8K load in figure 4.4e, where long ($\geq 10MB$) flows start to not be completed at that point, in figure 4.4d, showing that the server ports at that moment are not yet congested, and in figure 4.4f, showing that a select few non-server (network) ports are becoming increasingly congested. Other (Xpander) configurations only start to become congested and not finishing flows consistently when the server port utilization is beyond 80%. This is explained by VLB/HYBRID guiding (long lasting) flows away from shortest direct paths, creating less competition on these paths which become congested in the ECMP configuration.

A similar condition is shown for the 2.5%-PAR-{100K-500K} traffic scenario. As the load increases, Xpander ECMP first reaches its network bottleneck before the other configurations experience the server up-link bottleneck as is seen in the average FCT in figure 4.5a. Another observation follows from observing the short flows average FCT in figure 4.5b, in which VLB outperforms both HYBRID and ECMP as there are so few paths between the select

group of nodes that with the large number of arrivals there is a high probability of contention. ECMP, although it uses shortest paths, is not able to achieve proper FCTs at relatively light load, even for small flows.



(a) Avg. FCT (ms)

(b) 99th %-tile FCT (ms)

(c) Flows $\leq 100KB$ Avg. FCT (ms)

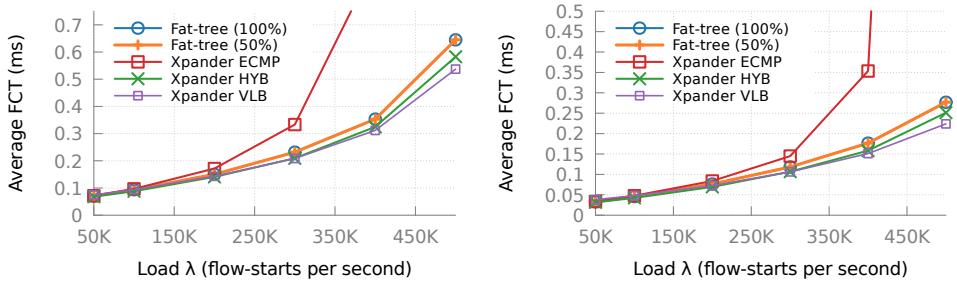
(d) 99th %-tile srv. port util. (%)

(e) $\geq 10MB$ Flow compl. (%)

(f) 99.9th %-tile non-srv. port util. (%)

Figure 4.4: Statistics for 2.5%-ALW-{2K-20K}

A 5% fraction significantly moves the moment of load at which Xpander ECMP would start to experience network congestion. Unlike at the 2.5% case where the network port congestion occurred significantly earlier (at $\lambda = 6$K in fig. 4.4f) than the server port congestion (at $\lambda = 10$K in fig. 4.4d), in the 5% case the network port congestion (see fig. 4.6d) scales practically simultaneously with the server port congestion (see fig. 4.6f). This shows that as the active fraction of nodes increases (in this case 2.5%→5%), ECMP attains more path diversity and thus is able to perform better. In the 5%



(a) Avg. FCT (ms)

(b) Flows $\leq 100KB$ Avg. FCT (ms)

Figure 4.5: Statistics for 2.5%-PAR-{100K-500K}

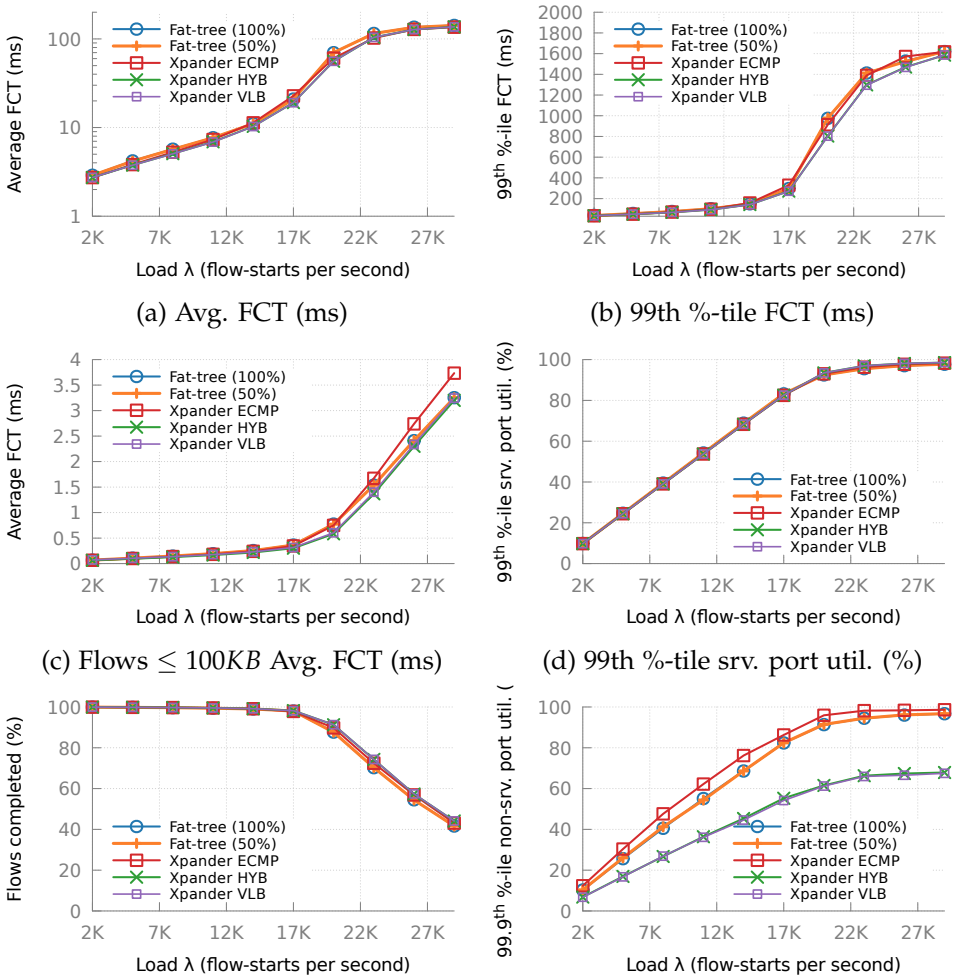
case, ECMP, HYBRID and VLB seem to achieve similar performance, as is seen in the average FCT for all (fig. 4.6a) and short (fig. 4.6c) flows. Although for ECMP the point of network congestion is earlier than that of its servers' congestion, which shows in its slightly worse performance beyond approximately 15K load.

The expander observations are summarized in observation 4.1.

Observation 4.1 *Under a skewed flow size distribution, with very few active nodes in an all-to-all, VLB and HYBRID perform better in an expander than ECMP overall as both route long-living flows over valiant paths avoiding congesting the few direct shortest paths available between the few nodes. As the fraction increases, ECMP starts to perform better as it attains more path diversity. With increasing load at a low fraction, VLB outperforms HYBRID, as HYBRID congests the few shortest paths too much due to the high number of flows starting that all have to transmit their first 100KB on the few shortest paths.* □

Fat-tree analysis

For a fat-tree, because the fraction is ordered from left to right pod, a low fraction of 2.5-5% will only select nodes in the same pod. This means that the full and oversubscribed fat-tree will perform identical, as oversubscription is done at the core layer. Only a small fraction of the links of the network are active. The in-pod links handle all the network traffic, all other available network capacity is untapped. As the full bottom-middle fabric in a pod is non-blocking, the server and non-server (network) port utilization scales perfectly with one-another, as is respectively seen for 2.5%-ALW in figure 4.4d and 4.4f, and for 5%-ALW in figure 4.6d and 4.6f. This is logical as the traffic is defined at a node-level, and each bottom node of a fat-tree has eight server up-links and eight network links to the middle layer.



(a) Avg. FCT (ms)

(b) 99th %-tile FCT (ms)

(c) Flows $\leq 100KB$ Avg. FCT (ms)

(d) 99th %-tile srv. port util. (%)

(e) $\geq 10MB$ Flow compl. (%)

(f) 99.9th %-tile non-srv. port util. (%)

Figure 4.6: Statistics for 5%-ALW-{2K-29K}

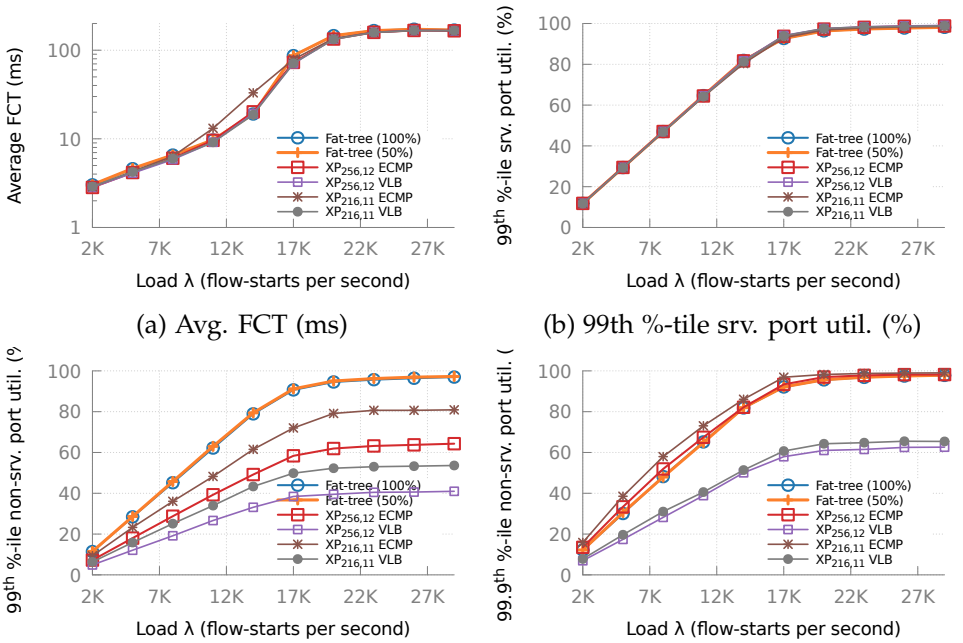
Does the performance of an expander rival that of a fat-tree for low all-to-all fractions?

The results leans towards a positive answer to the question for traffic matrices at node-level. In all cases of average FCT (see fig. 4.4a and 4.6a) and 99th %-tile FCT (see fig. 4.4b and 4.6b), fat-tree is outperformed by Xpander HYBRID and VLB (consistently approximately 6-10% better in average FCT before server congestion). The better ToR-ratio (see section 4.3.5) of the expander (1:1) avoids congestion better, whereas the fat-tree, both when the fraction is chosen in-pod or randomly, shows congestion occurring at the node-level (due to its ToR-ratio of 3:1). This ratio difference is illustrated in figure 4.2. Although a fat-tree is designed to provide a non-blocking switch

fabric, the routing is not able to perfectly spread the flow arriving at the ingress node to go outbound and at the egress node to go inbound. This is demonstrated by the 99.9th %-tile difference of non-server port utilization (see fig. 4.4f and 4.6f) for which fat-tree reaches quite high maximum port utilization levels for non-server ports in comparison to Xpander HYB/VLB and is across loads approximately equal to the 99th %-tile server port utilization (see fig. 4.4d and 4.6d).

How does cost of an expander influence its performance for low all-to-all fractions?

To investigate the effect of less network equipment, two topologies are compared: the $n = 256$, $d = 12$, $s = 4$ "large" Xpander topology with 1024 servers (80% of cost of full fat-tree), and the $n = 216$, $d = 11$, $s = 5$ "small" Xpander topology with 1080 servers (67.5% of cost of full fat-tree). The fat-tree is added as benchmark reference. As the main bottleneck in previous experiments were the server up-links, an equal amount of servers must be selected. For all the configurations a fraction of 4% is chosen, which translates to respectively 5 (fat-tree), 8 (small Xpander) and 10 (large Xpander) active nodes to accommodate a total of 40 active servers. As can be seen in figure 4.7a, the small Xpander performs similar to the large Xpander. The small and large Xpander perform similar for VLB, with a small difference of around 1-4% in average FCT (see fig. 4.7a). The ECMP routing approach however is significantly worse, fueled by the smaller amount of active nodes. As is seen in the 99.9th percentile of non-server port utilization in figure 4.7d: because of its lack of path diversity, the small Xpander reaches the bottleneck of its network links (on direct shortest paths) earlier. This results in its early steep incline in average FCT at 8K-17K, in which it has 5-65% worse FCT. At higher loads, the server up-links become the bottleneck for all (see fig. 4.7b), and thus the average FCT levels out (see fig. 4.7a).



(a) Avg. FCT (ms)

(b) 99th %-tile srv. port util. (%)

(c) 99th %-tile non-srv. port util. (%)

(d) 99.9th %-tile non-srv. port util. (%)

Figure 4.7: Statistics for 4%-ALW-{2K-29K}

4.3.8 Node all-to-all-fraction Experiments: Medium Fraction (19%)

Main findings. For expanders, with a significant large number of active nodes, short(est) path routing (such as ECMP) is the better routing approach than routing that consumes long paths (such as VLB). This is especially clear for workloads with an abundance of short flows, for which FCT is more reliant on round-trip time than for large flows. With a reasonably low fraction of active nodes (19%), which translate to a few pods in the fat-tree, a 50% core-oversubscribed fat-tree can be bottlenecked at sufficient load. This confirms the critique by [24], that oversubscribed fat-trees can perform poorly with only a fraction of servers active. The full fat-tree performs worse than any of the Xpander configurations (at 67.5-80% of its cost), as it is still plagued by the worse ToR-ratio.

Traffic scenario motivation

The results for low node fraction all-to-all in previous section 4.3.7 ended with the observation that as the fraction increased up to 5%, shortest path routing in the expander gained performance. This raises the question how much this performance would (relatively) improve with other higher fractions of active nodes. Moreover, this is a demonstration of the critique on fat-trees made by [24]. The critique (see section 2.1) states that an oversubscribed fat-tree performs poorly in a scenario where only a relatively small fraction of servers are active. For the medium fraction of 19%, exactly 3 out of 16 pods of the $k = 16$ fat-tree are active.

Workloads

A medium all-to-all-fraction traffic matrix is used of 19% (24 nodes FT, 48 nodes XP). A range of ALW and PAR workloads has been run for each of the fractions: 18.6%-ALW-{10K-100K}, 19%-ALW-{10K-100K}, 19.6%-ALW-{10K-100K} and 19%-PAR-{100K-500K}. Unless specified otherwise in figure labels, the 80%-cost Xpander is used.

Expander analysis

When a significantly large number of nodes communicate between each other, the flows over the available shortest paths between them will be less congested than for a small group of nodes at the same load. This is because (a) the amount of shortest paths increases quadratically with the fraction of active nodes, (b) nodes are spread throughout the expander fabric by design, and (c) an expander has a low average distance between nodes. Flow evaluation in section 3.3.2 showed that making use of paths larger than the shortest paths improves the performance less as the active fraction increases. In

practice, this suggests that routing techniques that consume more network capacity through longer paths will perform worse at higher fractions.

This is likewise evident in the packet-level simulations at medium active fractions. For 19%-ALW-{10K-100K}, before the server ports become the main source of congestion at around $\lambda = 60K$ (see fig. 4.8d), ECMP outperforms for all flows both HYBRID and VLB by respectively 7.8% and 8.1% at $\lambda = 10K$ and less significant 2.5% and 1.9% at $\lambda = 60K$. HYBRID outperforms both ECMP in the completion of small flows $\leq 100KB$ (see fig. 4.8c), achieving 1.1% slightly better average FCT at $\lambda = 10K$ and increasing to 2.7% at $\lambda = 100K$. This is due to its rerouting of long flows away from the shortest paths.

The 19%-PAR-{100K-500K} workload is light relatively speaking to the active percentage of 19%. For all configurations, all large flows are completed (see fig. 4.9e). Because of the low utilization, large flows are finished at high line-rate regardless of configuration. Thus, given the abundance of small flows, the small flows dominate the average FCT. At a slight cost for large flows (see fig. 4.9g), HYBRID marginally outperforms ECMP for short flows in average FCT (see fig. 4.9c) by rerouting large flows off the shortest direct paths. VLB consistently performs poorer than HYBRID and ECMP in average FCT for short, long and all flows. It unnecessarily increases flow sharing probability by using longer paths, as is seen in the higher average utilization of non-server (network) ports in figure 4.9h, and moreover having a larger round-trip time which is especially disadvantageous in the presence of an abundance of short flows.

