NUMFabric: Fast and Flexible Bandwidth Allocation in Datacenters

Kanthi Nagaraj*, Dinesh Bharadia†, Hongzi Mao†, Sandeep Chinchali*, Mohammad Alizadeh†, Sachin Katti*
*Stanford University, †MIT CSAIL

ABSTRACT

We present NUMFabric, a novel transport design that provides flexible and fast bandwidth allocation control. NUMFabric is flexible: it enables operators to specify how bandwidth is allocated amongst contending flows to optimize for different service-level objectives such as weighted fairness, minimizing flow completion times, multipath resource pooling, prioritized bandwidth functions, etc. NUMFabric is also very fast: it converges to the specified allocation 2.3× faster than prior schemes. Underlying NUMFabric is a novel distributed algorithm for solving network utility maximization problems that exploits weighted fair queueing packet scheduling in the network to converge quickly. We evaluate NUMFabric using realistic datacenter topologies and highly dynamic workloads and show that it is able to provide flexibility and fast convergence in such stressful environments.

CCS Concepts

•**Networks** → **Transport protocols; Data center networks;**

Keywords

Resource allocation; Convergence; Network utility maximization; Weighted max-min; Packet scheduling

1. INTRODUCTION

Bandwidth allocation in networks has historically been at the mercy of TCP. TCP's model of allocation assumes that bandwidth should be shared equally among contending flows. However, for an increasing number of networks such as datacenters and private WANs, such an allocation is not a good fit and is often adversarial to the operator's intent. Consequently, there has been a flurry of recent work on transport designs, especially for datacenters, that target different bandwidth allocation objectives. Many aim to minimize per-packet latency [2, 54] or flow completion time [23, 3], while others target multi-tenant bandwidth allocation [58, 7, 55, 26], while still others focus on sophisticated objectives like resource pooling [56], policy-based bandwidth allocation [35], or coflow scheduling [15, 17, 14]. In effect, each design supports one point in the bandwidth allocation policy design space, but operators ideally want a transport that can be tuned for different points in the design space depending on workload requirements.

In this paper, we present NUMFabric, a novel transport fabric that enables operators to flexibly specify bandwidth allocation policies, and then achieves these policies in the network using simple, distributed mechanisms at the switches and end-hosts. NUMFabric's design is based on the classic Network Utility Maximization (NUM) [33] framework which allows per-flow resource allocation preferences to be expressed using *utility functions*. Utility functions encode the benefit derived by a flow for different bandwidth allocations, and can be chosen by the operator to achieve different bandwidth and fairness objectives. In §2, we show how an operator can translate high level policies such as varying notions of fairness, minimizing flow completion times, resource pooling a la MPTCP [56] and bandwidth functions [35] into utility functions at end-hosts. NUMFabric then realizes the bandwidth allocation that maximizes the sum of the utility functions in a completely distributed fashion.

Network utility maximization of course is not new. There is a long line of work [61] on designing distributed algorithms based on gradient descent for NUM (§3). However, these algorithms are slow to converge to the optimal rate. For datacenter workloads, where a majority of the flows may last only a few RTTs due to the high link speeds, the convergence time of these algorithms is often much larger than the lifetime of the majority of flows, and hence no guarantees on resource allocation can be made. Moreover, gradient-descent algorithms are difficult to tune since they have a "step-size" parameter that needs to be tuned for each workload and resource allocation objective. In practice, given the scale of datacenters and the variety of objectives, getting the tuning right is a formidable task.

Our main technical contribution is a transport design for solving the NUM problem that converges significantly faster than prior work and is much more robust. The key insight

SIGCOMM '16, August 22 - 26, 2016, Florianopolis , Brazil

 ISBN 978-1-4503-4193-6/16/08…$15.00

DOI: http://dx.doi.org/10.1145/2934872.2934890

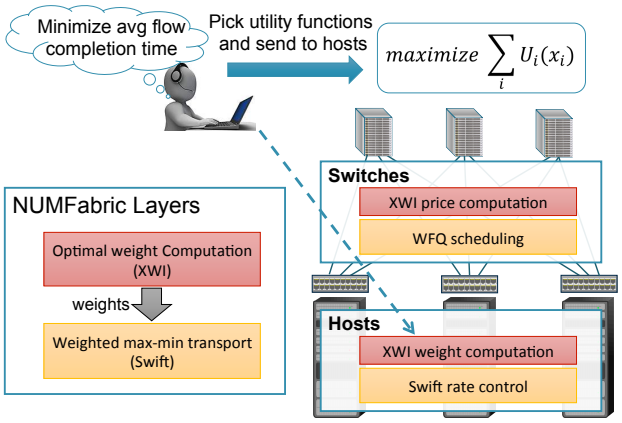


Figure 1: NUMFabric's high level architecture. The operator picks utility functions based on the desired allocation objective. NUMFabric realizes the allocation with distributed mechanisms at the hosts and switches that implement two logical layers: (1) Swift, a network-wide weighted max-min achieving transport; (2) xWI, an optimal weight computation algorithm.

underlying NUMFabric is to decouple the mechanisms for maximizing network utilization and achieving the optimal *relative* bandwidth allocation across competing flows. Existing NUM algorithms couple these objectives and try to accomplish both simultaneously through *price* variables at the links. This process is slow and brittle due to the need to balance between moving quickly towards the optimal allocation, and avoiding congestion or under-utilization (§4).

NUMFabric employs independent mechanisms for the two goals by decomposing the task of solving the NUM problem across two logical layers. Figure 1 shows the high level architecture. At the bottom layer, NUMFabric combines a packet scheduling mechanism based on weighted fair queuing (WFQ) [53, 16] at the switches and a simple rate control scheme at the hosts to achieve a *network wide weighted max-min* rate allocation for competing flows. NUMFabric's packet scheduling differs from standard WFQ in that the flow's weights are set dynamically at the hosts and carried in packet headers. This design, which we call the Swift transport (§4.1), exploits the fact that datacenter switches can be designed to support more sophisticated packet scheduling than simple FIFO queues [59]. Swift guarantees that the network is fully utilized while keeping flows isolated using WFQ. It achieves the weighted max-min allocation rapidly as the flows, or their weights, change.

Swift provides a convenient abstraction for controlling the relative bandwidth allocation of flows (via the flow weights). NUMFabric leverages this capability via a novel distributed algorithm called eXplicit Weight Inference (xWI) which runs on top of Swift (§4.2). In xWI, the sources and switches exchange information in packet headers to iteratively compute the weights that flows should use to achieve the optimal bandwidth allocation. Essentially, xWI iteratively guides the network from one weighted max-min allocation to the next until it finds the optimal NUM solution. Since there is no risk of congestion or under-utilization with weighted max-min allocations, xWI can aggressively update the link weights towards the optimal point and converge quickly.

We evaluate NUMFabric with detailed packet-level simulations in ns3 [49] using realistic datacenter topologies and workloads (§6). We compare NUMFabric for several resource allocation objectives with the best in class scheme for that objective, including pFabric [3] for FCT minimization, a variant of RCP [30] for α-fairness [47], and a well-known gradient-descent algorithm [40] for solving general NUM problems. We find that NUMFabric is fast and flexible. Specifically, compared to gradient descent based solutions [40, 30], NUMFabric converges to the optimal allocations $2.3\times$ faster at the median and $2.7\times$ faster at the 95^{th} percentile; it also allows operators to realize a wide variety of fairness-utilization objectives [47], policies that optimize flow completion time [3], and sophisticated policies such as resource pooling [56] and bandwidth functions [35].

2. HOW TO CHOOSE UTILITY FUNCTIONS?

NUMFabric adopts NUM [33] as a flexible framework for expressing fine-grained bandwidth allocation preferences as an optimization problem. In the most basic form, each network flow is associated with a *utility* as a function of its rate. The goal is to allocate rates to maximize the overall system utility, subject to link capacity constraints:

$$\begin{aligned} \text{maximize} \quad & \sum_i U_i(x_i) \\ \text{subject to} \quad & \mathbf{Rx} \leq \mathbf{c}. \end{aligned} \tag{1}$$

Here, $\mathbf{x}$ is the vector of flow rates; $\mathbf{R}$ is the $\{0,1\}$ routing matrix, i.e., $R(i,l) = 1$ if and only if flow i traverses link l; and $\mathbf{c}$ is the vector of link capacities. The utility functions, $U_i(\cdot)$ are assumed to be smooth, increasing, and strictly concave.[1] A flow is defined generically; for example, a flow can be a TCP connection, traffic between a pair of hosts, or traffic sent or received by a host.

The utility function choice depends on the bandwidth allocation objective that the operator wishes to achieve. We pick four popular and broad bandwidth allocation policies and show how they can be expressed using utility functions below; a similar exercise can be carried out for other policies. Table 1 provides a summary.

Fairness. Various notions of fairness can be expressed simply by changing the shape of the utility functions. The α-fair [47] class of utility functions, represented in the first row of Table 1, enable an operator to express different preferences on the fairness/efficiency trade-off curve by vary-

[1]Concave utility functions ensure the optimization problem is tractable and has a unique global optimum [12]. Some practically interesting utility functions are not concave, e.g., bandwidth guarantees for inelastic flows. In such cases, global optimization is generally intractable, but under certain conditions distributed algorithms have been shown to attain the global optimum [22]. A study of non-concave utility functions is beyond the scope of this paper.

Allocation Objective	NUM objective
Flexible α-fairness [47]	$\sum_i x_i^{1-\alpha}/(1-\alpha)$
Weighted α-fairness	$\sum_i w_i^{\alpha} x_i^{1-\alpha}/(1-\alpha)$
Minimize FCT [3]	$\sum_i x_i/s_i$
Resource pooling [68]	$\sum_i y_i^{1-\alpha}/(1-\alpha)$, where $y_i = \sum_{p \in Path(i)} x_{ip}$
Bandwidth functions [35]	$\sum_i \int_0^{x_i} F_i(\tau)^{-\alpha} d\tau$

Table 1: Example utility functions for several resource allocation policies. The case $\alpha = 1$ is to be interpreted in the limit $\alpha \to 1$; e.g., $\sum_i \log x_i$ for the first row.

ing α, a non-negative constant. $\alpha = 0$ is purely utilitarian: maximize overall throughput without concern for fairness. As α increases, the NUM solution gets "more fair", eventually converging to the egalitarian max-min fair allocation as $\alpha \to \infty$. An important case is $\alpha = 1$, which is a compromise between these extremes and is called *proportional fairness*. α-fair utility functions can also be generalized to express relative priorities using different weight multipliers for different flows, as shown in the second row of Table 1.

Minimizing Flow Completion Time. Size-based scheduling policies that are effective for minimizing (average) flow completion time can also be approximated within the NUM framework, as shown in the third row of Table 1. The utility functions are linear in the rates and associate a weight to each flow inversely proportional to its size (s_i). It is not difficult to see that the solution to this problem coincides with the Shortest-Flow-First policy for flows sharing a single link. As we show in §6.3, this objective also performs very well in the multi-link case. Similarly, the weights can be chosen inversely proportional to the remaining flow size or flow deadlines to approximate Shortest-Remaining-Processing-Time (SRPT) or Earliest-Deadline-First (EDF) scheduling for meeting deadlines [23, 3].[2]

Resource Pooling. The goal of resource pooling [68] is to make a collection of network links behave as though they make up a single link with the aggregate capacity (see [68], Figure 1, for an illustration). This is useful in datacenters, where the network fabric has a large number of paths and we would like flows to use the entire pool of capacity efficiently. The Multipath TCP (MPTCP) [69, 56] congestion control algorithm has recently been proposed for achieving resource pooling. MPTCP divides a flow into sub-flows that traverse different paths and implements a coordinated congestion control across them to realize resource pooling.

It turns out that resource pooling for multipath flows can be expressed as a NUM problem, as shown by Kelly [31]. The key idea is to consider the utility for a flow in terms of the *total* rate of all its sub-flows. Any sharing/fairness objective can be generalized for multipath resource pooling in this way. The fourth row of Table 1 shows this for the α-fairness objective. Here y_i is the total rate, summed over the rates of the subflows on different paths.

[2]The linear objective is not strictly concave, hence the NUM solution may not be unique. We can avoid this in practice by using $\sum_i x_i^{1-\epsilon}/s_i$ instead, with a small ϵ such as 0.1.

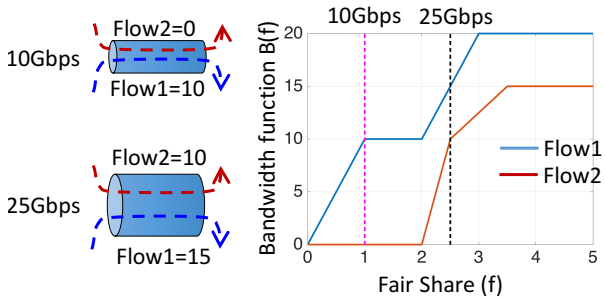


Figure 2: Flow1 (blue) and Flow2 (red) share a link and have the bandwidth functions shown. If the link speed is 10 Gbps, the blue flow gets all of the link, corresponding to a fair share of 1. But with a link speed of 25 Gbps, the blue flow gets 15 Gbps and the red flow gets 10 Gbps, for a fair share of 2.5.

Bandwidth Functions. Bandwidth functions are an intuitive abstraction for expressing bandwidth sharing policies that have been used in Google's Bandwidth Enforcer (BwE) system for their private WAN [25, 35]. A bandwidth function, $B(f)$, specifies the bandwidth to be allocated to a flow as a non-decreasing function of a dimensionless variable, f, called the *fair share*. The bandwidth function indicates the priority and weight of a flow relative to other flows for different values of f. As an illustration, Figure 2 shows the bandwidth functions of two flows. Here, flow 1 has strict priority over flow 2 for the first 10 Gbps of capacity ($f \leq 2$); beyond that, flow 2 receives bandwidth at twice the slope of flow 1 until it reaches 10 Gbps ($2 \leq f \leq 2.5$), and so on.

Formally, given the bandwidth functions, $B_i(f)$, for a set of flows sharing a single link, the bandwidth allocation is determined by finding the largest value of f such that (1) flow i is allocated bandwidth $B_i(f)$; (2) the link is not oversubscribed, i.e. $\sum_i B_i(f) \leq C$, where C is the link capacity. Computing this allocation is straight forward via a *water-filling* procedure: start with $f = 0$ and increase f until the link capacity is reached. Figure 2 shows the resulting bandwidth allocation for the two flows, when contending for a link of capacity 10 Gbps or 25 Gbps. This water-filling procedure can be generalized for an arbitrary number of links by calculating a *max-min* set of fair share values for the flows (see [35] for details).

We now show how, given a set of operator-defined bandwidth functions, $B_i(f_i)$, we can derive corresponding utility functions (shown in the last row of Table 1) such that the NUM solution achieves the desired allocation. For technical convenience, we assume that $B_i(\cdot)$ are *strictly* increasing. Let $F_i(x) \triangleq B_i^{-1}(x)$ be the inverse bandwidth function, giving the fair share as a function of allocated bandwidth. Now consider the following utility function:

$$U_i(x_i) = \int_0^{x_i} F_i(\tau)^{-\alpha} d\tau, \qquad (2)$$

where α is positive constant. $U_i(\cdot)$ is concave and increas-

ing, hence the NUM problem has a unique optimal solution. It turns out that for large α, the NUM solution is close to the allocation corresponding to the bandwidth functions.[3] Informally, the reason is that for large α, the marginal utility, $U_i'(x_i) = F_i(x_i)^{-\alpha}$, increases very sharply for smaller values of fair share, $F_i(x_i)$. Therefore, NUM favors increasing the fair share (and rate) of flows with smaller fair share. For large α, the result is an allocation that is approximately max-min in the fair shares, as desired. In practice, we find that $\alpha \approx 5$ is sufficient for very good approximation (§6.3).

3. PRIOR APPROACHES TO NUM

There are several well understood distributed algorithms for NUM [33, 29, 40] that have a structure similar to *end-to-end* congestion control. At a high level, sources determine the rates of flows based on congestion feedback from network links. Each link computes a congestion *price* based on the aggregate traffic at the link, and each source adjusts its rate (or window size in window-based schemes like TCP) based on the aggregate congestion price along its path. To make the description precise, we focus on a standard and well-studied end-to-end algorithm for NUM first proposed by Low *et al.* [40]. We call it the *Dual Gradient Descent* (DGD) algorithm for reasons which will become evident.