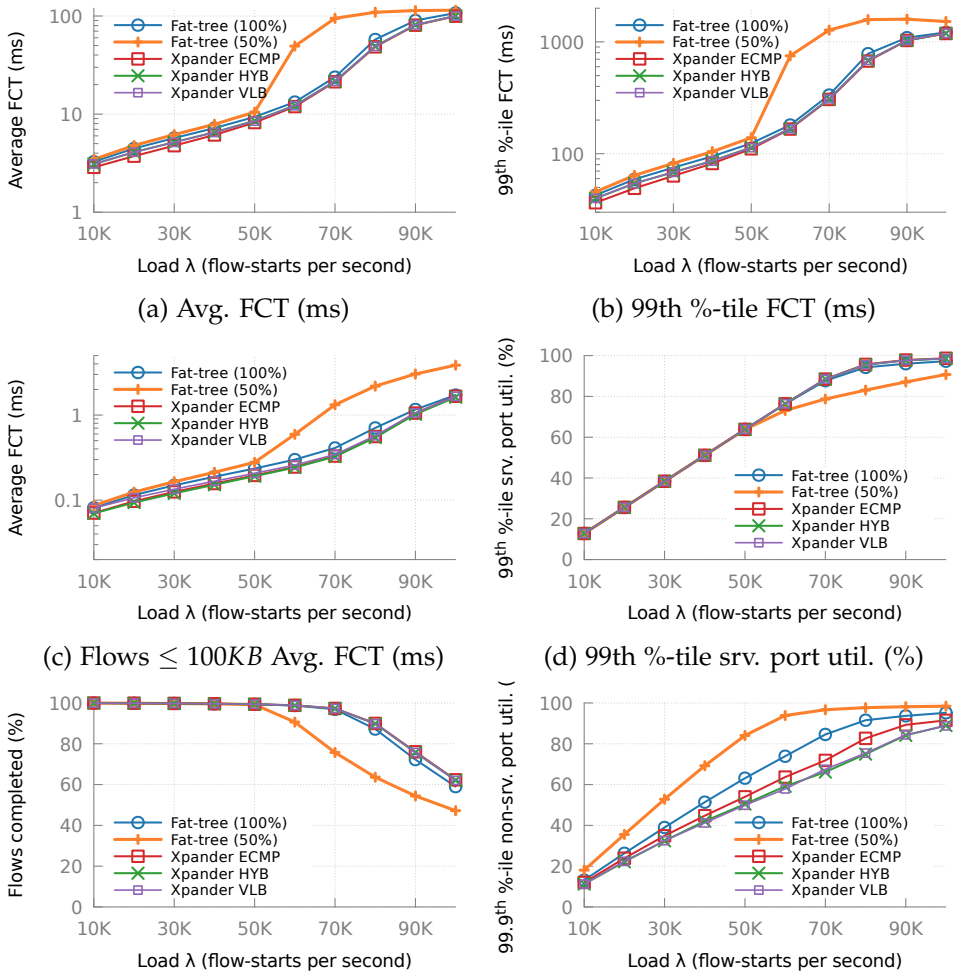
The expander observations are summarized in observation 4.2.

Observation 4.2 *Under a skewed flow size distribution, with many active nodes in an all-to-all, ECMP performs significantly better in an expander than VLB and slightly better than HYBRID. This is primarily due to the random placement of its communication partners results in shortest paths covering a lot of the available network capacity in the expander. HYBRID performs slightly better for short flows, due to the redirection of long flows over VLB paths after exceeding the threshold.* □

Fat-tree analysis

For higher ordered fractions, more cross-pod communication occurs for fat-trees. This scenario is significantly worse for β-oversubscribed fat-trees, as was shown in the flow evaluation (section 3.3.1): the core can only support a maximum of $(100 - \beta)\%$ inward and outward throughput to each pod. The pod however, covering the same amount of servers, can produce 100% traffic.

The 50%-fat-tree consistently performs worse than the 100%-fat-tree, achieving consistently a lower average FCT for both workloads. For ALW (see fig.



(a) Avg. FCT (ms)

(b) 99th %-tile FCT (ms)

(c) Flows $\leq 100KB$ Avg. FCT (ms)

(d) 99th %-tile srv. port util. (%)

(e) $\geq 10MB$ Flow compl. (%)

(f) 99.9th %-tile non-srv. port util. (%)

Figure 4.8: Statistics for 19%-ALW-{10K-100K}

4.8a), before it becomes congested after $\lambda \geq 60K$ (see next paragraph for explanation), it achieves 5.2% (at $\lambda = 10K$) and 10.8% (at $\lambda = 50K$) better average FCT. For PAR (see fig. 4.9a), it achieves 0.5% (at $\lambda = 50K$) and 4.7% (at $\lambda = 500K$) better average FCT. For PAR-{100K-500K}, the fat-tree is lightly loaded, as is shown in the low average non-server port utilization in figure 4.9h. The 99.9th percentile non-server port utilization (see fat-tree tail of fig. 4.9f) for this light highly skewed load is more unstable as for ALW (see fig. 4.8f), as only very few large flows occur and flowlet gaps of these few large flows can shift utilization significantly.

Most prominent in the ALW-{10K-100K} results is the clear decline in performance of the 50%-fat-tree at around 50-60K, shown by significantly higher

average FCT (see fig. 4.8a) and long flows not finishing (see fig. 4.8e). At this flex point the 99th %-tile of server port utilization reaches 65-75% mark (see figure 4.8d). This is in line with the knowledge of the 19% all-to-all fraction: assume that each server up-link in the fraction is active for 75% of the time. This means that the servers in one pod put in $0.75 \cdot 64$ units of flow at all times on average. Of these units put in, roughly $1/3$ of it is destined within the pod, but $2/3$ is destined inter-pod. This is thus $0.75 \cdot 64 \cdot 2/3 = 32$ units. This is exactly the maximum out-capacity that the pod has in a fat-tree with 50% of its core switches removed. This confirms the suspicion of [24], which predicts poor performance by a oversubscribed fat-tree when only a fraction of servers is active. In this traffic scenario, when a mere server fraction of 19% is active, the 50% oversubscribed-fat-tree is not able to offer the same level of performance as the full fat-tree.

Does an expander rival a (oversubscribed) fat-tree for medium all-to-all fractions?

For the heavy AWL workload, Xpander ECMP performed 10-15% better than the full fat-tree in average FCT (see fig. 4.8a). When relatively few pods (19% equals 3 pods) are active (although the fraction is called medium), it appears that expanders (ECMP/VLB/HYBRID) again profit vastly from their aforementioned better ToR-ratio of 3:1 for Xpander nodes versus 1:1 of the fat-tree bottom nodes. This is confirmed by observing that when choosing an *unordered* fraction of active nodes in the fat-tree, the same performance is achieved for the 50%-fat-tree as the 100% fat-tree (not shown separately in figures, equivalent to 100%-fat-tree line in fig. 4.8 and 4.9). This is even more visible in the light loaded PAR-{100K-500K} results, where even under light load the probability of conflicting flows and longer average path length (see more in answer (d) in section 4.3.9) make the fat-tree underperform relative to Xpander ECMP/HYBRID, and the gap to the longer average shortest path length Xpander VLB being closed with increasing load (see short flows average FCT in fig. 4.9c).

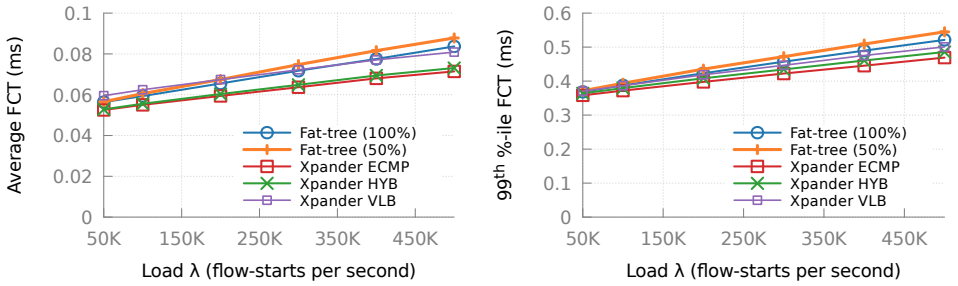
When the workload is dominated by an abundance of short flows (as for the PAR-{100K, 500K} workload), the round-trip time is quite important. This is shown in figure 4.9a for the FCT of small flows, in which the Xpander ECMP/HYBRID with its low average shortest path length (avg. 2.4) is able to significantly outperform the full fat-tree with its high fixed shortest path length (avg. 4) and Xpander VLB (avg. 4.8).

Concluding from the observation of the decline of the 50%-fat-tree once the bottleneck at the middle-core links is achieved, oversubscribed fat-trees perform significantly poorer in traffic scenarios with few pods communicating. It would be interesting to see the performance of the fat-tree when we shift from the concept of *hot nodes* to *hot servers*, which from the perspective of

fat-trees will presumably create a more spread use of the non-blocking fabric available and presumably overcome the ToR-ratio impediment (continue on to section 4.3.9 for the results).

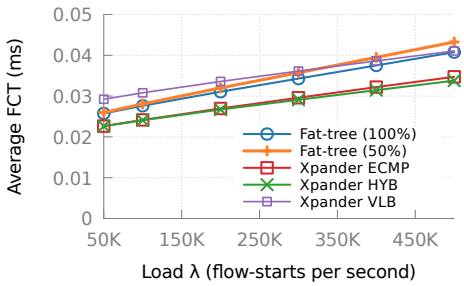
How does cost of an expander influence its performance for medium all-to-all fractions?

To investigate the effect of less network equipment, two topologies are compared: the $n = 256$, $d = 12$, $s = 4$ "large" Xpander topology with 1024 servers (80% of cost of full fat-tree), and the $n = 216$, $d = 11$, $s = 5$ "small" Xpander topology with 1080 servers (67.5% of cost of full fat-tree). The fat-tree is added as benchmark reference. As the main bottleneck in previous experiments were the server up-links, an equal amount of servers must be selected. For the large Xpander and fat-tree, an active node fraction of 19.6% (50 active nodes XP, 25 active nodes FT; 200 servers) was chosen, and for the small Xpander, an active node fraction of 18.6% (40 active nodes; 200 servers). In terms of average FCT, the small Xpander performs similar to the large Xpander with around a 2-4% difference (see fig. 4.10a) before congestion of the server ports becomes quite large at around $\lambda = 60$K (see fig. 4.10b). It is important to note that because of the smaller fraction of active nodes in the small Xpander, the congestion occurring at support fabric level is significantly higher. This is shown in the 99th and 99.9th percentile in figure 4.10c and 4.10d, in which the small Xpander configurations have significantly higher percentage of utilization than their large Xpander counterparts.

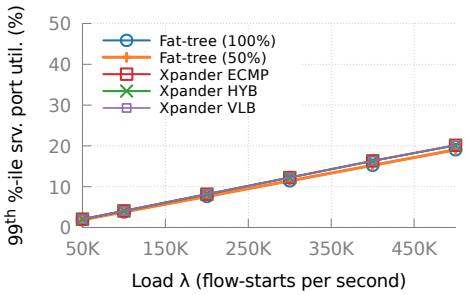


(a) Avg. FCT (ms)

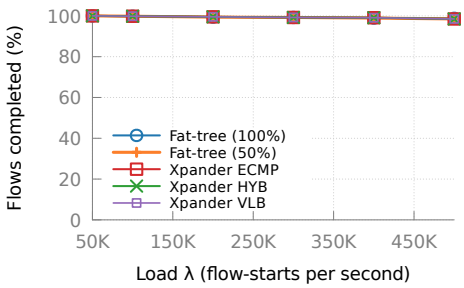
(b) 99th %-tile FCT (ms)



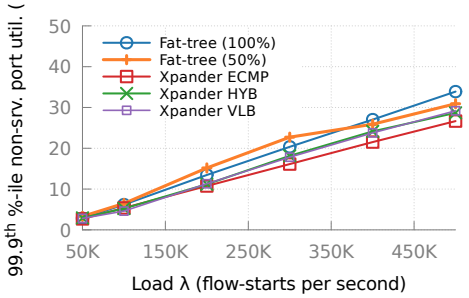
(c) Flows $\leq 100KB$ Avg. FCT (ms)



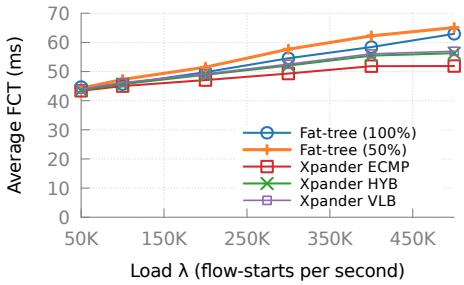
(d) 99th %-tile srv. port util. (%)



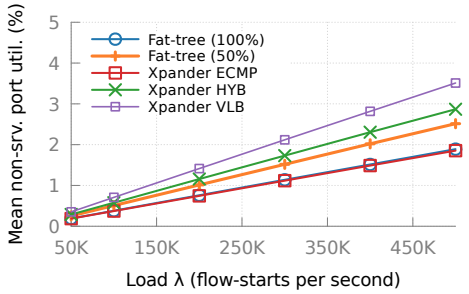
(e) $\geq 10MB$ Flow compl. (%)



(f) 99.9th %-tile non-srv. port util. (%)

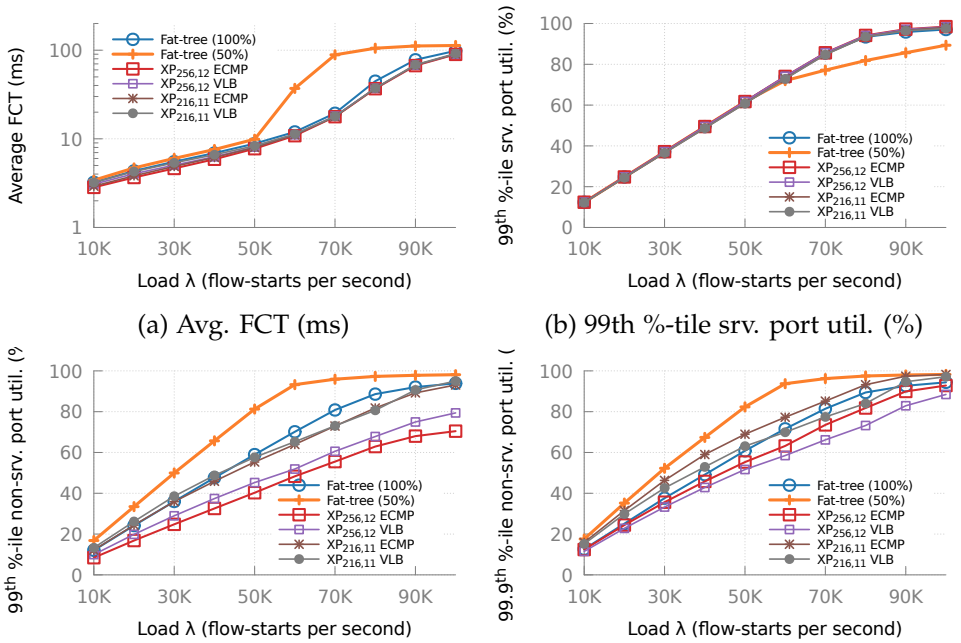


(g) $\geq 10MB$ Flows Avg. FCT (ms)



(h) Avg. non-srv. port util. (%)

Figure 4.9: Statistics for 19%-PAR-{100K-500K}



(a) Avg. FCT (ms)

(b) 99th %-tile srv. port util. (%)

(c) 99th %-tile non-srv. port util. (%)

(d) 99.9th %-tile non-srv. port util. (%)

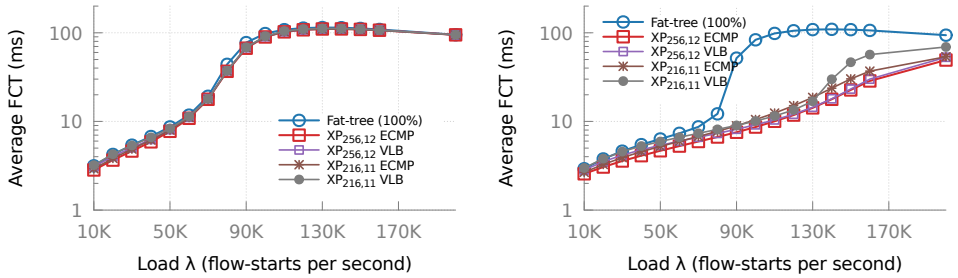
Figure 4.10: Statistics for 18.6/19.6%-ALW-{10K-100K}

How do all the configurations perform when they are all oversubscribed?

So far the only oversubscription seen was by removing core switches from the fat-tree. It interesting to see how the all topologies perform when more servers are added to the ToRs, thus to defer server bottlenecks becoming the limiting factor. Two scenarios are run with 19% all-to-all node fraction: (a) *Normal* (NOR), which assigns 8 servers per ToR to the full fat-tree, 4 for the large Xpander ($n = 256$, $d = 12$) and 5 for the small Xpander ($n = 216$, $d = 11$), and (b) *Oversubscribed* (OVR), which doubles the amount of servers, thus assigning 16 server per ToR to the full fat-tree, 8 for the large Xpander and 10 for the small Xpander. For all configurations (including fat-tree), the active ToRs are selected at random.