Dual Gradient Descent Algorithm

In the DGD algorithm, the flow rates and link prices are interpreted as the primal and dual variables of the NUM optimization problem (1). The DGD algorithm is an iterative gradient descent based procedure for computing the primal and dual optimal variables. We omit the derivation for brevity, but it can be shown that this can be done in a distributed fashion because the dual problem decomposes into independent subproblems, one for each flow, as shown in Low *et al.* [40] (Sec IV). We focus on the two key operational aspects of the algorithm: how prices at each link (represented by p_l for link l) are computed and updated at the switches, and how end-hosts compute and update their sending rates (represented by x_i for sender i).

Updating sending rates at the end-hosts. The DGD algorithm proceeds in iterations. In iteration t, given the (fixed) link prices, each flow sets its rate, x_i to:

$$x_i(t) = U_i'^{-1}\left(\sum_{l \in L(i)} p_l(t)\right), \qquad (3)$$

where $L(i)$ is the set of links on flow i's path. The rate update has an intuitive interpretation; it sets the rate of each flow to be such that the marginal utility is equal to the overall sum of prices of the links on the flow's path. Note that the flow only needs to know the overall sum of prices, not the individual prices of the links along the path.

Updating link prices at the switches. Subsequently, fixing the flow rates $x_i(t)$ for iteration t, the DGD algorithm calculates the price at each link for iteration $t+1$ using the following gradient descent step:

$$p_l(t+1) = \left[p_l(t) + \gamma\left(\sum_{i \in S(l)} x_i(t) - c_l\right)\right]_+ . \qquad (4)$$

Here, $S(l)$ is the set of flows incident on link l, and γ is the *step size*. The notation $[x]_+$ means $\max(x, 0)$. Equation (4) has an intuitive interpretation. The term $\sum_{i \in S(l)} x_i - c_l$ is the net traffic through the link minus its capacity, and turns out to be the gradient for the dual problem. The price is a measure of congestion: it increases when traffic exceeds the link's capacity and decreases otherwise. Further, the increase or decrease is controlled by the gradient. Equations (3) and (4) define the DGD algorithm. Low *et al.* [40] prove that the iterations converge to the NUM solution, provided the step size parameter is sufficiently small.

Drawbacks of DGD. The DGD algorithm is simple and elegant, but it has some key drawbacks in practice. First, it converges slowly and can take many iterations to find the optimal solution. This is important because if the underlying conditions change before the algorithm converges (e.g., a flow arrives or departs), then the algorithm is constantly trying to catch up to the new optimal allocation. Specifically, if the convergence time is greater than the coherence time of the workload, the algorithm will never converge.

Second, the DGD algorithm is very sensitive to the step size parameter, γ, in Equation (4). If γ is too small, the prices are prohibitively slow to converge, but set γ too large and the system becomes unstable and oscillatory. The "right" value of γ depends on a complex mix of network structure, flow pattern, and feedback latency. There is little theoretical guidance for tuning DGD beyond very conservative bounds that guarantee convergence [40].

4. DESIGN

NUMFabric solves NUM bandwidth allocation problems faster and more robustly than existing approaches. NUMFabric's insight is to decouple the underlying mechanisms for *maximizing network utilization* and achieving the optimal *relative rate allocation*. Existing NUM algorithms such as DGD couple these objectives and accomplish both in the price computation. Specifically, link prices in DGD directly dictate the sending rates of flows (Eq. (3)). DGD gradually adjusts the link prices (Eq. (4)) to (1) match the aggregate traffic at each bottleneck link to its capacity; (2) drive the relative rate allocations of the flows towards the optimal value. These two objectives are achieved simultaneously: as the prices react to local rate-capacity mismatches at each link, they also collectively converge to specific values such that the flows attain the optimal relative rate allocations.

The link prices in DGD essentially act both as a measure of congestion to control network utilization, and as a coordination signal between different flows to determine the relative rate allocation. This coupling makes for a brittle dynamic, where any change (e.g., a flow arrival or departure) that requires the link prices to change in order to achieve the correct relative allocation cannot occur without rate-capacity

[3] The desired allocation is achieved in the limit: $\alpha \to \infty$.

mismatches at the links. In fact, a link's price cannot change *unless* there is a rate-capacity mismatch at that link (Eq. (4)). For this reason, DGD must adjust the link prices gradually, in order to avoid under-utilization or packet drops.

NUMFabric decouples the objectives using two separate layers, as shown in Figure 1:

- The **Swift transport** (§4.1), a transport design that given a *weight* for each flow, quickly achieves the *network-wide weighted max-min fair* rate allocation for all flows.
- The **eXplicit Weight Inference (xWI)** algorithm (§4.2), a distributed algorithm that calculates the optimal weights for the flows such that the weighted max-min rate allocation (achieved by Swift) solves the NUM problem.

Swift provides the abstraction of a network with guaranteed high utilization and weighted max-min allocation [44, 9], where the flow weights can be set dynamically. This allows the relative bandwidth allocation of the flows to be controlled without having to worry about high utilization or network congestion. xWI leverages this capability to quickly search for the optimal weights and link prices, which now solely act as a coordination signal — *not* a measure of congestion — enabling xWI to converge quickly and safely.

4.1 The Swift Transport

Swift flows have a weight that is set by the source and is sent to the network in packet headers. The Swift transport uses a combination of WFQ [53, 16] in the switches and a simple rate control scheme at the end-hosts to achieve the network-wide weighted max-min rate allocation. We describe each component in turn.

Swift Switches

The switches implement a packet scheduling algorithm based on classical WFQ [53, 16]. WFQ services a set of flows contending at a link in proportion to their weights. Swift switches do the same, except that the flow's weight is allowed to change on a packet-by-packet basis. We leave the details of the algorithm and its practical realization using recently proposed hardware mechanisms [59] to §5.

Swift Rate Control

WFQ achieves weighted max-min allocation for a *single link*. To achieve *network-wide* weighted max-min allocation, each flow must also send traffic at the rate dictated by the WFQ scheduler at its bottleneck link. If a flow sends below or above the rate that WFQ allows, it can under-utilize the available capacity or cause packet drops.

We design a simple window-based rate control scheme to achieve network-wide weighted max-min. The rate control algorithm has two requirements. First, it must set the window size to be larger than the bandwidth-delay product (BDP) for the flow, so that the flow's rate is not limited by its window. Second, it must keep the buffer occupancy small at the switches. This is important for fast convergence: large buffer occupancies slow down convergence to weighted max-min when the set of flows (or their weights) change, because large buffers may take a long time to drain when a flow's bottleneck shifts from one link to another.

Our rate control algorithm is inspired by packet-pair [34] and packet-train [13] techniques for estimating available bandwidth in a network of switches with WFQ packet scheduling. The receiver measures the inter-packet time for each incoming packet, and sends this value to the sender in acknowledgments. Upon receiving an ACK, the sender calculates a rate sample, `bytesAcked / interPacketTime`, and smoothens these values using an exponentially weighted moving averaging (EWMA) filter [62] to estimate the available bandwidth, $\hat{R}$. The sender then sets its window size to $W = \hat{R} \times (d_0 + d_t)$, where d_0 is the baseline fabric RTT (without queuing delay) and d_t is a small slack factor (e.g., a few packets worth of delay) chosen to ensure that the window size is larger than the BDP (estimated by $\hat{R} \times d_0$).

To start, the sender initially sends a small burst (e.g., 3 packets in our implementation) into the network. This burst ensures that packets are queued at the bottleneck, and thus the inter-packet time observations at the receiver reflect the true available bandwidth. Upon receiving the first inter-packet time sample,[4] the sender initializes $\hat{R}$; thereafter, it updates $\hat{R}$ and the window size as explained above.

4.2 The xWI Algorithm

xWI is a novel distributed algorithm for solving NUM problems that runs on top of a weighted max-min achieving transport layer like Swift. xWI is inspired by iterative message passing [43] algorithms. In each iteration, sources exchange messages with switches to compute the weights to set for their flows in Swift. xWI iteratively refines the weights to arrive at the optimal NUM allocation.