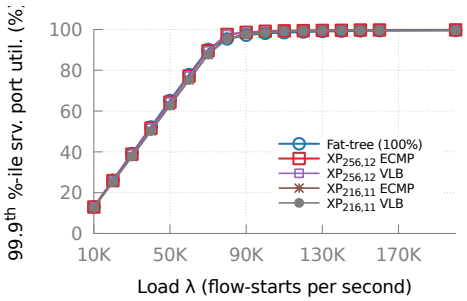
The normal fat-tree congests first in the oversubscribed case, as only a fraction of nodes are active and each node is limited to its eight network links. This is shown by the flattening of the 99.9th %-tile server port utilization at around 50-60% (see fig. 4.11d) for $\lambda = 90$K, at which point the non-server (network) port 99th %-tile reaches 90-100% for $\lambda = 90$K (see fig. 4.11f).

Another interesting observation is that around $\lambda = 90$K, for the small Xpander, VLB starts to perform slightly better than ECMP in average FCT, but at around $\lambda = 140$K, ECMP outperforms VLB again (see fig. 4.11b). A similar trend is seen for the large Xpander, in which ECMP does not surpass VLB but comes close at around $\lambda = 120$K. The explanation of this phenomenon is as follows: the fewer nodes are active, the fewer shortest direct paths there are, and thus the steeper the (active) network port utilization increases. This effect is shown in figure 4.11f, where at first the VLB has the more congested non-server ports, and at $\lambda \geq 70$K ECMP takes over the lead. Although at high load ($\lambda \geq 140$K) both 99th %-tile non-server port utilization is equal, VLB start to perform worse as it has higher probability of flow conflicts due to its longer path length. It is important to note that at this high load, much of the network capacity is highly used and thus completion of large flows starts to significantly drop (see fig. 4.11h). The longer path length played less of a role at lower load, as only 19% of the active nodes participate, thus conflict at ingress/egress due to valiant routing was lower.

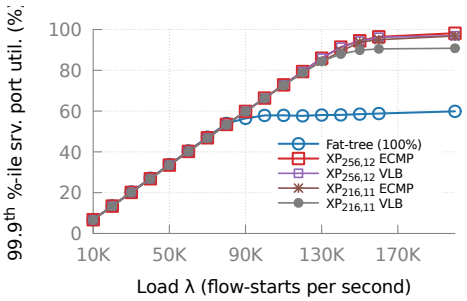


(a) NOR: Avg. FCT (ms)

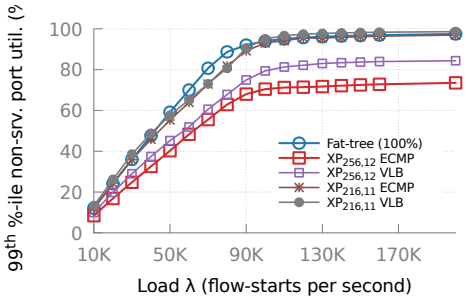
(b) OVR: Avg. FCT (ms)



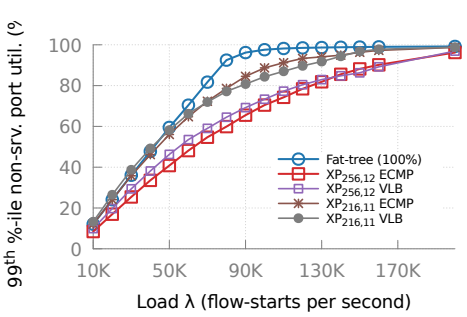
(c) NOR: 99.9th %-tile srv. port util. (%)



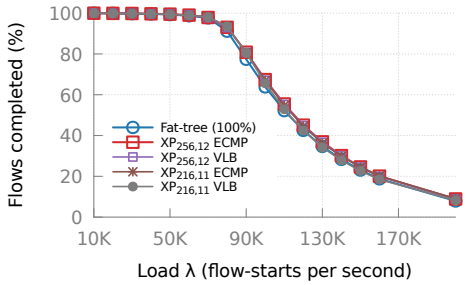
(d) OVR: 99.9th %-tile srv. port util. (%)



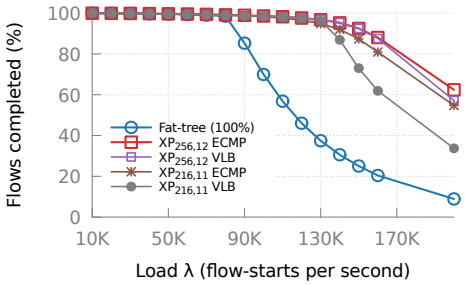
(e) NOR: 99th %-tile non-srv. port util. (%)



(f) OVR: 99th %-tile non-srv. port util. (%)



(g) NOR: $\geq$ 10MB flow compl. (%)



(h) NOR: $\geq$ 10MB flow compl. (%)

Figure 4.11: Statistics normal (NOR; left) and oversubscribed (OVR; right) for 18.6/19.6%-ALW-{10K-200K}

4.3.9 Pareto-skewed Experiments: Very Low Skewness ($\sigma 2$)

Main findings. In a very unskewed server traffic probability scenario, the expander (80% of full fat-tree cost) using shortest path routing (e.g. ECMP) performs best. In expanders, this shortest path routing outperforms approaches with longer paths (e.g. VLB) because these longer paths (a) consume more network capacity, and (b) increase the probability of flow conflicts. It is possible to achieve this performance for HYBRID by adjusting the Q threshold. The expander (67.5-80% of full fat-tree cost) rivals the full fat-tree's performance, due to the expander's shorter average path length despite having a lower amount of total network capacity.

Traffic scenario motivation

In previous sections, only node-level pair probabilities have been employed that are afterwards separated into server-level pair probabilities. In this section, a very unskewed Pareto distribution is used to assign communication probabilities directly to the servers. This which means that in general there will be only hot *servers* rather than hot *nodes* in the pair distribution. The skewness σ was chosen such that the resulting pair distribution was rather unskewed, resulting in a practical all-to-all traffic scenario. This scenario illustrates the performance when all servers have traffic demand, which would intuitively play into the cards of the fat-tree. The fat-tree is balanced and offers a theoretically non-blocking fabric to all its servers. It would be interesting to see how the expander, with its lower average shortest path length but also less network capacity, stacks up against it.

Workload and results

In figure 4.12, the ALW-{10K-100K} flow size distribution is used with Pareto-$\sigma = 2$ pair probability. It is a very unskewed pair probability, resulting in a practical all-to-all scenario. As ALW is very heavy with an average of 2.4MB and 10% of flows greater than 10MB, the FCT is dominated by the large flows. The performance of HYBRID is similar to VLB, as most flow size is routed over VLB paths when using a mere threshold of $Q = 100KB$ given the low skewness of the flow size distribution and high mean flow size. The 50% and 100% fat-tree perform better in average FCT (see fig. 4.12) than Xpander VLB/HYBRID, but are both outperformed by Xpander ECMP. Explaining this last observation properly requires answering the following questions consecutively.

(a) Why does the average FCT increase with load?

It is important to first note that the server port utilization, the amount of data put into the network, is the same for all configurations as can be seen in figure 4.13a. This is because each configuration is in a stable state, being

able to process all flows in time as is evident in the 100% flow completion of the large flows in figure 4.13b. The only difference is in **how quickly** the flows finish. The speed at which a flow finishes is dependent on whether it shares its current path with another flow. Thus, the higher the probability of conflicting, the more likely it is for a flow to share (for some time, dependent on the flowlet gap occurring and its effect) its path with other flows. There is significant load being put on the network, forcing flows to compete for links, as can be seen in the rising line in average FCT for each configuration in figure 4.12.

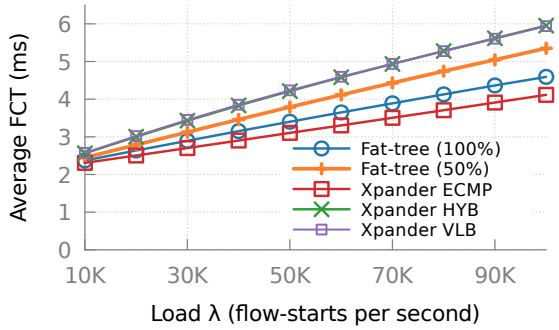


Figure 4.12: Avg. FCT ($\lambda = [10K - 100K]$, Pareto-σ2)

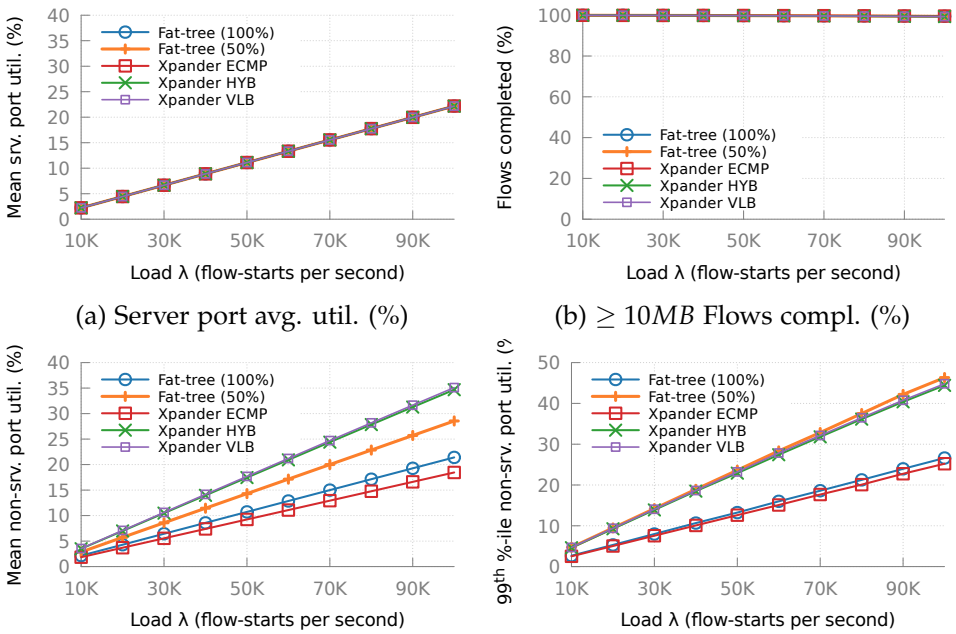
(b) Why does the 50%-fat-tree perform worse than the 100%-fat-tree?

The explanation for this has to lie in the absence of 50% of the middle-core links, as this is the only distinction between the two topologies. In the ALW-100K (highest load) case, the 50%-fat-tree has a mean utilization of 22.0% ($\sigma = 2.1\%$) of bottom-middle links and 41.6% ($\sigma = 2.6\%$) of middle-core links[4]. On the other hand, the 100%-fat-tree has an exactly equal mean utilization of 22.0% ($\sigma = 2.1\%$) of bottom-middle links, but a mean utilization of only 20.8% ($\sigma = 2.0\%$) at middle-core links. Using this information and the answer of (a), it can be inferred that the flows in the 50%-fat-tree encounters more congestion at the middle-core links and thus its flows will have a higher average FCT being forced to share links with a higher probability.

c) Why does Xpander ECMP beat Xpander VLB/HYBRID?

The flow evaluation results in section 3.3.2, specifically figure 3.2, showed that mere shortest paths can achieve maximum throughput per server in an all-to-all (which is equivalent to this scenario). As such, it gives an indication that performing VLB, expanding the path diversity, will not yield improving results. Following this hunch, looking at the non-server (so net-

[4]It is also possible to have within-pod communication, thus the total middle-core utilization is lower than that of the total bottom-middle utilization.)



(a) Server port avg. util. (%)

(b) $\geq 10MB$ Flows compl. (%)

(c) Avg. non-srv. port util. (%)

(d) 99th %-tile non-srv. port util. (%)

Figure 4.13: Statistics $\lambda = [10K - 100K]$ for Pareto-$\sigma 2$

work) port average utilization in figure 4.13c, it is clear that VLB roughly utilizes twice as much network capacity. On top of this, it also forces a 2x as long path length, which increases the probability of interference between flows. These observations in combination with the answer of (a) explain why Xpander ECMP outperforms Xpander VLB/HYBRID in this case. It is moreover possible to move the HYBRID threshold Q to different values to improve its performance closer to that of ECMP, as is demonstrated for $Q = \{100KB$ (default), $2.5MB$, $10MB\}$ in figure 4.16.

d) Why does the 50%-fat-tree beat Xpander VLB/HYBRID in the lower ranges and is beaten at higher ranges?

First, it is important to examine the average amount of travel a packet does. Using the calculation of [25], a random regular graph with $n = 256$ and $d = 12$ has an average path length of 2.4. Assuming acknowledgements are negligible, a single data packet in an all-to-all thus consumes roughly 4.8 units of network capacity for Xpander VLB. A similar analysis can be applied to the fat-tree in an all-to-all. Each inter-pod flow takes 4 units of capacity (bottom-middle, middle-core, core-middle, middle-bottom) and each in-pod flow takes on average takes 2 units of capacity (bottom-middle, middle-bottom). For $k = 16$ this means the average network units consumed is $((k-1) \cdot (k/2) \cdot 4 + (k/2-1) \cdot 2)/(k \cdot k/2 - 1) = 3.9$.

Second, now that the average network unit consumption is known, it is important to evaluate the network capacity. An expander with $n = 256$ and $d = 12$ has $256 \cdot 12$ network links. The 100% fat-tree has $k^3 = 16^3 = 4096$ network links, whereas a 50% fat-tree has $128 \cdot 8 + 128 \cdot 12 + 32 \cdot 16 = 3072$ network links with 50% of its core switches removed. It is furthermore important to note that the 50%-fat-tree has a severe bottleneck at the middle-core layer.

Because it is a practical all-to-all, there is very little chance of congestion occurring at the ingress/egress node. This is contrary to the previously tested all-to-all-fraction scenarios, in which the ingress/egress node was the main source of conflicting flows. In a fat-tree, once a flow has left its ingress node and reaches one of the middle layer nodes which in its turn offers k (100%-fat-tree) or $k/2$ (50%-fat-tree) path possibilities. Because it is an all-to-all, the 50%-fat-tree, although providing higher probability of conflicting flows at the middle-core links than the 100% fat-tree (see answer (b)), has less probability of two flows conflicting than the valiant routing taking place in Xpander VLB/HYBRID.

As load becomes higher, the probability of flows conflicting at the fabric level increases. This expresses itself for the 50%-fat-tree through the flattening of its mean server port utilization at 50% in figure 4.15a, as the middle-core links have become fully congested as is visible in the 99th %-tile of non-server port utilization in figure 4.15d. From that point of congestion on, the 50%-fat-tree consistently performs poorer (e.g. in average FCT in fig. 4.14) than the other configurations. After the 50%-fat-tree point of failure, the Xpander VLB/HYB follows next as their mean non-server port utilization reaches an average utilization of 87% ($\sigma = 2.04\%$) at $\lambda = 300K$ (see fig. 4.15c). Its failure afterwards to deal with the load is shown in the sharp decline of the completion of large flows in figure 4.15b.

e) Why does Xpander ECMP rival the 100%-fat-tree's performance?

Given that (following answer of (c)) the shortest path achieve a near-perfect coverage of the expander graph, and (following answer of (d)) the average network units consumed by a packet of an expander is 2.4, the expander with its 75% network capacity of 3072 can achieve a ratio of 3072/2.4=1280. The 100% fat-tree, with its average network unit consumption of 3.9 and a network capacity of 4096 can achieve a ratio of 4096/3.9=1050. This means that in this specific traffic scenario, the practical all-to-all, Xpander ECMP can outperform a 100%-fat-tree, as is shown in figure 4.12.

Performance of other Xpander configurations

The Xpander and fat-tree appear significantly at odds in this traffic scenario. Additional configurations of the Xpander were run to further understand the performance further, namely (a) by varying the threshold Q of HYBRID

to observe the VLB-to-ECMP trade-off, and (b) by adding an additional configuration $n = 216$ and $d = 11$, which has 58% of the network links of the full fat-tree compared to that of 75% of the $n = 256$ and $d = 12$ configuration.

First, different HYBRID threshold Q values are evaluated: 100KB (default), 2.5MB (approximately the avg. flow size of 2.4MB) and 10MB. The average FCT in both low ranges (see fig. 4.16a) and high ranges (see fig. 4.16b) declines with increasing value of Q, bounded by the VLB and ECMP ranges at respectively upper and lower bound. This indicates that dynamic adjustment of threshold Q could yield better performance for the HYBRID approach, research into which is deferred to future work (see section 6).

Second, an additional configuration of $n = 216$ and $d = 11$ (referred to as small Xpander) is run, supporting respectively $216 \cdot 5 = 1080$ servers. This is 56 more than the traditional configurations used until this point. At lower ranges, the smaller topology performs significantly worse for both VLB (e.g. 15.4% worse in avg. FCT at $\lambda = 50k$) and ECMP (3.9% worse in avg. FCT at $\lambda = 50k$) in figure 4.17a than the large expander. Because of the lower network capacity, as load becomes higher, small Xpander VLB experiences unsustainable non-server (network) port utilization earlier as is evident by its steeper rise with load in figure 4.17d and its order of magnitude difference in average FCT at high load (e.g. at $\lambda = 300K$) in comparison to large Xpander in figure 4.17b. Small Xpander ECMP has slightly lower mean server port utilization than its large counterpart due to its larger amount of servers, though has a significantly higher average non-server port utilization (see fig. 4.17d).

Though small Xpander ECMP performs similar to large Xpander ECMP and 100%-fat-tree at the highest load, the reason of decline in performance at the 300K-500K is for the small Xpander due to its non-server ports becoming congested, whereas for the large Xpander and 100%-fat-tree it is due to congestion of the server up-link ports. This is due to the small Xpander having more servers as is shown by consistently lower average port utilization in figure 4.17c and consistently higher average non-server port utilization in figure 4.17d. Similarly at lower ranges, the slight improvement upon fat-tree could be accounted to its higher amount of servers. This is an inherent problem of the difficulty to translate traffic scenarios to structurally different topologies. It is a point of improvement of the field in general, as is noted in future work (chapter 6).

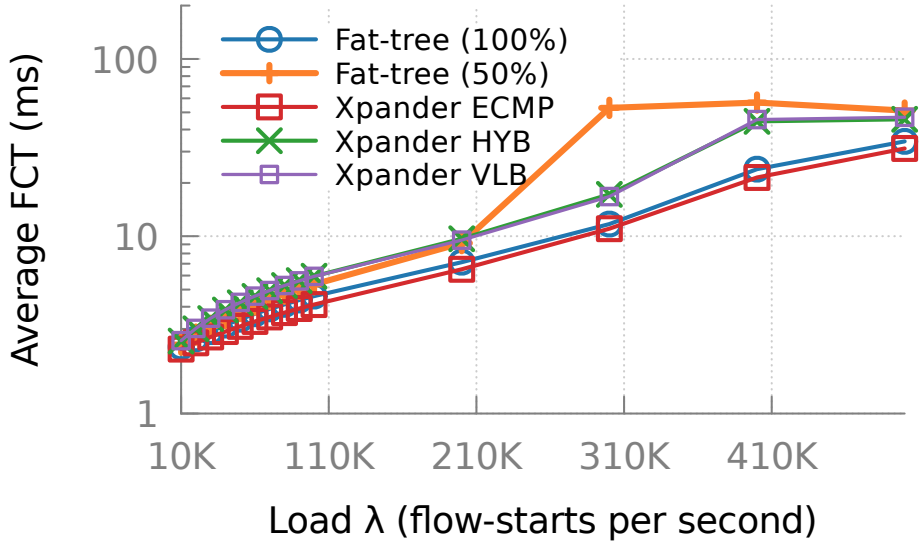
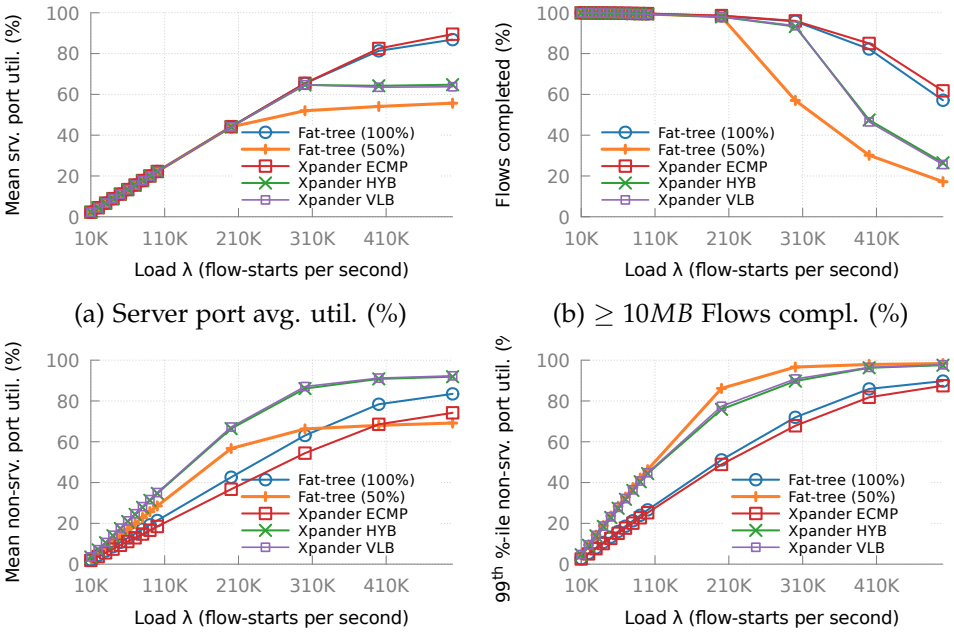


Figure 4.14: Avg. FCT ($\lambda = [10K - 500K]$, Pareto-$\sigma 2$)



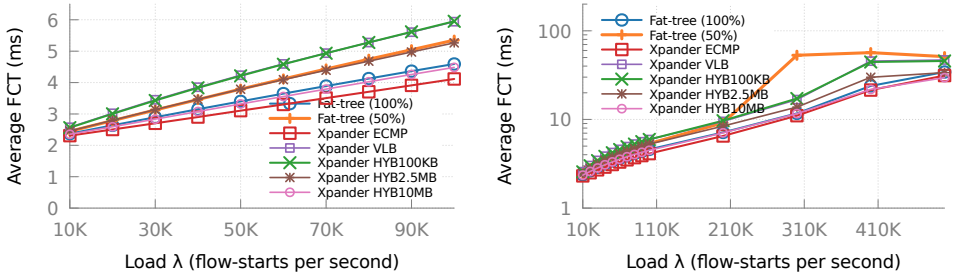
(a) Server port avg. util. (%)

(b) $\geq 10MB$ Flows compl. (%)

(c) Avg. non-srv. port util. (%)

(d) 99th %-tile non-srv. port util. (%)

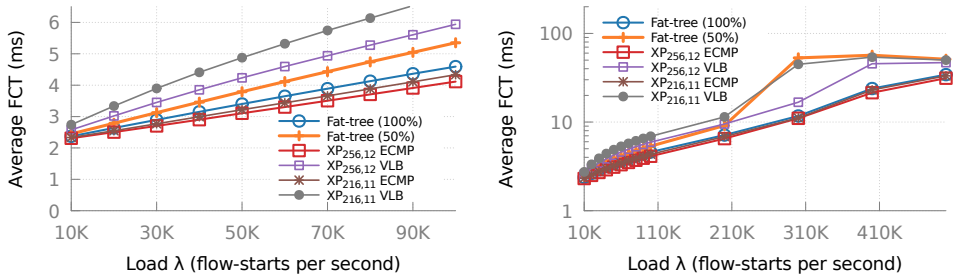
Figure 4.15: Statistics $\lambda = [10K - 500K]$ for Pareto-$\sigma 2$



(a) Avg. FCT (ms)

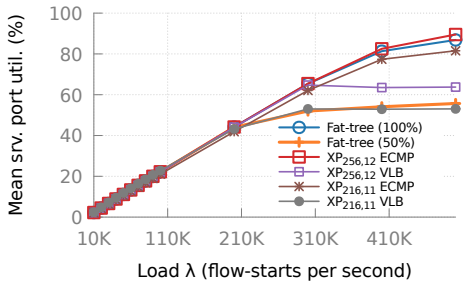
(b) Avg. FCT (ms)

Figure 4.16: Additional HYBRID threshold $Q = \{100KB,\ 2.5MB,\ 10MB\}$ boundaries for Pareto-$\sigma 2$

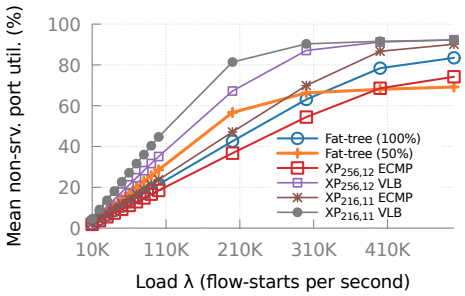


(a) Avg. FCT (ms) (low range)

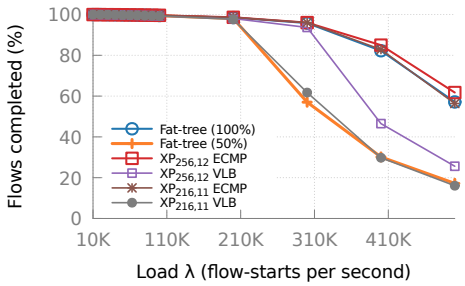
(b) Avg. FCT (ms) (full range)



(c) Server port avg. util. (%)



(d) Non-server port avg. util. (%)



(e) $\geq 10MB$ Flow compl. (%)

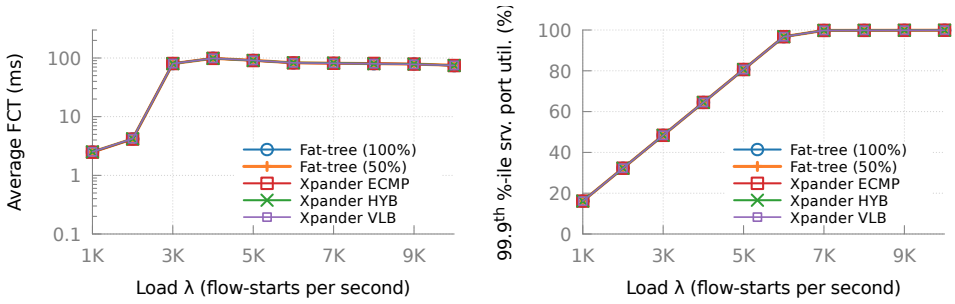
Figure 4.17: Additional Xpander (addition of $n = 216, d = 11$ Xpander) configuration for Pareto-$\sigma 2$

4.3.10 Pareto-skewed Experiments: High Skewness (σ0.97)

Main findings. Controlling skewness subtly through the σ in the Pareto pair distribution turned out to be difficult. At high skewness, with very few servers having communication probability, all configurations perform equal.

In an attempt to explore high skewness at the server-level, a Pareto server pair distribution was run with shape $\sigma = 0.97$. A load of ALW-{1K-10K} was used. As was expected, given the high network capacity of the supporting network fabric, the server ports of the few active servers were the main bottleneck as is shown in the 99.9th percentile of server port utilization in figure 4.18b. The average FCT, as congestion was only happening at the server level, were practically indistinguishable as is demonstrated in figure 4.18a.

This avenue of traffic scenarios turned thus out to not be an easily controllable method, due to the sensitivity of σ to small value changes and the lack of straight-forward interpretation. Thus, in the section 4.3.11, all-to-all-fractions are introduced again, but now at the server-level.



(a) Avg. FCT (ms)

(b) 99.9th %-tile srv. port util. (%)

Figure 4.18: Results for Pareto-σ0.97

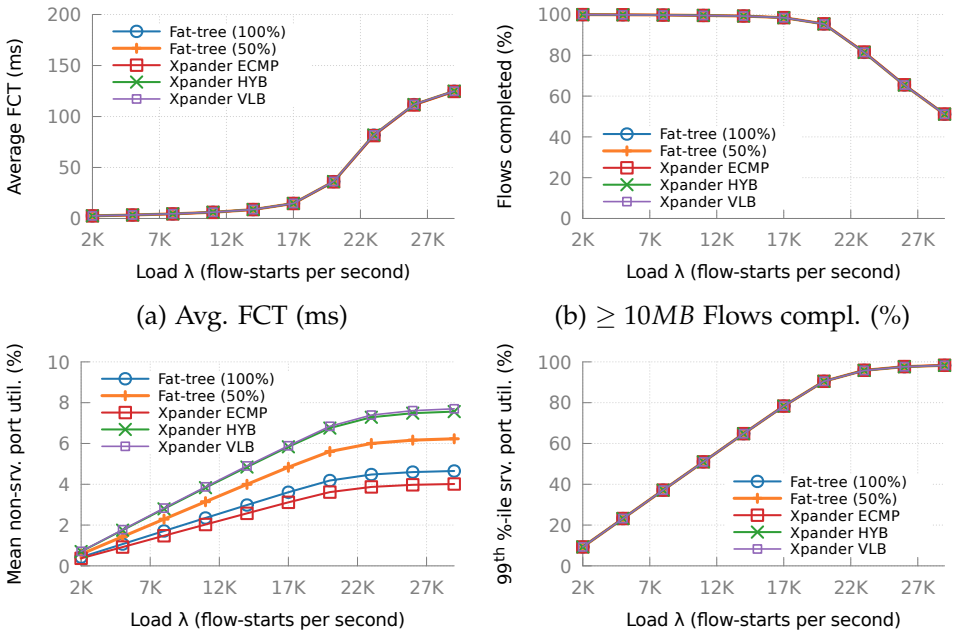
4.3.11 Server All-to-all-fraction Experiments

Main findings. When very few servers participate, the performance among configurations is equal. As the fraction of active servers increases the trend similar to that of the Pareto-σ2 is seen (see section 4.3.9): it is dominated by the probability of flow conflict in the supporting network fabric of each configuration. Best performance is achieved in the expander (80% cost of full fat-tree) using shortest path routing.

To lessen the effect of the ToR-ratio difference, and to not overload single server ports by introducing high skewness as was done in Pareto-σ0.97, another set of experiments was done using an all-to-all-fraction of servers. A fraction of 2.5%, 5%, 30% and 70% was run. The workloads AWS-[2K-29K] and AWS-[10K-500K] was used.

Low server fraction performance (2.5%-5%)

For the low fractions of 2.5% and 5%, all network fabrics were able to fully support the load put on by this randomly selected fraction of servers. This is evident (only 5% shown, same trend for 2.5%) by the near-equal FCT (see fig. 4.19a) (some negligible differences due to varying RTT), the equal amount of flow completion (see fig. 4.19b), and the same point of failure caused by the server up-link ports becoming congested by seeing the direct correlation of the flow completion in fig 4.19b and the 99th percentile server port utilization in figure 4.19d. The mean non-server port utilization in figure 4.19c shows how much across the spectrum each configuration needed to use its network fabric. It is important to note, the 50%-fat-tree's port utilization is skewed and high for the middle-core links (it's average is thus low).



(a) Avg. FCT (ms)

(b) $\geq 10MB$ Flows compl. (%)

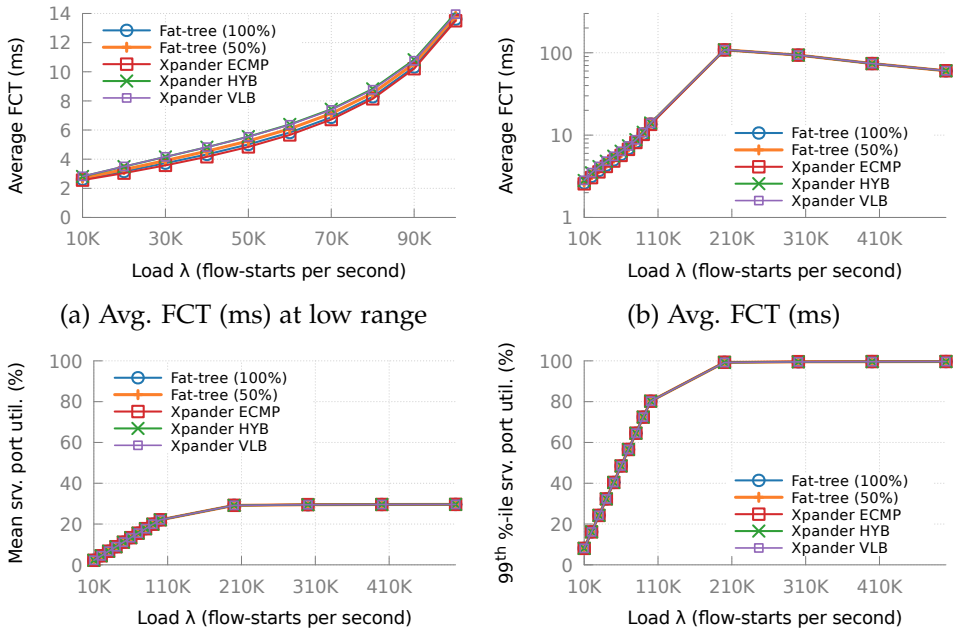
(c) Avg. non-srv. port util. (%)

(d) 99th %-tile srv. port util. (%)

Figure 4.19: Results for server all-to-all-fraction of 5%

Medium server fraction performance (30%)

The 30% indeed showed performance in line with observations at the lower all-to-all server fraction and the practically all-to-all Pareto-σ2 workload. At lower ranges, the Xpander VLB/HYBRID performed equally bad, followed by the 50%-fat-tree, the 100%-fat-tree and as best performer the Xpander ECMP (see fig. 4.20a). At fractions of active servers rather than nodes, the ToR-ratio is of lesser importance because the servers are chosen randomly across ingress nodes. What is dominant is the probability of flow collision at the support network fabric level, of which the results are in line with the answers to questions posed in section 4.3.9 with respect to average path utilization and network capacity in the configurations. At higher ranges, the selected 30% random fraction of servers reaches its limit, as is seen in the average server port utilization flat-lining at 30% after around $\lambda = 200K$ in figure 4.20c and the 99th percentile of server port utilization approaching its ceiling in figure 4.20d. After this moment, flows in every configuration are bottlenecked at the same place and thus perform equally bad as is seen in average FCT in figure 4.20b. The server fraction of 30% is not able to load any of the supporting network fabrics to its limits.