The key idea in xWI is to iteratively solve the KKT system of equations [12] for the NUM problem:

$$U_i'(x_i) = \sum_{l \in L(i)} p_l \qquad \text{for all flows } i, \qquad (5)$$

$$p_l \left(\sum_{i \in S(l)} x_i - c_l \right) = 0 \qquad \text{for all links } l, \qquad (6)$$

where x_i are the flow rates, and p_l are the link prices. Intuitively, the first condition implies that at the optimal point, a flow's marginal utility is equal to the total price it must pay on the path it is traversing. The second condition implies that either a link is fully utilized (i.e. the sum of the rates of flows on that link is equal to the link capacity), or if it is underutilized, the link price is zero. The flow rates and link prices are optimal iff they satisfy the above equations and are *feasible*, i.e., $\mathbf{Rx} \leq \mathbf{c}$, $\mathbf{p} \geq \mathbf{0}$ [12, 61].

Flow weight assignment and rate allocation. Recall the rate assignment step in the DGD algorithm (Eq. (3)), which sets the rate of each flow based on the sum of the link prices on its path. xWI uses the same function of the sum of the link prices, but uses it to set the flow's *weight* to be used by Swift, *not its rate*. Specifically, in iteration t, the flow's

[4]The first inter-packet time arrives with the second ACK (following the standard three-way handshake). The sender ignores the first ACK and sends nothing.

weight is assigned as

$$w_i(t) = U_i'^{-1}\left(\sum_{l \in L(i)} p_l(t)\right). \quad (7)$$

The Swift transport then takes these weights and allocates rates to all flows according to weighted max-min:

$$\{w_i(t)\} \xrightarrow{\text{weighted max-min}} \{x_i(t)\} \quad (8)$$

The intuition for this step is that the above function of the link prices calculates rates such that Eq. (5) is satisfied. If the prices are at the optimal values, this calculation gives the optimal rates, but generally, the calculated values are not optimal (or even feasible) rates for incorrect prices. Here, using weights has a crucial advantage: it lets Swift find a feasible and efficient allocation that approximately satisfies Eq. (5) even if the link prices are not optimal. By contrast, assigning rates (as done in DGD), can cause over- or under-utilization if the link prices have not converged to the optimal values. As the prices reach the optimal values, the weights computed by Eq. (7) will be the same as the optimal rates for the NUM problem; and Eq. (5) will be satisfied exactly.

Price computation. Next, link prices at iteration $t+1$ are updated based on the values (link prices and flow rates) in iteration t to approach the optimal values. For this purpose, each link independently updates its price towards satisfying the two optimality conditions in Eqs. (5) and (6).

The update rule consists of two terms, corresponding to the two optimality conditions. The first term is given by

$$p_l^{res} \triangleq p_l(t) + \min_{i \in S(l)} \left(\frac{U_i'(x_i(t)) - \sum_{k \in L(i)} p_k(t)}{|L(i)|} \right), \quad (9)$$

and tries to satisfy the system of equations (5). Here, $|L(i)|$ denotes the number of links in flow i's path. Notice that for each flow passing through link l, the corresponding equation in (5) has a *residual*: $U_i'(x_i(t)) - \sum_{k \in L(i)} p_k(t)$. The intuition behind Eq. (9) is to adjust the link prices to push these residual values to zero as much as possible. To see how, consider a flow with residual e traversing L links. If each link on the flow's path adds e/L to its price, the residual will become zero. Equation (9) applies the same idea, except that each link adjusts its price based on the smallest residual, such that after the update, the residual for the flow with the smallest residual before the update becomes zero.

For the second optimality condition (Eq. (6)), notice that Swift's weighted max-min allocation ensures that the link capacities are never exceeded; i.e. $\sum_{i \in S(l)} x_i(t) \leq c_l$, for all links l and iterations t. Equation (6) is automatically satisfied for *bottleneck* links, for which $\sum_{i \in S(l)} x_i(t) = c_l$. For *underutilized* links, however, the price must be driven to zero to satisfy (6). We achieve this by subtracting a term based on underutilization from p_l^{res}:

$$p_l^{new} \triangleq \left[p_l^{res} - \eta \left(1 - \frac{\sum_{i \in S(l)} x_i(t)}{c_l}\right) p_l(t) \right]_+, \quad (10)$$

where η is a positive constant. The parameter η may appear similar to the step size parameter γ of the DGD algorithm in Eq. (4). But η has a much less crucial role: it only kicks in for underutilized links to drive the price to zero. The second term above will be zero for all bottlenecks links at all times. Therefore, xWI is largely insensitive to the value of η.

The final refinement is an *averaging* of the new price estimate and the current value, which is a standard technique [24, 38, 63] for ensuring such non-linear dynamical systems converge to a fixed point:

$$p_l(t+1) = \beta p_l(t) + (1-\beta) p_l^{new}. \quad (11)$$

Here, $\beta \in (0,1)$ is the averaging parameter (set to 0.5 in our implementation). We have found averaging to be important for improving system stability, particularly in the presence of noise (e.g., due to traffic burstiness, measurement noise, etc) in our packet-level simulations.

We have proven that the xWI dynamical system has a unique fixed point, and this fixed point solves the NUM optimization problem (1). We have also conducted extensive numerical simulations of the algorithm, and found that xWI converges to the NUM optimal solution across a wide range of randomly generated topologies and flow patterns. We leave these results to the extended version of this paper [48]. In §6, we evaluate NUMFabric using packet level simulations for realistic datacenter topologies and workloads.

Distributed realization. xWI can be implemented in a completely decentralized fashion. The flow weight calculation in Eq. (7) requires the sources to know the sum of the link prices along their path, which can be obtained using end-to-end feedback (same as in DGD). Swift then realizes the weighted max-min allocation in Eq. (8) as described in §4.1. For the price computation, to evaluate p_l^{res} in Eq. (9), the switches need to know the *normalized residual*, i.e. residual divided by path length: $\left(U_i'(x_i(t)) - \sum_{k \in L(i)} p_l(t)\right)/|L(i)|$, for each flow through the link. Each flow calculates this value and sends it to the switches on its path in packet headers. Finally, the underutilization term in Eq. (10) only requires local information: the total traffic through the link and the current link price. We describe the NUMFabric protocol that implements xWI in detail next.

5. NUMFABRIC'S PRACTICAL DESIGN

We functionally sketch out the actions at the receiver, the sender, and the switch. NUMFabric adds five fields to packet headers in the transport layer: `virtualPacketLen` and `interPacketTime` for Swift; `pathPrice`, `pathLen`, and `normalizedResidual` for xWI.

The NUMFabric Receiver and Sender

The receiver gets the sum of the link prices and the number of links on the path from the `pathPrice` and `pathLen` fields in received packets. It then simply reflects these values and the latest inter-packet time (used by Swift's rate control algorithm; see §4.1) back to the sender in ACKs.

The sender maintains the inverse of the marginal utility function, $U_i'^{-1}(\cdot)$, and uses the `pathPrice` obtained from

each ACK to compute the flow's weight (Eq. (7)). The weight is used to set the `virtualPacketLen` field for outgoing packets as the packet length divided by the weight (we discuss how switches use this field below). The sender also sets the `normalizedResidual` field in each outgoing packet as the marginal utility minus the path price divided by the path length (see §4.2). The marginal utility is calculated as $U_i'(\hat{R})$, where $\hat{R}$ is the current estimated rate of the flow. As explained in §4.1, $\hat{R}$ is computed by applying an EWMA filter to the inter-packet times obtained from ACKs and is also used to set the flow's window size: $W = \hat{R} \times (d_0 + d_t)$. For control packets such as SYNs and pure ACKs, the `virtualPacketLen` field is set to zero, and the `normalizedResidual` field is ignored by the switches.

The NUMFabric Switch

The switch implements WFQ packet scheduling (§4.1) and the xWI price computation (§4.2).

NUMFabric's WFQ design. The full hardware implementation of WFQ for NUMFabric is not the focus of this paper. We briefly sketch out a design conceptually based on Start Time Fair Queuing (STFQ) [20]. STFQ approximates the order with which packets would depart in WFQ by assigning a *virtual start and virtual finish time* to each packet. It then schedules packets in ascending order of virtual start time. Let p_i^k be the k-th packet of flow i. Upon p_i^k's arrival, STFQ computes the virtual start and finish time as follows:

$$S(p_i^k) = \max\left(V, F(p_i^{k-1})\right), \tag{12}$$

$$F(p_i^k) = S(p_i^k) + \frac{L(p_i^k)}{w_i}. \tag{13}$$

Here, $L(p_i^k)$ is the length of packet p_i^k and w_i is the weight for flow i. V is the *virtual time* at the switch at the time of packet p_i^k's arrival (see [19, 20] for details). To compute these values, the switch maintains V (a single register), as well as the virtual finish time of the last packet, $F(p_i^{k-1})$, for each active flow. The `virtualPacketLen` field of packet p_i^k provides $L(p_i^k)/w_i$, which the switch uses for Eq. (13).