(a) Avg. FCT (ms) at low range

(b) Avg. FCT (ms)

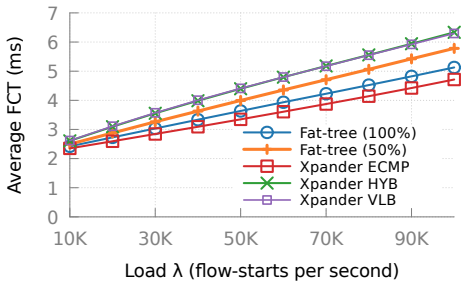
(c) Avg. non-srv. port util. (%)

(d) 99th %-tile srv. port util. (%)

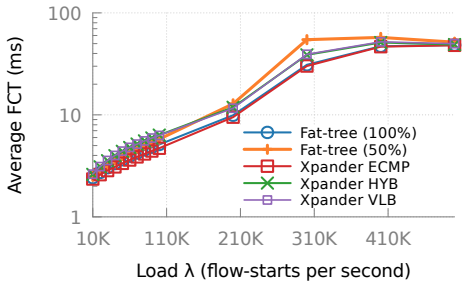
Figure 4.20: Results for server all-to-all-fraction of 30%

High server fraction performance (70%)

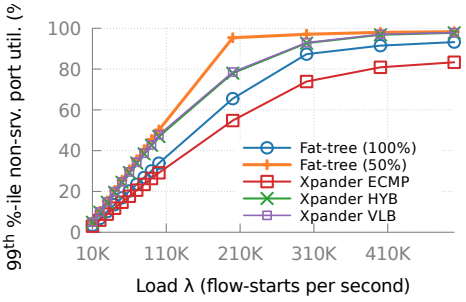
The 70% all-to-all server fraction results showed a trend similar to that of the Pareto-σ2, as they both are both a very un-skewed server traffic pair distributions. The same trend at the lower ranges as for 30% is observed, as can be seen by comparing the average FCT in previous figure 4.21a to figure 4.21a. As anticipated, as was also the case in the Pareto-σ2 traffic scenario, the 50%-fat-tree's middle-core layer becomes congested at high enough load, before the server up-links become congested. The server port congestion causes the later drop in performance for the other configurations. This is shown by 50%-fat-tree's quicker increase in its 99th percentile non-server port utilization in figure 4.21c and the quicker flattening of the 99th percentile server port utilization in figure 4.21d. This earlier congestion at supporting network fabric level causes its average FCT to perform worse than Xpander VLB/HYBRID, as is shown in figure 4.21b.



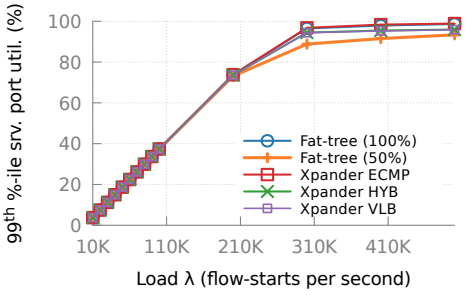
(a) Avg. FCT (ms) at low range



(b) Avg. FCT (ms)



(c) 99th %-tile non-srv. port util. (%)



(d) 99th %-tile srv. port util. (%)

Figure 4.21: Results for server all-to-all-fraction of 70%

4.4 Expander routing deployment analysis

Main findings. Deployment of the routing strategies requires significant adjustments to packets and switches in the network. Valiant load balancing (VLB) requires little additional header information, whereas the more complicated source routing (SR) has high demands and might require a custom encapsulation. Each ingress switch must maintain state to decide to which valiant node or on which source path to send a packet. In order for VLB or SR to work in practice, it is necessary to continuously update switches with network state changes. Flowlet support and tracking Q in the HYBRID approach is quite feasible, as it would each only require the addition of a single table at each ingress switch.

Although the deployment of an expander does not nearly require as complicate of a setup as for a dynamic topology, it is does have significant hurdles to overcome. To attain proper performance in skewed traffic cases, (e.g. the 2.5-5% all-to-all node fraction in section 4.3.7), it requires path diversity beyond what shortest paths can offer. As such, maintaining mere next-hop-to-destination ECMP tables on the switches is not sufficient to achieve optimal performance in all cases. Any logic presented in this section could be implemented using OpenFlow [20].

Supporting valiant load balancing

The proposed answer in this work to obtain the necessary path diversity at routing is valiant load balancing (VLB). Adding support for VLB is simplistic. First, either via an encapsulation or by using the IP Options field (96-bit capacity), additional data to the packet is added: (a) the 32-bit address of the valiant switch (VLB_DEST), and (b) a 1-bit flag $PASSED\_VLB$ indicating whether the packet has passed the valiant node. The switch logic would be the following:

Algorithm 3 Switch VLB packet handling

```
1: procedure RECEIVE(packet)
2:     passed ← packet.VLB_DEST == this.id || packet.PASSED_VLB
3:     packet.PASSED_VLB ← passed
4:     if passed then
5:         forward(packet, packet.TRUE_DEST)
6:     else
7:         forward(packet, packet.VLB_DEST)
8:     end if
9: end procedure
```

Either the ingress switch, or the server itself, has to modify the IP header

or add an encapsulation when a packet is being transmitted to the network. Either choice would require it to have knowledge about the network, specifically which addresses are valiant node candidates. It would be more natural for this to be kept at the ingress switch, which is responsible for the network in general. Moreover, if an update in network state needs to be distributed, it would not need to be propagated to all servers. The egress switch in its turn removes the encapsulation or added IP Options.

Supporting the hybrid proposal

As was evident in the work, only shortest paths (ECMP) or only long paths (VLB) was not capable of achieving good performance in all cases. The hybrid proposal, which stated that if the flow has sent more than threshold Q, requires additional state. The state would need to be kept at the ingress switch. The switch is able to find out which flow a packet belongs to by combining the source address, destination address, source port and destination port into a hash (similar to ECMP). The switch needs to keep an list of $[flow\_hash, size\_sent]$. When a packet come in from one of its servers, it increments the $size\_sent$ for the corresponding $flow\_hash$. It can either set the initial value of the $PASSED\_VLB$ flag to $true$ or not add the encapsulation/Option at all if and only if the $size\_sent$ is smaller than threshold Q. It should remove a flow entry from the list, when the flow has not been seen for some time (time-out based).

Supporting source routing

The path diversity required by the expander could also be achieved through a K-paths routing mechanism, which selects paths based on some criteria. K-paths routing requires source routing to be supported in the network. Source routing is the practice of the source planning the full path of the packet when transmitting. The network guarantees that the path is followed exactly: the switch does not make any routing decisions by themselves.

Similar to supporting valiant load balancing, source routing as well can either use an encapsulation or modification of the packet Options. The main difference to VLB at packet level is that encoding a full path takes up significantly more space. The packet needs to contain all routing hops (HOP_LIST) and a counter to indicate at which hop it is currently ($HOP\_COUNTER$). Routing steps can be encoded naively by defining full network addresses (each 32-bit). A more intelligent strategy is to associate, for each outgoing link of a switch with degree r, the identifiers $[0, 1, ..., r-1]$. This would allow each hop to be encoded in $log_2(r_{max})$ bits. The path encoding is pushing the limit of the IP Options (96-bit limit), but could also be added to the TCP Options (320-bit limit) if space is available.

An ingress switch, upon receiving a packet from one of its servers, inspects the destination address and chooses one of the available paths to that des-

tination. Afterwards, it modifies the packet or adds the encapsulation, and sends it along the first hop. Each in-between switch selects to which port to forward the packet by looking up the $HOP\_COUNTER$ in the HOP_LIST, after which it increments the $HOP\_COUNTER$.

Encapsulation selection

Generic Routing Encapsulation (GRE): a common purpose encapsulation header, such as GRE [8], adds an encapsulating 20-byte IP header over the packet, as well as a 4-byte GRE header with 2 bytes of space for the application to define. This would lend itself easily for valiant load balancing, as the encapsulating 20-byte IP header is set to the VLB_DEST; at the valiant node the encapsulation is removed and the payload packet is transmitted "normally". It is less suited for supporting source routing, because, besides the additional IP Options, there is little space available for self-defined data in its header.

Network Virtualization: network virtualization techniques, such as NVGRE [9][5], offer the ability to decouple physical addresses from the workload's addresses. The major advantage (as noted by RFC7637) is that a data center operator can move workloads around, without having to reconfigure the network every time. These virtualization encapsulations however are focused on merely this address translation, and do not offer any explicit space for self-defined routing variables. For this reason, they are not useful in supporting more complicated routing state.

Custom: instead of hacking existing (encapsulation) headers to encode source routing information, it is also possible to create a custom source routing header.

Unaddressed routing difficulties

Other challenges that are not addressed in this work, are regarding non-modeled network phenomena such as link failure or switch failure. This could have a significant impact on routing and its resilience. Both the VLB and source routing proposal require the network to maintain additional state. For example, VLB requires the available selection of valiant nodes, whereas source routing requires full knowledge of available paths. Link and device failure will affect the ability to reach switches, thus forcing the network to broadcast new updates on the network state continuously to prevent packets from becoming undeliverable. Routing techniques could add redundancy in their design to aid this effort. SlickFlow [23] for example proposes a path recovery procedure for source routing, based on a construction of primary and alternative paths. There is significant indication by [27] that

[5]NVGRE is proposed in RFC 7637; https://tools.ietf.org/html/rfc7637 (retrieved 15-03-2017)

expanders are, due to their structure, near-optimal in resiliency to failures and that there is graceful performance degradation relative to the fail rate of links (using flow-level experiments in an all-to-all scenario). This finding bodes well for the ability of routing to recover when parts of the network fail.

Supporting flowlets

In order to support flowlets, a table that tracks the flowlet gaps is required, coined the Flowlet Table by [2]. The Flowlet Table maps a flow to the last-sent time, as such that the switch is able to distinguish when a flowlet gap occurs. This would again become part of the state a switch has to maintain. [2] states this is quite feasible by stating that "*tracking 64K flowlets is feasible at low cost*" for the "*4500 virtualized and bare metal hosts across* $\sim$ 30 *racks of servers*" its production cluster consists of.

Interplay of congestion control and routing

In this work, the endpoint's transport layer (congestion control) is agnostic with respect to the paths selection performed by the network's switches (routing). More in-depth investigation into the interplay between these two mechanisms is necessary, and is deferred to future work. An unconventional solution would be to apply a universal packet scheduler to the expander to bypass this interplay all-together, such as FastPass [21]. Given the network complexity of an expander, this would put extra strains on the path selection logic. This is in contrary to the topology discussed in the paper, which focuses on tree-like topologies. It would of course inherit all additional requirements such as additional (arbiter) devices, and NIC modification.

4.5 Discrete Packet Simulation Conclusion

The discrete packet simulations have been run over a wide variety of traffic scenarios and topologies. The importance of modeling the server up-links was demonstrated in the comparison to a scenario equivalent to that of ProjecToR [10]. It was found that without modeling the server up-link, similar performance gains (7-96% achieved vs. 30-95%) were shown for the expander topology in comparison to the fat-tree. With the addition of up-links, these massive gains were replaced with only a small less-significant gain of approximately 5-6%. Moreover, this was achieved while putting the expander at a disadvantage by offering no compensation for the use of cheaper - static - ports.

A wide variety of traffic pair distributions and two flow size distributions were employed. The traffic pair distributions were modeled based on recent dynamic topology research [10, 12], which highlights the prevalence of hot spots in actual data centers. This was combined with critique of the fat-tree by [24] to create a fairly representational spectrum of traffic scenarios.

In accordance with the flow evaluation, it was found that for expanders, at low fraction of randomly chosen active nodes, only using shortest paths (ECMP) is not sufficient. As the active amount of nodes increases, so does the performance of ECMP improve in the expander as it covers more of the network capacity. Valiant load balancing (VLB) shows great promise in the low fraction of active nodes, making use of the network capacity through its path diversity. Conversely, at higher fractions, valiant load balancing performed worse as its probability of conflicting flows increased significantly more due to its higher path length than ECMP. An in-between variant, HYBRID, which switches from ECMP to VLB at a threshold Q is proposed and shows itself as potentially a combination of the best of both.

As expected by the flow evaluation, once the load is too high and the servers up-links do not throttle flows, the 50%-core-oversubscribed-fat-tree's performance drops. The expander topology at 67.5-80% of its cost was able, using at least one of its configurations, to rival the 100%-fat-tree in all traffic scenarios. Three factors had significant positive impact for the expander topology. First, the higher ToR-ratio (3:1) of the expander in comparison to that of the fat-tree (1:1). This is because its ingress/egress nodes are actually part of the supporting network fabric. The better ratio decreased the probability of a flow conflicting when exiting the ingress node at the start of its journey and entering the egress node at the end of its journey. Second, the shorter average path length enabled efficient usage of its network capacity. This efficient usage even overcame the lesser availability of network capacity as the expander only used 67.5-80% of the full fat-tree's equipment. Third, the property of an expander that every cut in the network is traversed by many links [27] which enables path diversity.

A routing deployment analysis has shown that the proposed solutions for routing in an expander are possible. Valiant load balancing is possible with small changes to the packet's header, whereas source routing would most likely require full encapsulation. Keeping track of flowlets gaps and threshold Q for the HYBRID proposal is feasible. It is necessary to continuously update the network knowledge of switches, in order for the routing to remain functional. Thorough feasibility analysis, evaluation of more state-requiring routing approaches, investigation into the interplay between congestion control and routing, and challenges regarding link or device failure, have been deferred to future work.

Chapter 5

Discussion

The work is a humble continuation of a series of pivotal research into what is now coined *expander* network topologies. Jellyfish [26] introduced the concept of employing the random regular graph as a data center network fabric. It noted that there were still significant challenges in routing, physical layout, and wiring. [25] presented an upper-bound on network throughput under uniform traffic patterns, and showed that the Jellyfish topology achieves close to optimal performance. [15] broadened the spectrum of traffic matrices, and developed a framework to benchmark of network topologies. This benchmark was extended and is the key mechanism enabling the flow evaluation in this work. Xpander [27], driven by the results of previous studies showing high performance of random regular graphs, harmonized the space by finding that a random regular graph is an *expander* with high probability. [27] asserted the many desirable properties an expander possesses: near-optimal throughput, high robustness to traffic variations, high resiliency to failures, incremental expandability, and short average path lengths and diameter. It furthermore addressed the earlier mentioned challenges in physical layout and wiring through a preliminary analysis, but does not yet "*capture all the intricacies of building an actual data center*". Most importantly, [27] addressed the challenge of routing in an expander, by employing K-shortest paths and MPTCP congestion control (earlier suggested as well by [26]). It uses permanent flows and measured at the aggregate level of flow throughput. It showed that near-optimal throughput was achievable in the traffic matrices tested.

In continuation of the routing exploration, this work re-evaluated and confirmed the critique of [24] using an extended version of the benchmark made by [15]. The main contribution of this work to the landscape are its packet-level experiments. The experiments enabled the comparison against the state-of-the-art dynamic topology ProjecToR [10] and the traditional fat-tree [1]. The results indicate that in fact, the expander is able to rival the

performance of either. The work offers an insight into how routing should be done in practice in an expander. More specifically, it enables the projection of performance under a select range of relevant traffic scenarios for two routing approaches: when using routing that (a) uses short paths with little path diversity (ECMP), and (b) that uses longer paths which offer more path diversity at the cost of network capacity (VLB).

The discrete packet simulator *NetBench* has been written as meticulously as possible, despite the size of its undertaking. Unlike the flow optimizer *TopoBench* which employs perfect traffic engineering, the amount of parameters and logic possibilities are plentiful. The packet-level results in chapter 4 are heavily dependent on these choices. A constant effort was made to select as universally applicable parameters as possible, and to observe logs to correspond with performance claims of used techniques (e.g. for DCTCP, see appendix E.1). To verify its results, it would be especially useful to compare it to simulations run on other existing packet simulators or to a (small-scale) deployment.

Many challenges still lay ahead of the expander topology. These include further investigation and evaluation of more workloads characteristic of data centers, and physical deployment challenges (e.g. wiring, spatial planning). In the area of routing specifically, it would be interesting to (a) investigate practical situations such as device crashes and link failure, (b) explore the space of routing schemes (e.g. ECMP, VLB, MPTCP) and its parametrization (e.g. RTO, flowlet gap), and (c) examine the interplay between congestion control and routing. Moreover, as it is the contender for data center network fabric of the future, a more thorough understanding of the ability of dynamic topologies is necessary. This envisioned framework of understanding should incorporate routing optimality, buffer analysis, and deployment feasibility. The completion of these tasks will increase the understanding of data center network fabrics, and will shape the way data centers should be designed in order to achieve the best performance.

Chapter 6

Future Work

The work, limited by time, has been written while endeavoring to cover as much of the scope as possible. Given the vast breadth of directions in which a data center network fabric can be evaluated, the work was inherently unable to cover all grounds. The following points are considered the most relevant to pursue next.

Workload variety and standardization

For proper evaluation of the performance of *data center* networks, a more clear understanding of the typical workloads being run therein is required. The work has taken some simplifications with respect to communication probabilities and flow size, by independently drawing from them when generating a flow arrival schedule. A better understanding of what is going on in a data center is required, beyond the scarcely published data sets heavily biased towards the data center configuration in which they were run, such as that of [10]. Publication of flow size distributions and communication probabilities at fine granular level would aid this effort significantly. A tool, or guidelines beyond intuition in general, are necessary to translate these data sets and traces into topology-independent traffic scenarios. This translation would lead to the ability to truly compare data center network fabrics in an objective and thorough way.

Steps towards physical deployment

The thesis is written solely based on simulations, either at abstract flow-level, or at packet-level. It would be interesting to see the actual deployment of a topology in the class of expanders in a real data center. Beyond the modest proposition of the deployment in section 4.4, there are still many engineering questions that need to be addressed. To name a few: translation of the graph to a physical space, wiring strategies and cost estimation thereof, cost estimation in general, writing of the Linux kernel module to perform the routing, and an analysis of financial feasibility respective to e.g. main-

tainability. Furthermore, more complicated routing schemes could impose additional requirements of the equipment needed for a proper deployment. These engineering aspects can also have significant impact on attainable performance.

Expansion and verification of the network simulation model

In the discrete packet simulator, crucial simplification assumptions have been made for the sake of understandability, scalability and the ability to interpret results. Besides more carefully characterizing the equipment being emulated, other network-level phenomena could also be emulated such as the occurrence of crashes, black holes, link failure, or duplicate segment creation. Moreover, it would be interesting to scale the simulator further towards topologies of the size of contemporary data center networks. This might go at the cost of additional simplifying assumptions to increase speed. Besides expansion of the model, it is necessary to verify the simulator's results more thoroughly. In future work the results should be compared to simulations run on other existing packet simulators or to a (small-scale) deployment.

Routing schemes and their parametrization

A rather basic version of routing was used, a simplified version of TCP with DCTCP. A more thorough exploration of protocol and routing logic is required, as well as the parametrization of the proposed logic (e.g. the flowlet gap and RTO). Specifically, for routing logic, the exploration of K-paths and some (oblivious) selection mechanism to select the K paths is presumably interesting. Additionally, the simulations showed a need to adjust the threshold Q parameter of the HYBRID approach such that it performs well in most, if not all, traffic scenarios.

Congestion-control

A significantly interesting avenue is in general the investigation of interplay between congestion control at the transport layer level and performed by the routing switch. For example: a worst-case scenario in which the route of a specific flow is switched to an non-congested path with DCTCP receiving ECN acknowledgment packets from the previous flowlet, unnecessarily decreasing the congestion window on the new route. A more thorough understanding of this interplay could aid the design of future routing and congestion schemes.

Theoretical/practical understanding of dynamic topologies' performance

The flow evaluation was only able to express very basic models of the dynamic topologies. The buffer overhead analysis for dynamic topologies in an all-to-all in appendix A offers some indication of the trade-off that dynamic topologies might expect. This is however far from being able to express tight

bounds on its best possible performance, as is done for e.g. the random regular graph for a subset of traffic matrices [25]. A theoretical framework covering buffering and making use of optimal routing could be envisioned to address this gap of knowledge.

Chapter 7

Conclusion

The work has investigated the claim that static data center network fabrics are fundamentally limited, especially being unable to handle skewed workloads with hot-spots. It provides evidence that the criticism should be directed at the traditional tree-like topologies currently widely deployed such as the fat-tree [1], instead of static topologies as a whole. Either choice of the harsh dichotomy forced by traditional tree-like topologies, to either deploy (a) a costly full-bandwidth network, or (b) a cheaper oversubscribed network, was evaluated. It was shown that oversubscribed fat-trees are indeed limited for example in a case of only a fraction of servers active [24].

The performance claims made by the recent state-of-the-art dynamic topology Projector [10] have been rivaled by the static *expander* data center network fabric [26, 27]. The expander was able to perform 7-96% better than the fat-tree versus the 30-95% performance improvement claimed by ProjecToR. In the comparison, the static expander topology was not compensated for its cheaper static ports, further strengthening the claim that static expander topologies can rival dynamic topologies. The ProjecToR comparison did not provide results when server bottlenecks are bottlenecked as well. With the addition of server bottlenecks, the performance improvement of the expander was 5-6%.

The work has shown that the static expander topology is in fact able to handle these supposedly adverse traffic scenarios for static topologies, rivaling performance of the full-bandwidth fat-tree at 67.5-80% of its cost using at least one of its routing strategies. Three factors had significant impact in this achievement: (a) the high ToR-ratio (per-ToR network up-links divided by per-ToR server up-links), (b) the low average shortest path length [25], and (c) the property that every cut in the network is traversed by many links [27]. Both primary routing strategies, shortest path (ECMP) and valiant load balancing (VLB), have shown to be best in specific traffic scenarios. Shortest path routing was specifically effective at either a high fraction of nodes

active or when the active servers are spread over the nodes. Longer path routing, namely valiant load balancing, proved to be best in traffic scenarios with a low fraction of nodes active. A hybrid routing strategy (HYBRID), that switches from shortest path routing to valiant load balancing after a certain threshold, has the ability to combine the best of the two strategies at correct parametrization.

In the work, a path is shown beyond traditional fat-trees. The expander is able to overcome the flaws signified by recent literature: the expander topology (a) is incrementally expandable [27], (b) can handle skewed traffic scenarios adverse to fat-trees, and (c) is able to achieve performance rivaling that of the full-bandwidth fat-tree at a lower cost. Moreover, it does not suffer from many concerns associated with state-of-the-art dynamic equipment such as environmental factors (e.g. dust, vibration) and operator experience. All these achievements project a bright future of the static expander topology as the viable candidate for data center network fabrics.

I would like to give a special thanks to my supervisor Ankit Singla, and collaborators on the project, Asaf Valdarsky and Michael Shapira. Their support throughout the project has proven invaluable, aiding with provisioning of resources, engaging discussions and sharing of knowledge and intuitions.

Appendix A

Dynamic Network All-to-All Buffer and Throughput Analysis

Recent dynamic topology proposals [10, 12, 13] argue for the use of greedy matching algorithms to determine which dynamic links to create. These algorithms match switches with high demand directly to each other, in favor of short routing in order to prevent wasting network capacity. In this chapter a characterization of the performance and buffer trade-off of direct coupling of in-demand nodes is made. Particularly, the concept of rounds is used throughout the analysis, mimicking what is employed by ProjecToR [10].

An especially difficult traffic scenario for a dynamic network using only direct coupling is the All-to-All traffic scenario, as it intuitively forces every node to contact all other nodes. In this appendix the buffer cost and throughput performance that traffic adaptation network topologies can achieve under that traffic scenario is discussed. In the upcoming three sections, three topology adaptation strategy based on round-robin are described. First, in section A.1, the *full* strategy is discussed which couples all links of two nodes together each round. Second, in section A.2, the *one-by-one* strategy is introduced which one moves one link each round, maximizing throughput. Third, in section A.3, the *fan-out* strategy is examined which moves all links in a spread fashion, minimizing buffer occupancy.

A.1 Full Coupling Strategy

In this topology adaptation strategy, each time-slot every node connects to the next node with all its network ports. This means that all other traffic for servers located on the currently not-connected nodes is buffered. An example of the strategy in action is shown in figure A.1.

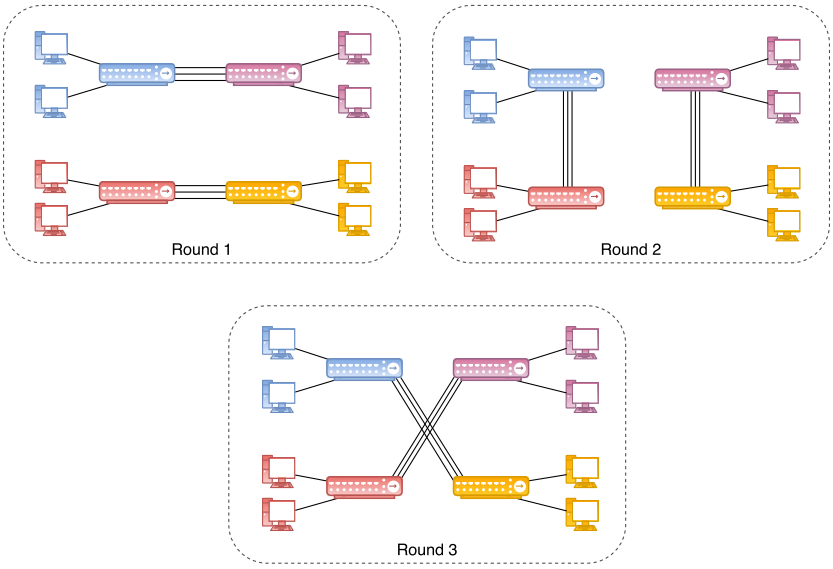


Figure A.1: All $n-1$ rounds for a round-robin full coupling with $k = 2$, $r = 3$ and $n = 4$.

A.1.1 Variables and constants

t_{change}	Time to change a link in each round;
$t_{transmit}$	Transmission time, the amount of time data can be transmitted per round;
t_{round}	Total time of a round: $t_{change} + t_{transmit}$;
n	Number of switches;
c_{link}	Link capacity;
k	Servers per switch;
r	Network ports per switch.

A.1.2 Derivation

The amount of traffic a server can at maximum generate when connected is as follows:

$$\alpha_{base} = \frac{r \cdot c_{link}}{k}$$

The ratio of time at which a server can generate traffic is as follows:

$$ratio_{active} = \frac{t_{transmit}}{t_{transmit} + t_{change}}$$

The amount of total traffic generated by a server to all other servers is as follows:

$$\alpha_{real} = ratio_{active} \cdot \alpha_{base}$$

Because there is no within-switch communication, there are in total $(n-1) \cdot k$ destination servers for each server. Every server thus maintains $(n-1) \cdot k$ output queues. The amount of traffic generated for each of these queues with a single other server as destination is thus as follows:

$$\alpha_{single\ queue.} = \frac{\alpha_{real}}{(n-1) \cdot k}$$

After a switch has júst started to change connections from switch x to y, it will take $n-2$ full rounds and a single change time before it returns to switch x. After this the server-server pair buffer has reached it largest size. The buffer of a single queue (for a single destination server) is thus as follows:

$$\beta_{revisited} = ((n-2) \cdot t_{round} + t_{change}) \cdot \alpha_{single\ queue.}$$

From then on, the buffer $\beta_{revisited}$ is worked away with $\frac{\alpha_{base}}{k}$ as it uniformly sends its base throughput to all k destination servers from the k single queues "whose turn it now is". This mean that it is able to consume the following amount:

$$\beta_{consumed} = \frac{\alpha_{base}}{k} \cdot t_{transmit}$$

It worth noticing that $\beta_{revisited} < \beta_{consumed}$, this is because during the transmission period, some small amount of traffic is also generated:

$$\beta_{during\ visit} = t_{transmit} \cdot \alpha_{single\ queue.}$$

As such:

$$\beta_{build-up} = \beta_{revisited} + \beta_{during\ visit}$$

Equilibrium preservation introduced by $ratio_{active}$ is required, thus it must be proven that $\beta_{build-up} = \beta_{consumed}$:

$$\beta_{build-up} =$$

$$\beta_{revisited} + \beta_{during\ visit} =$$

$$((n-2) \cdot t_{round} + t_{change}) \cdot \alpha_{single\ queue.} + t_{transmit} \cdot \alpha_{single\ queue.} =$$

$$((n-1) \cdot t_{round}) \cdot \alpha_{single\ queue.} =$$

$$((n-1) \cdot t_{round}) \cdot \frac{\alpha_{real}}{(n-1) \cdot k} =$$

$$\frac{t_{round} \cdot \alpha_{real}}{k} =$$

$$\frac{t_{round} \cdot ratio_{active} \cdot \alpha_{base}}{k} =$$

$$\frac{t_{round} \cdot \frac{t_{transmit}}{t_{transmit}+t_{change}} \cdot \alpha_{base}}{k} =$$

$$\frac{(t_{transmit} + t_{change}) \cdot \frac{t_{transmit}}{t_{transmit}+t_{change}} \cdot \alpha_{base}}{k} =$$

$$\frac{\alpha_{base}}{k} \cdot t_{transmit} =$$

$$\beta_{consumed}$$

Now, the only open question remains: when is the sum of all buffers of all single queues on each server the highest? Naturally this is the moment júst before another transmission period starts. k queues will be only filled with $t_{change} \cdot \alpha_{single\ queue.}$ as they have júst been emptied in the previous round, then the k queues in the round before that filled with $(t_{round} + t_{change}) \cdot \alpha_{single\ queue}$. This continues until all $k \cdot (n-1)$ queues are accounted for. As such, it is defined as follows:

$$\beta_{server\ total\ max.} = \sum_{i=1}^{n-1} k \cdot ((i-1) \cdot t_{round} + t_{change}) \cdot \alpha_{single\ queue.}$$

The sum of all buffers of all single queues on each server is the lowest after a transmission period has júst ended. As such, this is defined as follows:

$$\beta_{server\ total\ min.} = \sum_{i=1}^{n-1} k \cdot ((i-1) \cdot t_{round}) \cdot \alpha_{single\ queue.}$$

A final observation is made regarding the amount of buffer versus the throughput of the network as a whole. The total amount of data transferred in a complete round-trip is $n \cdot k \cdot \alpha_{real} \cdot (t_{round} \cdot (n-1))$. The minimum total buffer at network level is $n \cdot k \cdot \beta_{server\ total\ min.}$ and the maximum total buffer at network level is $n \cdot k \cdot \beta_{server\ total\ max.}$. The maximum total buffer will never be more than or equal to half of the total amount of data transferred in each round-trip. This is due to the fact that the network situation in a round is bi-directional and never to itself.