The design sketched above requires a priority queue to schedule packets in increasing order of virtual start times. While today's switches do not support priority queues, recent work [59, 60] has shown that it is feasible to implement programmable priority queues in emerging programmable switching chips [11]. This design allows packets to be inserted into the queue based on a programmable rank value that is computed before the packet is enqueued. The paper [60] shows how STFQ can be realized with this design. We omit the details and refer the reader to [60].

xWI price computation. The switch computes and updates the price for each of its outgoing links periodically. We assume the price updates are synchronized at all switches. This can be accomplished in datacenters with Precision Time Protocol [37], which is now a common feature in commodity switches [46]. The price is computed as shown in Figure 3. The procedure is a faithful implementation of the xWI price calculation described in §4.2.

```
# enqueueing packet p upon arrival
def enqueue(p):
  if p is DATA: # not control (e.g., SYN)
    minRes = min(p.normalizedResidual, minRes)

# dequeueing packet p for departure
def dequeue(p):
  bytesServiced += p.length
  p.pathPrice += price
  p.pathLen += 1

# price update timeout
def priceUpdateTimeout():
  u = bytesServiced / (priceUpdateTime *
      linkCapacity)  # link utilization
  newPrice = max(price + minRes - eta * (1 - u) *
       price, 0)
  price = beta * price + (1 - beta) * newPrice
  bytesServiced = 0
  minRes = inf
```

Figure 3: Price computation in the NUMFabric switch.

6. EVALUATION

In this section, we present an extensive ns3 simulation-based evaluation of NUMFabric. The goal is to evaluate NUMFabric's (1) *fast convergence* to the optimal allocation in dynamic settings; and its (2) *flexibility* for meeting various bandwidth allocation objectives precisely and robustly. The code used for all our simulations can be found at [50].

Topology. We simulate a data center network built using a leaf-spine architecture. There are a total of 128 servers connected to 8 leaf switches with 10 Gbps links. Each leaf switch is connected to 4 spine switches using 40 Gbps links, thus ensuring full bisection bandwidth. The switches are modeled as standard output-queued switches, with a buffer of size 1 MB per port. We chose this large limit to avoid complications for comparing the convergence times of different algorithms which are sensitive to packet drops. All of the implemented schemes target a small queue occupancy, and thus avoid packet drops well below this buffer size; the queue occupancies are typically only a few packets at equilibrium. The network RTT is 16 μs.

Schemes compared. We have implemented the following: *DGD rate control,* an idealized rate control protocol based on the Dual Gradient Descent algorithm described in §3. The sources calculate their sending rates from the network price (obtained from ACKs) according to Eq. (3). They then transmit at exactly this rate on a packet-by-packet basis. The switches implement a price update rule similar to Eq. (4), with an additional term to control the queue occupancy:

$$p_l \leftarrow \left[p_l + a(y - C) + bq\right]_+. \tag{14}$$

Here, y is the link throughput, C is the link capacity, q is the queue occupancy, and a and b are constant parameters. The price is updated periodically.

*RCP**[30], a generalization of RCP [18] for α-fairness [47].[5] As in standard RCP, switches in RCP* allocate a *fair share rate* to all flows passing through each of their links. The fair

[5]Standard RCP achieves max-min fairness [18].

share is updated periodically according to:

$$R_l \leftarrow R_l \left(1 + \frac{\frac{T}{d}\left(a(C-y) - b\frac{q}{d}\right)}{C}\right). \quad (15)$$

Here, R_l is the fair-share rate that is advertised by the switch to flows passing though link l, T is the update interval, and d is the running average of the RTT of the flows. C, y, and q have the same interpretation as in DGD. When a packet is served by link l, $R_l^{-\alpha}$ is added to a field in the packet header (similar to the path price field in DGD). The source calculates the sending rate for flow i as follows:

$$x_i = \left(\sum_{l \in L(i)} R_l^{-\alpha}\right)^{\frac{-1}{\alpha}}. \quad (16)$$

The sum is over the links $L(i)$ on flow i's path and is obtained through ACKs. RCP* is similar to DGD since both algorithms set the sending rates directly based on explicit network feedback in the form of a sum over per-link variables that depend on link congestion.

NUMFabric, our design as described in §5.

Oracle, a numerical fluid model simulation that takes the current network state, including the topology and current set of flows, as input and outputs the optimal rate allocation according to the NUM problem. We use the Oracle to test the correctness and speed of convergence of the NUMFabric, DGD and RCP* algorithms.

Table 2 shows the default parameter settings for all schemes.

Scheme	Parameters
DGD [Eq. (14)]	`priceUpdateInterval` = 16 μs `a` = 4×10^{-9} Mbps^{-1} `b` = 1.2×10^{-10} B^{-1}
RCP* [Eq. (15)]	`rateUpdateInterval` = 16 μs `a` = 3.6 `b` = 1.8
NUMFabric [§5]	`ewmaTime` = 20 μs `dt` = 6 μs `priceUpdateInterval` = 30 μs `eta` = 5 [Eq. (10)] `beta` = 0.5 [Eq. (11)]

Table 2: Default parameter settings in simulations.

Note on the implementation of DGD and RCP*. We enhance DGD and RCP* to limit the number of unacknowledged bytes that flows can have to 2 $\times$ the Bandwidth-Delay Product. This ensures that flows are large enough to saturate the network yet restricts them from building up large queues when the rates have not converged to the correct values. Large queues adversely affect stability and convergence times for both DGD and RCP* by increasing feedback delay. Therefore, the results that we report are better than what can be achieved with the standard rate-based implementation of these schemes. Also, the performance of DGD and RCP* is very sensitive to the gain parameters associated with utilization and queue occupancy (a and b). We swept across the parameter space and picked parameters that gave the fastest convergence time while maintaining stability.

6.1 Convergence to NUM Allocations

We design two scenarios to quantify the speed of convergence: *semi-dynamic*i scenarios where we can inject network events in a controlled manner and measure convergence times accurately, and realistic *dynamic* scenarios from measured datacenter workloads.

Semi-dynamic Workload

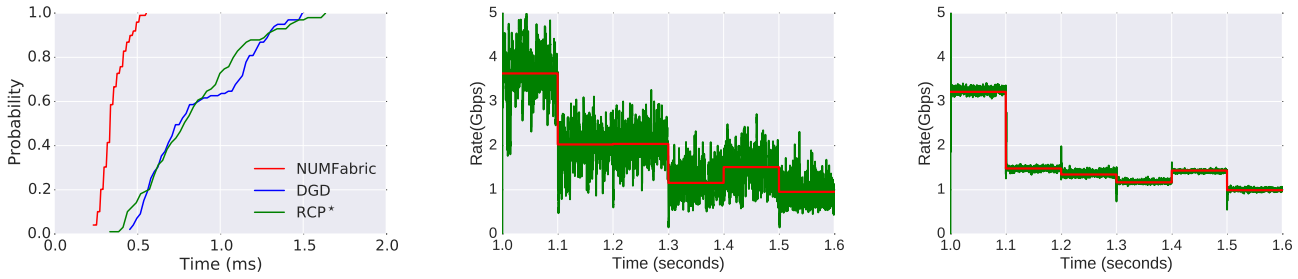
For this experiment, we randomly pair 1000 senders and receivers among the 128 servers in our network to create 1000 random flow paths. To inject dynamism, we create "network events," where we randomly choose 100 of these paths to either start 100 new flows or stop 100 active flows. For each network event, we define the convergence time as the time it takes for the rates of at least 95% of the flows to reach within 10% of the optimal NUM allocation (which we compute using the Oracle mentioned earlier). We also make sure that the flows stay within this margin for at least 5 ms before declaring they have converged. Once the rates of 95% of the flows converge, we trigger the next flow start/stop event. We ensure that there are 300-500 flows active after each event, and simulate 100 such events. The NUM objective for these experiments is proportional fairness: $\sum_i \log(x_i)$.