A.1.3 Example

Take for example the ProjecToR dynamic network [10] with the following variables:

$$t_{change} = 12\mu s\ ,\ t_{transmit} = 120\mu s\ ,\ t_{round} = 132\mu s\ ,\ n = 1001\ ,$$
$$c_{link} = 10\,{}^{Gb}/_{s}\ ,\ k = 25\ ,\ r = 25$$

The calculations based on the variables are as follows:

$$\alpha_{base} = \frac{r \cdot c_{link}}{k} = \frac{25 \cdot 10}{25} = 10\,{}^{Gb}/_{s}$$

$$ratio_{active} = \frac{t_{transmit}}{t_{transmit} + t_{change}} = \frac{120}{120 + 12} = 0.9090909$$

$$\alpha_{real} = ratio_{active} \cdot \alpha_{base} = 0.9090909 \cdot 10 = 9.090909\,{}^{Gb}/_{s}$$

$$\alpha_{single\ queue.} = \frac{\alpha_{real}}{(n-1) \cdot k} = \frac{9.090909}{(1001-1) \cdot 25} = 0.00036363636\,{}^{Gb}/_{s}$$

$$\beta_{revisited} = ((n-2) \cdot t_{round} + t_{change}) \cdot \alpha_{single\ queue.} =$$
$$((1001-2) \cdot 132\mu s + 12\mu s) \cdot 0.00036363636\,{}^{Gb}/_{s} = 5.99455\ KB$$

$$\beta_{during\ visit} = t_{transmit} \cdot \alpha_{single\ queue.} = 120\mu s \cdot 0.00036363636\,{}^{Gb}/_{s} = 0.00545\ KB$$

$$\beta_{build-up} = \beta_{revisited} + \beta_{during\ visit} =$$
$$5.99455\ KB + 0.00545\ KB = 6\ KB$$

$$\beta_{consumed} = \frac{\alpha_{base}}{k} \cdot t_{transmit} = \frac{10\,Gb/s}{25} \cdot 120\mu s = 6\ KB$$

$$\beta_{server\ total\ max.} = \sum_{i=1}^{n-1} k \cdot ((i-1) \cdot t_{round} + t_{change}) \cdot \alpha_{single\ queue.} =$$

$$\sum_{i=1}^{1001-1} 25 \cdot ((i-1) \cdot 132\mu s + 12\mu s) \cdot 0.00036363636\,Gb/s =$$

$$\sum_{i=1}^{1001-1} (i-1) \cdot 150\ bytes + (1001-1) \cdot 13.636363\ bytes =$$

$$499500 \cdot 150\ bytes + 1000 \cdot 13.636363\ bytes \approx 74.925\ MB + 13.6\ KB \approx 74.94\ MB$$

$$\beta_{server\ total\ min.} = \sum_{i=1}^{n-1} k \cdot ((i-1) \cdot t_{round}) \cdot \alpha_{single\ queue.} =$$

$$\sum_{i=1}^{1001-1} 25 \cdot ((i-1) \cdot 132\mu s) \cdot 0.000363636\,Gb/s =$$

$$\sum_{i=1}^{1001-1} (i-1) \cdot 150\ bytes = 74.925\ MB$$

A.2 One-by-One Coupling Strategy

The Full Coupling strategy only tackles k queues of every server at any point in time, resulting in unnecessary queueing is happening in the initial state. This queueing can be reduced by making use of the fan-out of the r network ports to r different switches. As it is *All-to-All*, round-robin is still the optimal way to schedule this topology adaptation strategy. Because only a single link changes every round, all other links can continue to send. This will result in higher throughput. An example of the strategy in action is shown in figure A.2.

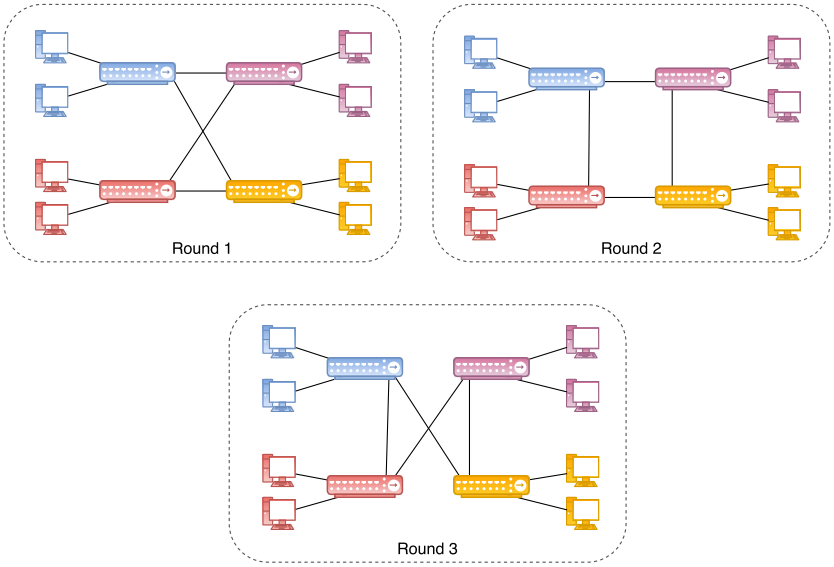


Figure A.2: All $n-1$ rounds for a round-robin one-by-one coupling with $k=2$, $r=2$ and $n=4$.

A.2.1 Variables and constants

The variables and constants are the same as section A.1.1.

A.2.2 Derivation

The amount of traffic a server that can at maximum be generated when connected:

$$\alpha_{base} = \frac{r \cdot c_{link}}{k}$$

Because only one link is changed every round, all $r-1$ other links can stay and continue to transmit in the changing time of that lone link. The ratio of

time at which a server can generate traffic is thus as follows:

$$ratio_{active} = \frac{(r-1)\cdot(t_{transmit}+t_{change})+t_{transmit}}{r\cdot(t_{transmit}+t_{change})}$$

The amount of total traffic generated by a server to all other servers is as follows:

$$\alpha_{real} = ratio_{active}\cdot\alpha_{base}$$

There are in total $(n-1)\cdot k$ destination servers. Each server thus maintains $(n-1)\cdot k$ output queues. The amount of traffic generated for each of these queues with a single other server as destination:

$$\alpha_{single\ queue.} = \frac{\alpha_{real}}{(n-1)\cdot k}$$

After a switch has júst started to change connections from switch x to y, it will take $n-2-(r-1)$ full rounds and a single change time before it returns to connect to the servers on switch x. During the visit of r rounds, it also builds up while it is transmitting its buffer. After the visit, the buffer should be exactly empty. The buffer of a single queue (for a single destination server) at this moment is thus as follows:

$$\beta_{build-up} = ((n-1)\cdot t_{round})\cdot\alpha_{single\ queue.}$$

The buffer is consumed over r rounds, over which it can transmit during $r-1$ full rounds and a single transmission time. The final change time is used to switch. In this time, he can thus devote to this single queue $(k\cdot r)$'th of the base throughput per server:

$$\beta_{consumed} = \frac{\alpha_{base}}{k\cdot r}\cdot(t_{round}\cdot(r-1)+t_{transmit})$$

Equilibrium preservation introduced by $ratio_{active}$ is required, thus it must be proven that $\beta_{build-up} = \beta_{consumed}$:

$$\beta_{build-up} =$$
$$((n-1)\cdot t_{round})\cdot\alpha_{single\ queue.} =$$
$$((n-1)\cdot t_{round})\cdot\frac{\alpha_{real}}{(n-1)\cdot k} =$$

$$\frac{(n-1)}{(n-1)} \cdot t_{round} \cdot \frac{\alpha_{real}}{k} =$$

$$\frac{\alpha_{real}}{k} \cdot t_{round} =$$

$$\frac{t_{round}}{k} \cdot \alpha_{real} =$$

$$\frac{t_{round}}{k} \cdot ratio_{active} \cdot \alpha_{base} =$$

$$\frac{t_{round}}{k} \cdot \frac{(r-1) \cdot (t_{transmit} + t_{change}) + t_{transmit}}{r \cdot (t_{transmit} + t_{change})} \cdot \alpha_{base} =$$

$$\frac{\alpha_{base}}{k \cdot r} \cdot \frac{(r-1) \cdot (t_{transmit} + t_{change}) + t_{transmit}}{(t_{transmit} + t_{change})} \cdot t_{round} =$$

$$\frac{\alpha_{base}}{k \cdot r} \cdot \frac{(r-1) \cdot t_{round} + t_{transmit}}{t_{round}} \cdot t_{round} =$$

$$\frac{\alpha_{base}}{k \cdot r} \cdot (t_{round} \cdot (r-1) + t_{transmit})$$
$$= \beta_{consumed}$$

What remains is to determine the total buffers sizes per server. Naturally, the maximum total buffer size is the moment júst before another transmission period starts. k queues will be only filled with $t_{change} \cdot \alpha_{single\ queue.}$ as they have júst been emptied in the previous round, then the k queues in the round before that filled with $(t_{round} + t_{change}) \cdot \alpha_{single\ queue}$. This continues until all $k \cdot (n-1)$ queues are accounted for. Additionally, $k \cdot (r-1)$ other queues have also been reduced; the next k queues that will finish next round will have already sent $(r-1) \cdot \frac{\beta_{consumed}}{r}$ away, the ones that finish the round after that $(r-2) \cdot \frac{\beta_{consumed}}{r}$, and so forth. This results in the following formulation:

$$\beta_{server\ total\ max.} =$$
$$\sum_{i=1}^{n-1} k \cdot ((i-1) \cdot t_{round} + t_{change}) \cdot \alpha_{single\ queue.} - \sum_{i=1}^{r-1} k \cdot i \cdot \frac{\beta_{consumed}}{r}$$

Likewise, the same applies to the minimum total buffer size, but then júst after a transmission has finished:

$$\beta_{server\ total\ min.} =$$
$$\sum_{i=1}^{n-1} k \cdot ((i-1) \cdot t_{round}) \cdot \alpha_{single\ queue.} - \sum_{i=1}^{r-1} k \cdot i \cdot \frac{\beta_{consumed}}{r}$$

A.2.3 Example

Take the example of the ProjecToR dynamic network [10] with the following variables:

$$t_{change} = 12\mu s\ ,\ t_{transmit} = 120\mu s\ ,\ t_{round} = 132\mu s\ ,\ n = 1001\ ,$$
$$c_{link} = 10\,Gb/s\ ,\ k = 25\ ,\ r = 25$$

The calculations based on the variables are as follows:

$$\alpha_{base} = \frac{r \cdot c_{link}}{k} = \frac{25 \cdot 10}{25} = 10\,Gb/s$$

$$ratio_{active} = \frac{(r-1)\cdot(t_{transmit} + t_{change}) + t_{transmit}}{r \cdot (t_{transmit} + t_{change})} =$$
$$\frac{(25-1)\cdot(120+12)+120}{r\cdot(120+12)} = 0.99636363$$

$$\alpha_{real} = ratio_{active} \cdot \alpha_{base} = 0.99636363 \cdot 10 = 9.9636363\,Gb/s$$

$$\alpha_{single\ queue.} = \frac{\alpha_{real}}{(n-1)\cdot k} = \frac{9.9636363}{(1001-1)\cdot 25} = 0.00039854545\,Gb/s$$

$$\beta_{build-up} = ((n-1)\cdot t_{round})\cdot \alpha_{single\ queue.} =$$
$$((1001-1)\cdot(132\mu s))\cdot 0.00039854545\,Gb/s = 6.576\ KB$$

$$\beta_{consumed} = \frac{\alpha_{base}}{k\cdot r}\cdot(t_{round}\cdot(r-1)+t_{transmit}) =$$
$$\frac{10\,Gb/s}{25\cdot 25}\cdot(132\mu s\cdot(25-1)+120\mu s) = 6.576\ KB$$

$$\beta_{server\ total\ max.} =$$
$$\sum_{i=1}^{n-1} k\cdot((i-1)\cdot t_{round}+t_{change})\cdot\alpha_{single\ queue.} - \sum_{i=1}^{r-1} k\cdot i\cdot\frac{\beta_{consumed}}{r} =$$

$$\sum_{i=1}^{1001-1} 25\cdot((i-1)\cdot 132\mu s+12\mu s)\cdot 0.00039854545\,Gb/s - \sum_{i=1}^{25-1} 25\cdot i\cdot\frac{6.576\ KB}{25} =$$

$$\sum_{i=1}^{1001-1} (i-1) \cdot 164.4\ bytes + 1000 \cdot 14.9454545\ bytes - \sum_{i=1}^{25-1} i \cdot 6.576\ KB =$$

$$499500 \cdot 164.4\ bytes + 1000 \cdot 14.9454545\ bytes - 300 \cdot 6.576\ KB \approx 80.16\ MB$$

$$\beta_{server\ total\ min.} =$$
$$\sum_{i=1}^{n-1} k \cdot ((i-1) \cdot t_{round}) \cdot \alpha_{single\ queue.} - \sum_{i=1}^{r-1} k \cdot i \cdot \frac{\beta_{consumed}}{r} =$$
$$499500 \cdot 164.4\ bytes - 300 \cdot 6.576\ KB = 80.145\ MB$$

A.3 Fan-Out Coupling Strategy

The One-by-One strategy does tackle k^2 server traffic pairs at any point in time, but moving one link at a time incurs high buffering. This queueing can be further reduced by making use of the fan-out of the r network ports to r different switches, but also moving all network ports at the same time. The throughput is the same as the full coupling strategy, as no changing times can be used to transmit. An example of the strategy in action is shown in figure A.3.

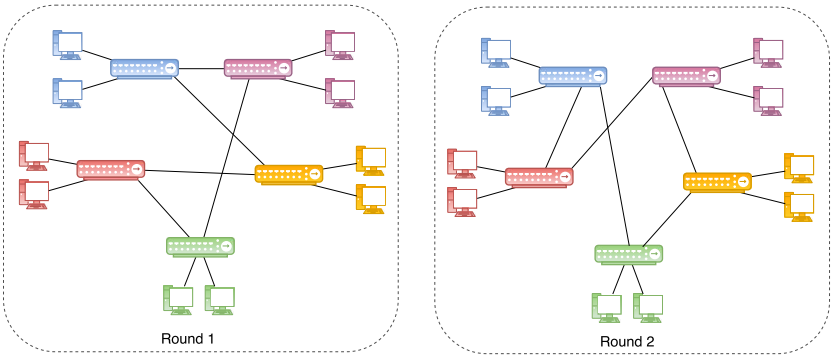


Figure A.3: All $(n-1)/r$ rounds for a round-robin fan-out coupling with $k = 2$, $r = 2$ and $n = 5$.

A.3.1 Variables and constants

The variables and constants are the same as section A.1.1. The derivation until $\alpha_{single\ queue.}$ is also the same, as the production per queue does not depend on the fan-out of the r links. We assume, for the simplicity of the formulas, that $(n-1)\ modulo\ r = 0$.

A.3.2 Derivation

After a switch has júst started to change all its connections from switches $[x, ..., x+r-1]$ to $[x+r, x+2r-1]$, it will take $(n-1)/r$ (a round-trip) is against finished with serving all the servers attached to switches $[x, ..., x+r-1]$. The buffer of a single queue (for a single destination server) at this moment is thus as follows:

$$\beta_{build-up} = (((n-1)/r) \cdot t_{round}) \cdot \alpha_{single\ queue.}$$

In this full round-trip, the buffer is worked away with $\frac{\alpha_{base}}{k \cdot r}$ as it uniformly sends its base throughput to all $k \cdot r$ destination servers from the $k \cdot r$ single

queues "whose turn it now is". This mean that it is able to consume the following amount in the round-trip:

$$\beta_{consumed} = \frac{\alpha_{base}}{kr} \cdot t_{transmit}$$

Equilibrium preservation introduced by $ratio_{active}$ is required, thus it must be proven that $\beta_{build-up} = \beta_{consumed}$:

$$\beta_{build-up} =$$
$$(((n-1)/r) \cdot t_{round}) \cdot \alpha_{single\ queue.} =$$
$$(((n-1)/r) \cdot t_{round}) \cdot \frac{\alpha_{real}}{(n-1) \cdot k} =$$
$$\frac{1}{r} \cdot \frac{t_{round} \cdot \alpha_{real}}{k} =$$

(See derivation in section A.1.2)

$$\frac{1}{r} \cdot \frac{\alpha_{base}}{k} \cdot t_{transmit} =$$
$$\frac{\alpha_{base}}{kr} \cdot t_{transmit} =$$
$$\beta_{consumed}$$

Now, the only open question remains: when is the sum of all buffers of all single queues on each server the highest? Naturally this is the moment júst before another transmission period starts. k queues will be only filled with $t_{change} \cdot \alpha_{single\ queue.}$ as they have júst been emptied in the previous round, then the k queues in the round before that filled with $(t_{round} + t_{change}) \cdot \alpha_{single\ queue}$. This continues until all $k \cdot (n-1)$ queues are accounted for. As such, it is defined as follows:

$$\beta_{server\ total\ max.} = \sum_{i=1}^{(n-1)/r} k \cdot r \cdot ((i-1) \cdot t_{round} + t_{change}) \cdot \alpha_{single\ queue.}$$

The sum of all buffers of all single queues on each server is the lowest after a transmission period has júst ended. As such, this is defined as follows:

$$\beta_{server\ total\ min.} = \sum_{i=1}^{(n-1)/r} k \cdot r \cdot ((i-1) \cdot t_{round}) \cdot \alpha_{single\ queue.}$$