Measuring convergence time accurately at microsecond timescales is tricky. At these timescales, rate measurements are noisy because of small variations in inter-packet times (e.g., due to bursty packet scheduling at the switches [8]). To overcome the noise, we use exponential averaging [62] with a time constant of 80 μs to filter the rates measured at the destination. It takes $\log_e(10) \times 80 \approx 185$ μs for the filter's output to reach 90% of its final value. We subtract this additional delay from all measured convergence times since it's an artifact of our measurement that would exist even if the flows converged instantaneously.

Figure 4(a) compares the convergence times for NUMFabric, DGD and RCP*. NUMFabric converges in 335 μs at the median: $\sim 2.3\times$ faster than DGD and RCP*. At the 95^{th} percentile, NUMFabric converges in 495 μs: $\sim 2.7\times$ faster than the other schemes. It is important to note that these values are for 95% of a few hundred flows to converge after a network event. NUMFabric's convergence time for individual flows is much lower.

The primary reason for NUMFabric's fast convergence is the agility that the decoupled combination of Swift's convergence to weighted max-min and xWI's fast computation of link prices provides. Since DGD and RCP* have to optimize both objectives simultaneously, they move towards the optimal allocation more gingerly.

Comparison with DCTCP. We were also hoping to compare NUMFabric with a deployed congestion control algorithm like DCTCP. However, although the rates achieved by DCTCP flows are stable when averaged over longer timescales (several milliseconds), they are very noisy at timescales of 100s of microseconds. As a result, DCTCP flows essentially never converge, unless we measure the rates over timescales much longer than the convergence time of the other algorithms. For example, Figure 4(b) shows the rates achieved by a typical DCTCP flow during several network



(a) CDF of convergence time (b) Rate of a typical DCTCP flow (c) Rate of a typical NUMFabric flow

Figure 4: Convergence behavior of NUMFabric, DGD, RCP* and DCTCP in semi-dynamic scenario. The rates are measured using an EWMA filter with a 80 μs time constant. To calculate the convergence time, we subtract 185 μs (the rise time of the filter) from the measured convergence times.

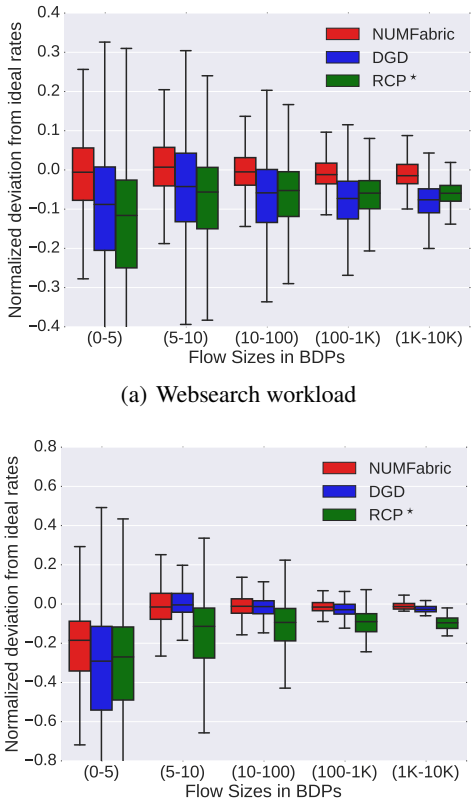
events. The expected rate of the flow is shown by the solid red line. It is clear from the figure that DCTCP flows will never converge to within 10% of the expected rate. The rates achieved by the corresponding flow under NUMFabric are shown in Figure 4(c). Note that the expected rates for the same flow are different in these figures since DCTCP does not optimize for proportional fairness.

Dynamic Workloads

Next, we evaluate NUMFabric's fast convergence for more realistic dynamic settings. We consider two workloads based on measurements from a web search [3] cluster and a large enterprise [4]. In the web search workload, about 50% of the flows are smaller than 100 KB, but 95% of all bytes belong to the larger 30% of the flows that are larger than 1 MB. The enterprise workload is also heavy-tailed, but has many more short flows with 95% of the flows smaller than 10 KB. The flows arrive as a Poisson process of different rates to simulate different load levels.

It is difficult to define convergence time for such dynamic workloads since a majority of the flows finish before they converge (esp. flows smaller than a BDP). Hence we compare the average rates of the flows achieve to what they would have achieved with an ideal Oracle that assigns all flows their optimal NUM rates instantaneously. Specifically, we calculate the rate of a flow as its size divided by its completion time, and calculate the normalized deviation from ideal for scheme `X` as $(\texttt{rateWithX} - \texttt{idealRate})/\texttt{idealRate}$, where `idealRate` is the rate of the flow with the Oracle. For example, a normalized rate deviation of +1 means that the flow's rate is 2× larger than its rate with the Oracle, while negative values indicate that the flow's rate was lower than with the Oracle.

We bin the flows into different sizes and compare the rate deviation for each bin separately. The bin boundaries are chosen log-scale in the bandwidth-delay product (BDP), which is 200 KB in our network. Figure 5(a) shows the normalized rate deviation over all flows belonging to each bin for the web search workload. The smaller flows have larger errors for DGD and RCP* since the flows may not last enough RTTs for these schemes to converge. NUMFabric achieves

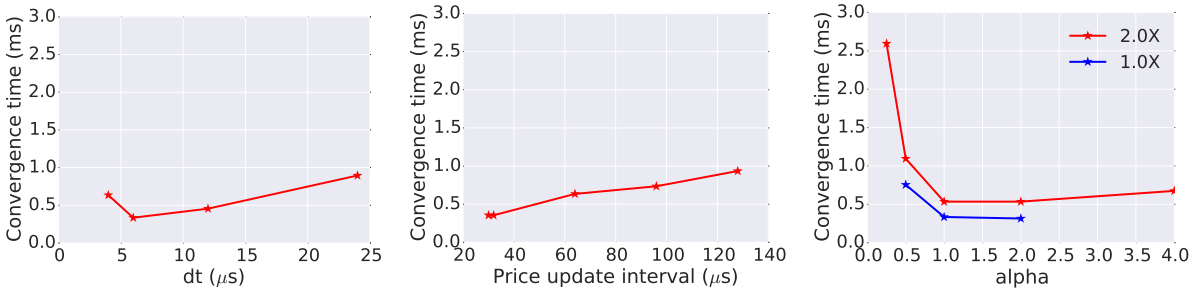


(a) Websearch workload

(b) Enterprise workload

Figure 5: Deviation from ideal rates for NUMFabric, DGD and RCP* in two dynamic workloads. The box shows the 25th and 75th percentiles and whiskers extend to show 1.5 times the box length. Outliers are omitted.

rates fairly close to the rates achieved by the Oracle even in this bin. As the flow sizes increase, NUMFabric's faster convergence brings it closer to the Oracle's rates while DGD and RCP* still lag. The median error of NUMFabric is around zero for all the bins beyond a flow size of 100 KB. This means that under NUMFabric, flows with size above ∼5 BDP converged on the average. One interesting observation here is that the median errors of DGD and RCP* are nega-



(a) Sensitivity to d_t (b) Sensitivity to price update interval (c) Sensitivity to α

Figure 6: Median convergence times as we vary difference parameters in NUMFabric. The 2× line in part (c) is for NUMFabric slowed down by a factor of 2.

tively biased. This indicates that the flows under DGD and RCP* achieved lower than the ideal rates. This is a direct consequence of the slow convergence of these algorithms and the consequent inability to grab available bandwidth.

Figure 5(b) plots the performance for the enterprise workload. This workload is more skewed, nearly 70% of the flows send 1 or 2 packets. Convergence for such flows is meaningless since the notion of a rate itself is hard to define. This clearly shows up in the plots: the deviation for the very small flows is relatively large for all 3 schemes. The median deviation from ideal for NUMFabric is nearly zero for all other bins for this workload as well.

6.2 Parameter Sensitivity Analysis

NUMFabric performs well for a wide range of values of its parameters, but requires tuning to extract the best performance. We present some insights for tuning the parameters of the two layers of NUMFabric, Swift and xWI, and the interplay between them. We also use the semi-dynamic scenario designed in §6.1 to quantify parameter sensitivity.

Swift parameters

The Swift sender applies an EWMA filter to the inter-packet times reported with each ACK to estimate the flow's rate. The EWMA filter's time constant, `ewmaTime`, determines how fast or slow the rate estimate matches the available capacity at the bottleneck. This parameter should be set large enough for the filter to "span" enough samples to get a reliable estimate; otherwise, the rate estimate can be noisy due to natural variations in inter-packet times. The noise causes undesirable oscillations in Swift's window size as well as the normalized residual calculation discussed in §5. We found an `ewmaTime` of 20 μs or more is required for good stability in a 10 Gbps network. The higher the link speed, the faster we can collect inter-packet time samples and the faster we can run the filter.

The delay slack parameter, d_t, used by Swift to calculate the window size (§4.1) exhibits a trade-off: setting it too small risks underutilizing the available bandwidth, but if too large, it causes a large buffer occupancy and slows down convergence. In Figure 6(a), we vary the value of d_t from 3 μs to 24 μs. We observed that with $d_t = 3$ μs, many events don't converge at all — the plot shows the median for events that did converge. This is because for WFQ to work correctly, each flow must have at least 1 packet queued at its bottleneck link at all times. With a very small d_t, the window sizes become so small that this condition is sometimes not satisfied. On the other hand, a very high d_t also leads to slower convergence and sometimes oscillating rates. We find that it is sufficient to set d_t to the delay-equivalent of a small number (e.g., 5-10) packets. For example, for our 10Gbps network, we use $d_t = 6$ μs which targets a buffer occupancy of 5 packets (1500 bytes each) at every bottleneck link.

xWI parameters

Link prices are updated periodically to reflect changes in the normalized residual values of flows and utilization of links since the last price update. Once a link price is updated, the switch has to wait for the sources to learn of the new prices, adjust the flow's weight, and measure the impact on the rate achieved by Swift to calculate the new normalized residual values (§5). There is no benefit, and indeed it can be detrimental to stability, to update prices before Swift has had time to react. This process takes *at least* 1.5-2 RTTs: 1 RTT for the weights to reflect the new prices; and 0.5-1 RTTs for Swift's WFQ and rate control mechanisms to achieve the new weighted max-min allocation (and measure it). In practice, achieving and measuring the new weighted max-min allocation can take longer because of dependencies among flows for convergence[6] and EWMA filtering for rate measurement. While worst-case bounds can be found for convergence to weighted max-min [57], they depend on the traffic pattern, and our experiments indicate the bounds are overly conservative for NUMFabric.

In Figure 6(b), we measure the impact of the price update interval on NUMFabric's convergence by varying it from 30 μs to 128 μs. As expected, the median convergence time increases as the price update interval increases. In practice, we find that a price update interval of around 2 RTTs usually works well and is fast; increasing it beyond 2 RTTs improves robustness, but is slower, while decreasing it much below 2 RTTs degrades convergence and is not recommended.

The two other parameters of xWI are the multiplier for the

[6]See [57] for an analysis of *dependency chains* during convergence to max-min.

utilization term (η; Eq. (10)) and the averaging parameter (β; Eq. (11)) for the price calculation. xWI is largely insensitive to these parameters. We use $\beta = 0.5$ and $\eta = 5$.

Different utility functions

We also tested NUMFabric with different utility functions by varying α in the α-fair family of utility functions. We found that while the default parameters of NUMFabric work fine for moderate values of α, for α below 0.5 or above 2.0, some events in the semi-dynamic scenario don't converge. The reason is that, at more extreme values α, some of the calculations become numerically unstable; e.g., with very low values of α, the weight calculation in Eq. (7) is very sensitive to noise. Therefore, a faster control loop that does not smooth over enough rate samples to filter out the noise can cause oscillations. To test this, we slowed down NUMFabric 2×, by increasing the price update interval to 60 μs and `ewmaTime` to 40 μs, and found that NUMFabric converges for nearly all events at all values of α. As Figure 6(c) shows, this slowdown comes at a modest cost to the median convergence time.

We repeated the same experiment with lower α with DGD and RCP*, and found that the algorithms do not converge for values of α lower than 0.5 with the default parameters. We could not get either algorithm to converge reliably even after slowing down the gain factors by 4×.

6.3 Flexibility

In this section, we experimentally evaluate NUMFabric's ability to achieve a wide variety of bandwidth allocation objectives. We focus on three policies discussed in §2 which have been the focus of much recent work: minimizing flow completion times [3], resource pooling [56], and bandwidth allocation objectives specified via bandwidth functions as in the recent BwE system [35].

Minimizing Flow Completion Time (FCT)

As discussed in §2, the utility function: $\sum_i (1/s_i) x_i^{1-\epsilon}$, where s_i is the flow size and ϵ is a small constant approximates the Shortest-Flow-First allocation policy for minimizing FCT. We use $\epsilon = 0.125$; note that this objective is similar to α-fairness with $\alpha = 0.125$. For NUMFabric to converge to optimal values for such a small α, we slow down the system 2× as described in § 6.2.[7] We compare NUMFabric with pFabric [3], the state-of-the-art transport for FCT minimization. We replicate the same evaluation topology and workload as in the pFabric paper [3] (the web search workload), and compare the FCTs achieved by NUMFabric and pFabric as network load varies from 20% to 80%. Figure 7 shows the results. We observe that NUMFabric achieves FCTs close to pFabric. The average normalized FCT of NUMFabric is within 4-20% of pFabric under different load conditions.

Resource Pooling

Next, we evaluate if NUMFabric can achieve a resource pooling bandwidth allocation objective. We use a leaf-spine topology with 128 servers, 8 leaf switches and 16 spine switches. All links have 10 Gbps speed. We use a permutation traffic pattern exactly as in the MPTCP paper [56], where servers 1–64 each send to one server among 65–128. Each source destination pair *flow* uses one or more *sub-flows*, with each sub-flow hashed onto a path at random. We vary the number of sub-flows between each source-destination pair and plot the total achieved throughput and the per-flow throughputs in Figure 8.

[7] Mimicking pFabric [3], we also set the initial window size to be equal to the BDP, so that short flows are able to send enough data in the first RTT.

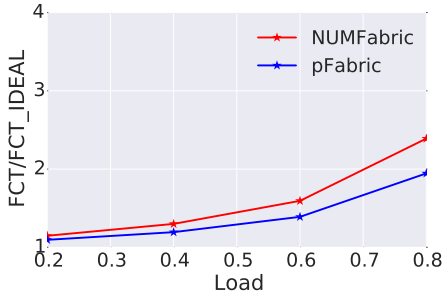


Figure 7: NUMFabric with the FCT minimization utility function vs. pFabric. The results are normalized to the lowest possible FCT for each flow given its size.

We consider two utility functions: (1) No Resource Pooling, which is just proportional fairness at the sub-flow level; (2) Resource Pooling, which aims for proportional fairness at the level of the aggregate rates of the source-destination pairs (summed across all their sub-flows). This is the utility function shown in the fourth row of Table 1 with $\alpha = 1$.

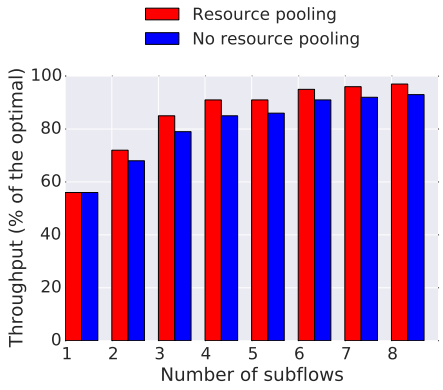
To implement this utility function, we need a small change to the NUMFabric sender. Specifically, every sub-flow first computes a weight according to Eq. (7) based on the aggregate price on its path. It turns out, however, that this value corresponds to the total weight for the entire flow between the source-destination pair, from the perspective of the price (congestion) on the sub-flow's path. To compute the weight for the sub-flow itself, we multiply the total weight by the fraction of the total throughput being served by the sub-flow. This intuitive heuristic works well in our experiments, and we leave its detailed analysis to future work.

Figure 8(a) shows that as the number of sub-flows increases to 8, NUMFabric with resource pooling achieves close to the optimal throughput of 1. Moreover, as shown in Figure 8(b), the allocation is extremely fair across all flows (source-destination pairs). This is despite the fact that the sub-flows of these flows are mapped randomly to different paths, and thus compete with different numbers of other sub-flows. The flow-level fairness is a direct consequence of the resource pooling objective. In fact, without resource pooling, the allocation is much less fair across flows.