A.3.3 Example

Take the example of the ProjecToR dynamic network [10] with the following variables (r is set to 25, which satisfies the $(n-1)\ mod\ r = 0$ condition):

$$t_{change} = 12\mu s\ ,\ t_{transmit} = 120\mu s\ ,\ t_{round} = 132\mu s\ ,\ n = 1001\ ,$$
$$c_{link} = 10\ ^{Gb}/_{s}\ ,\ k = 25\ ,\ r = 25$$

$$\alpha_{base} = \frac{r \cdot c_{link}}{k} = \frac{25 \cdot 10}{25} = 10\ ^{Gb}/_{s}$$

$$ratio_{active} = \frac{t_{transmit}}{t_{transmit} + t_{change}} = \frac{120}{120 + 12} = 0.9090909$$

$$\alpha_{real} = ratio_{active} \cdot \alpha_{base} = 0.9090909 \cdot 10 = 9.090909\ ^{Gb}/_{s}$$

$$\alpha_{single\ queue.} = \frac{\alpha_{real}}{(n-1) \cdot k} = \frac{9.090909}{(1001-1) \cdot 25} = 0.00036363636\ ^{Gb}/_{s}$$

$$\beta_{build-up} = (((n-1)/r) \cdot t_{round}) \cdot \alpha_{single\ queue.} =$$
$$((1001-1)/25 \cdot 132\mu s) \cdot 0.00036363636\ ^{Gb}/_{s} = 240\ bytes$$

$$\beta_{consumed} = \frac{\alpha_{base}}{k \cdot r} \cdot t_{transmit} = \frac{10\ ^{Gb}/_{s}}{25 \cdot 25} \cdot 120\mu s = 240\ bytes$$

$$\beta_{server\ total\ max.} = \sum_{i=1}^{(n-1)/r} k \cdot r \cdot ((i-1) \cdot t_{round} + t_{change}) \cdot \alpha_{single\ queue.} =$$
$$\sum_{i=1}^{(1001-1)/25} 25 \cdot 25 \cdot ((i-1) \cdot 132\mu s + 12\mu s) \cdot 0.00036363636\ ^{Gb}/_{s} =$$
$$\sum_{i=1}^{40}(i-1) \cdot 3750\ bytes + 40 \cdot 340.9\ bytes = 780 \cdot 3750\ bytes + 40 \cdot 340.9\ bytes \approx 2.939\ MB$$

$$\beta_{server\ total\ min.} = \sum_{i=1}^{(n-1)/r} k \cdot r \cdot ((i-1) \cdot t_{round}) \cdot \alpha_{single\ queue.} =$$
$$\sum_{i=1}^{(1001-1)/25} 25 \cdot 25 \cdot ((i-1) \cdot 132\mu s) \cdot 0.00036363636\ ^{Gb}/_{s} =$$
$$\sum_{i=1}^{40}(i-1) \cdot 3750\ bytes = 2.925\ MB$$

Appendix B

Oversubscribed Fat-tree Calculation

Constants:
Total number of nodes: N_{total}
Switch degree: d
Total number of servers to support: $n_{servers}$

Calculation fat-tree:
Number of bottom nodes: N_{bottom}
Number of aggregation nodes: N_{agg}
Number of core nodes: $N_{core} = N_{agg}/2$
Fraction of links used at bottom for network up-links: x

Total number of nodes constraint:
$N_{total} = N_{bottom} + N_{agg} + N_{core} = N_{bottom} + 1.5 \cdot N_{agg}$

Aggregation layer up and down links are the same constraint:
$N_{agg} \cdot 0.5 \cdot d = N_{bottom} \cdot x \cdot d$

Bottom layer must support the desired amount of servers:
$(1 - x) \cdot N_{bottom} \cdot d = n_{servers} \iff (1 - x) \cdot N_{bottom} = n_{servers}/d$

System of equations[1]
$[\ N_{total}\ ] = a + 1.5 * b$
$0.5 * b = a * x$
$(1 - x) * a = [\ n_{servers}\ ]/[\ d\ ]$
$t = x/(1 - x)$

In which:

[1]The system of equations can for example be solved using Wolfram-Alpha; https://www.wolframalpha.com/ (retrieved 12-03-2017)

a: N_{bottom}
b: N_{agg}
x: fraction of links used at bottom layer for network up-links
t: throughput fraction of the fat-tree

Example figure 4a of [24]
$N_{total} = 578$
$d = 49$
$n_{servers} = 13872$

gives

$N_{bottom} = 356.827$
$N_{agg} = 147.449$
$x = 0.2066$
$t = 0.2604$

Example figure 4b of [24]
$N_{total} = 512$
$d = 18$
$n_{servers} = 4096$

gives

$N_{bottom} = 298.667$
$N_{agg} = 142.222$
$x = 0.2381$
$t = 0.3125$

Appendix C

All-to-All Regular Network Bound

The maximum node throughput achievable in an all-to-all for any r-regular static network is $\frac{r}{\langle D \rangle}$ [25], as such $\frac{r}{\langle D \rangle \cdot s}$ as server throughput for s servers per node. The lower bound d^* on the average shortest path length $\langle D \rangle$ in regular graphs is as follows [7]:

$$\langle D \rangle \geq d^* = \frac{\sum_{j=1}^{k-1} jr(r-1)^{j-1} + k \cdot R}{N-1}$$

where:

$$R = N - 1 - \sum_{j=1}^{k-1} r(r-1)^{j-1} \geq 0$$

In which the value of k is the largest integer such that the inequality holds. It can be trivially optimized by trying out increasing values of k, starting at 0, until the inequality is violated (when R is lower than 0). The found k and R can then be plugged into the first formula to determine the lower bound d^*.

Appendix D

NetBench Algorithms

D.1 ECN Tail Drop Output Port

Algorithm 4 Packet handling of ECN tail drop output port

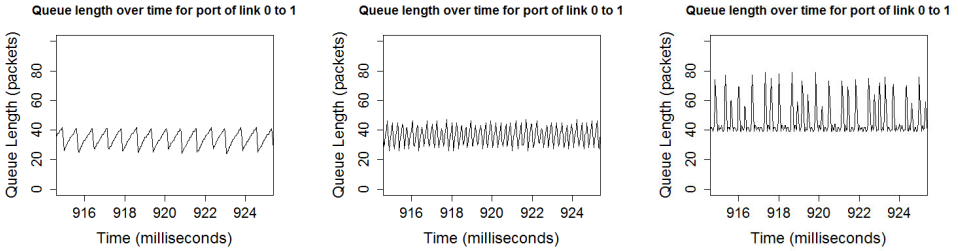
Require: *queue*, *is_sending*, *buffer_occupied_bits*, *link*, *target_network_device*,
Require: *ecn_threshod_bits*, *max_queue_size_bits*

```
 1: procedure ENQUEUE(packet)
 2:     if buffer_occupied_bits ≥ ecn_threshold_bits then
 3:         packet.mark_ecn()
 4:     end if
 5:     if buffer_occupied_bits + packet.size_bits ≤ max_queue_size_bits then
 6:         guaranteed_enqueue(packet)
 7:     end if
 8: end procedure
 9:
10: procedure GUARANTEED_ENQUEUE(packet)
11:     if is_sending then
12:         buffer_occupied_bits += packet.size_bits
13:         queue.add(packet)
14:     else
15:         register(new packet_dispatch_event(
16:             packet.size_bits / link.bandwidth, packet))
17:         is_sending ← true
18:     end if
19: end procedure
20:
21: procedure DISPATCH(packet)
22:     if link.does_not_drop(packet) then
23:         register(new packet_arrival_event(
24:             link.delay, packet, target_network_device))
25:     end if
26:     is_sending ← false
27:     if !queue.is_empty() then
28:         new_packet ← queue.poll()
29:         buffer_occupied_bits −= new_packet.size_bits
30:         register(new packet_dispatch_event(
31:             new_packet.size_bits / link.bandwidth, new_packet))
32:         is_sending ← true
33:     end if
34: end procedure
```

Appendix E

NetBench Benchmarks

E.1 DCTCP



(a) DCTCP $N = 2$ (b) DCTCP $N = 10$ (c) DCTCP $N = 40$

Figure E.1: Port queue length of single bottleneck with $K = 40$, $g = 1/16$ with number of flows $N = \{2, 10, 40\}$. Reproduce using *NetBench/micro/single_bottleneck.properties*.

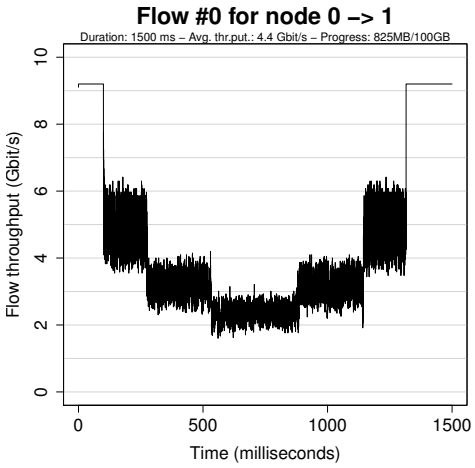


Figure E.2: Flow throughput for the continuous flow for $K = 40$, $g = 1/16$ with six phases with respectively 1, 2, 3, 4, 3, 2 and 1 flow active. Reproduce using *NetBench/micro/flow_convergence.properties*.

Bibliography

[1] Mohammad Al-Fares, Alexander Loukissas, and Amin Vahdat. A scalable, commodity data center network architecture. In *ACM SIGCOMM Computer Communication Review*, volume 38, pages 63–74. ACM, 2008.

[2] Mohammad Alizadeh, Tom Edsall, Sarang Dharmapurikar, Ramanan Vaidyanathan, Kevin Chu, Andy Fingerhut, Francis Matus, Rong Pan, Navindra Yadav, George Varghese, et al. Conga: Distributed congestion-aware load balancing for datacenters. In *ACM SIGCOMM Computer Communication Review*, volume 44, pages 503–514. ACM, 2014.

[3] Mohammad Alizadeh, Albert Greenberg, David A Maltz, Jitendra Padhye, Parveen Patel, Balaji Prabhakar, Sudipta Sengupta, and Murari Sridharan. Data center tcp (dctcp). In *ACM SIGCOMM computer communication review*, volume 40, pages 63–74. ACM, 2010.

[4] Mohammad Alizadeh, Abdul Kabbani, Tom Edsall, Balaji Prabhakar, Amin Vahdat, and Masato Yasuda. Less is more: trading a little bandwidth for ultra-low latency in the data center. In *Proceedings of the 9th USENIX conference on Networked Systems Design and Implementation*, pages 19–19. USENIX Association, 2012.

[5] Mohammad Alizadeh, Shuang Yang, Milad Sharif, Sachin Katti, Nick McKeown, Balaji Prabhakar, and Scott Shenker. pfabric: Minimal near-optimal datacenter transport. In *ACM SIGCOMM Computer Communication Review*, volume 43, pages 435–446. ACM, 2013.

[6] Mark Allman, Vern Paxson, and Ethan Blanton. Tcp congestion control rfc 5681. Technical report, 2009.

[7] Vinton G Cerf, Donald D Cowan, RC Mullin, and RG Stanton. A lower bound on the average shortest path length in regular graphs. *Networks*, 4(4):335–342, 1974.

[8] D Farinacci, T Li, S Hanks, D Meyer, and P Traina. Rfc 2784-generic routing encapsulation (gre). *IETF, March 2000*, 2000.

[9] Pankaj Garg and Yu-Shun Wang. Nvgre: Network virtualization using generic routing encapsulation. 2015.

[10] Monia Ghobadi, Ratul Mahajan, Amar Phanishayee, Nikhil Devanur, Janardhan Kulkarni, Gireeja Ranade, Pierre-Alexandre Blanche, Houman Rastegarfar, Madeleine Glick, and Daniel Kilper. Projector: Agile reconfigurable data center interconnect. In *Proceedings of the 2016 conference on ACM SIGCOMM 2016 Conference*, pages 216–229. ACM, 2016.

[11] Phillipa Gill, Navendu Jain, and Nachiappan Nagappan. Understanding network failures in data centers: measurement, analysis, and implications. In *ACM SIGCOMM Computer Communication Review*, volume 41, pages 350–361. ACM, 2011.

[12] Daniel Halperin, Srikanth Kandula, Jitendra Padhye, Paramvir Bahl, and David Wetherall. Augmenting data center networks with multi-gigabit wireless links. In *ACM SIGCOMM Computer Communication Review*, volume 41, pages 38–49. ACM, 2011.

[13] Navid Hamedazimi, Zafar Qazi, Himanshu Gupta, Vyas Sekar, Samir R Das, Jon P Longtin, Himanshu Shah, and Ashish Tanwer. Firefly: a reconfigurable wireless data center fabric using free-space optics. *ACM SIGCOMM Computer Communication Review*, 44(4):319–330, 2015.

[14] Christian E Hopps. Analysis of an equal-cost multi-path algorithm. 2000.

[15] Sangeetha Abdu Jyothi, Ankit Singla, P Brighten Godfrey, and Alexandra Kolla. Measuring and understanding throughput of network topologies. 2016.

[16] Srikanth Kandula, Dina Katabi, Shantanu Sinha, and Arthur Berger. Dynamic load balancing without packet reordering. *ACM SIGCOMM Computer Communication Review*, 37(2):51–62, 2007.

[17] Naga Katta, Mukesh Hira, Aditi Ghag, Changhoon Kim, Isaac Keslassy, and Jennifer Rexford. Clove: How i learned to stop worrying about the core and love the edge.

[18] Charles E Leiserson. Fat-trees: universal networks for hardware-efficient supercomputing. *IEEE transactions on Computers*, 100(10):892–901, 1985.

[19] Vincent Liu, Daniel Halperin, Arvind Krishnamurthy, and Thomas E Anderson. F10: A fault-tolerant engineered network. In *NSDI*, pages 399–412, 2013.

[20] Nick McKeown, Tom Anderson, Hari Balakrishnan, Guru Parulkar, Larry Peterson, Jennifer Rexford, Scott Shenker, and Jonathan Turner. Openflow: enabling innovation in campus networks. *ACM SIGCOMM Computer Communication Review*, 38(2):69–74, 2008.

[21] Jonathan Perry, Amy Ousterhout, Hari Balakrishnan, Devavrat Shah, and Hans Fugal. Fastpass: A centralized zero-queue datacenter network. In *ACM SIGCOMM Computer Communication Review*, volume 44, pages 307–318. ACM, 2014.

[22] Jon Postel et al. Transmission control protocol rfc 793, 1981.

[23] Ramon Marques Ramos, Magnos Martinello, and Christian Esteve Rothenberg. Slickflow: Resilient source routing in data center networks unlocked by openflow. In *Local Computer Networks (LCN), 2013 IEEE 38th Conference on*, pages 606–613. IEEE, 2013.

[24] Ankit Singla. Fat-free topologies. 2016.

[25] Ankit Singla, P Brighten Godfrey, and Alexandra Kolla. High throughput data center topology design. In *11th USENIX Symposium on Networked Systems Design and Implementation (NSDI 14)*, pages 29–41, 2014.

[26] Ankit Singla, Chi-Yao Hong, Lucian Popa, and P Brighten Godfrey. Jellyfish: Networking data centers randomly. In *Presented as part of the 9th USENIX Symposium on Networked Systems Design and Implementation (NSDI 12)*, pages 225–238, 2012.

[27] Asaf Valadarsky, Gal Shahaf, Michael Dinitz, and Michael Schapira. Xpander: Towards optimal-performance datacenters. In *Proceedings of the 12th International on Conference on emerging Networking EXperiments and Technologies*, pages 205–219. ACM, 2016.

[28] Damon Wischik, Costin Raiciu, Adam Greenhalgh, and Mark Handley. Design, implementation and evaluation of congestion control for multipath tcp. In *NSDI*, volume 11, pages 8–8, 2011.

[29] Danyang Zhuo, Qiao Zhang, Vincent Liu, Arvind Krishnamurthy, and Thomas Anderson. Rackcc: Rack-level congestion control.

ETH

Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Declaration of originality

The signed declaration of originality is a component of every semester paper, Bachelor's thesis, Master's thesis and any other degree paper undertaken during the course of studies, including the respective electronic versions.

Lecturers may also require a declaration of originality for other written papers compiled for their courses.

I hereby confirm that I am the sole author of the written work here enclosed and that I have compiled it in my own words. Parts excepted are corrections of form and content by the supervisor.

Title of work (in block letters):

STATIC YET FLEXIBLE: EXPANDER DATA CENTER NETWORK FABRICS

Authored by (in block letters):

For papers written by groups the names of all authors are required.

Name(s):	**First name(s):**
KASSING	SIMON ARNOLD

With my signature I confirm that

- I have committed none of the forms of plagiarism described in the 'Citation etiquette' information sheet.
- I have documented all methods, data and processes truthfully.
- I have not manipulated any data.
- I have mentioned all persons who were significant facilitators of the work.

I am aware that the work may be screened electronically for plagiarism.

Place, date	**Signature(s)**
ZURICH, 14 MARCH 2017	[redacted]

For papers written by groups the names of all authors are required. Their signatures collectively guarantee the entire content of the written paper.