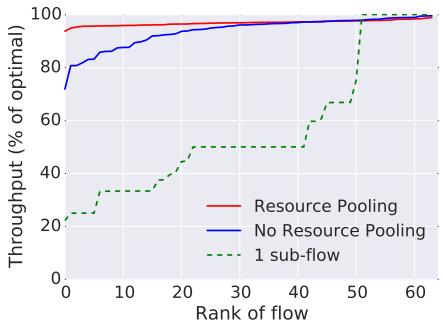
Bandwidth Functions

Next, we evaluate how well NUMFabric can achieve more complex bandwidth allocation objectives specified by bandwidth functions [35]. Specifically, we use the utility function derived in §2 to closely approximate bandwidth functions.

We use the same scenario shown in Figure 2 (originally described in the BwE paper [35]). Two flows with the band-



(a) Total throughput



(b) Flow-level fairness

Figure 8: Resource pooling: (a) Total throughput achieved as we increase the number of sub-flows per flow (b) Flows ranked by throughput.

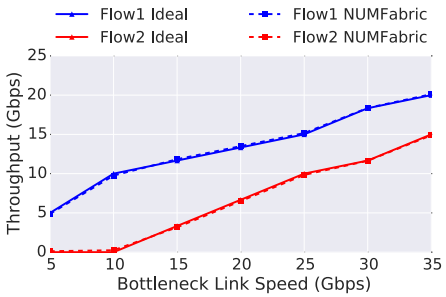


Figure 9: The throughput achieved by two competing flows on a bottleneck link with varying capacity. The bandwidth functions of the flows are shown in Figure 2.

width function shown in Figure 2 are competing on a link. We vary the link capacity from 5 Gbps to 35 Gbps, and plot the throughput achieved by each flow in Figure 9. The results show that NUMFabric's allocation is almost identical to the expected allocation at all link capacities.

Bandwidth Functions & Resource Pooling

An interesting question is whether we can combine bandwidth functions and resource pooling. In other words, can operators specify bandwidth functions for flows over a network that is also expected to provide resource pooling? Such a system doesn't exist to the best of our knowledge, but would be quite useful in datacenters. To implement this,

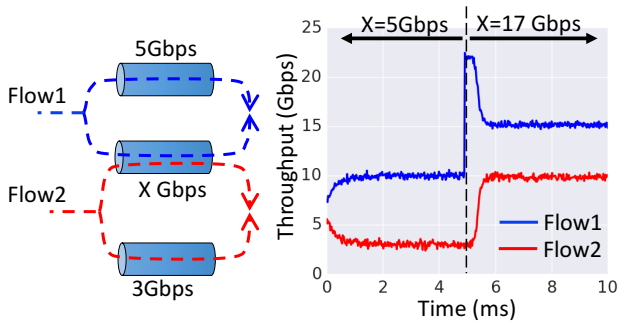


Figure 10: The middle link in the topology has variable link capacity (X), which is initially set to 5 Gbps and changes to 17 Gbps after 5 ms. The plot shows the time series of the *aggregate* throughput achieved by the two flows before and after the capacity change. The two flows use the bandwidth functions shown in Figure 2.

we modify the utility function modeling the bandwidth function to consider the aggregate throughput achieved by a flow across all its sub-flows, instead of the throughput of individual sub-flows. As in the previous experiment, we have two flows over a topology of three links as shown in Figure 10. The link in the middle is shared by the two flows, while only Flow 1 sends on the top link and only Flow 2 sends on the bottom link. These two flows use the same bandwidth functions as in previous section (shown in Figure 2).

To evaluate how well NUMFabric can combine bandwidth functions and resource pooling, we choose a scenario where we change the capacity of the middle link from $X = 5$ Gbps to $X = 17$ Gbps, after 5 ms of simulation. Figure 10 shows the throughput of each flow before and after this transition. Initially, when the speed of the middle link is 5 Gbps, we should expect an overall allocation of 10 Gbps to Flow 1 and 3 Gbps to Flow 2 according to the bandwidth function. This would imply that the middle link is solely used for Flow 1's traffic as Figure 10 shows. When the middle link speed increases to 17 Gbps, we would expect 15 Gbps for Flow 1 and 10 Gbps for Flow 2 given their bandwidth functions. NUMFabric quickly achieves the correct bandwidth allocation as shown in the figure.

7. RELATED WORK

Datacenter Transport. Most existing datacenter transport techniques focus on a specific allocation objective, e.g., meeting deadlines [67, 64], minimizing flow completion times [3, 23], network latency [1, 2], providing bandwidth guarantees [58, 7, 55, 26] or improving convergence times [27]. Our design is most closely related to pFabric [3], which also uses in-network packet scheduling to decouple the network's scheduling policy from rate control. However, pFabric only supports the SRPT scheduling policy for minimizing FCT. NUMFabric supports *any* policy which can be expressed as a NUM problem, including minimizing FCT. Our simulations show that NUMFabric has nearly as good performance as pFabric while being much more flexible.

Fastpass [54] is a centralized arbiter for scheduling *every* packet in the datacenter. This gives flexibility to implement

any network resource allocation policy. However, a centralized scheme is inherently prone to scaling problems due to (1) the large scale and churn in datacenter networks which will stress the communication and computation capabilities of the centralized arbiter; and (2) the overhead of communicating with a centralized arbiter for very short flows.

The XCP protocol [28] designed in the Internet context is also based on the insight that utilization control and fairness should be decoupled. XCP, however, decouples these functions *within* the rate control layer, and is therefore similar to other gradient-based designs which need many iterations to converge. It is also not designed for flexible objectives.

FCP [21] is a congestion control algorithm that provides flexibility to end-hosts to realize different objectives for resource allocation among flows originating from the same end-host. Realizing network-wide objectives in FCP requires modifying the switch algorithms to expose differential pricing tailored for specific objectives. NUMFabric supports a much wider range of network-level objectives (e.g., flexible bandwidth functions [35]) in a unified framework.

Packet scheduling. The PIFO programmable scheduler [60] demonstrates the feasibility of implementing programmable priority queues in modern switch hardware, which can be used to realize NUMFabric's Swift transport fabric, (§5). Universal Packet Scheduling [45] attempts to realize *any* given packet schedule with a single scheduling algorithm in the switches (which coincidentally also requires a priority queue). Although promising, flexible packet scheduling alone cannot achieve arbitrary network-wide objectives such as α-fairness, resource pooling, and bandwidth functions (§2). These require coordination among competing flows, which is exactly what NUMFabric achieves via link prices and the xWI algorithm (§4.2).

Network Utility Maximization. NUMFabric builds on a long line of work that began with Kelly's seminal paper [33] on an optimization framework for resource allocation in the Internet. The NUM literature is vast: many distributed algorithms [40, 52, 66], congestion control protocols [65, 47], and active queue management schemes [6] have been designed based on NUM. In addition, the optimization viewpoint provides a theoretical framework for analyzing existing congestion control protocols (such as TCP) as distributed algorithms for solving NUM for specific utility functions [39, 42, 41, 32, 36] (refer to [61] for a comprehensive survey).

Several distributed algorithms have been proposed in the theoretical literature [5, 10, 66] for NUM that aim to speed up convergence using Newton-method-based update steps. The calculations in these algorithms are much more involved than traditional gradient-based schemes (and NUMFabric), but they may result in faster convergence.

The NUM framework has also been generalized in various ways: utilities can be functions of rate, delay, jitter, reliability, etc. and can even be coupled across flows [51]. The assumption of continuity and concavity of the utility functions can be relaxed [22]. While we do not consider these extensions in this paper, we believe our new approach for solving NUM may apply more generally.

8. CONCLUSION

NUMFabric enables operators to flexibly optimize the network's bandwidth allocation in order to meet service level objectives. By decoupling network utilization from relative bandwidth allocation via a weighted max-min achieving transport layer based on WFQ, NUMFabric shows how in-network packet scheduling can help implement diverse policies at datacenter speeds and scales.

NUMFabric pushes the envelope on the convergence speed of practical mechanisms for achieving NUM at scale, but it is still not fast enough for the shortest of flows which last fewer than 5-10 RTTs (§6.1). In the future, NUMFabric's two layers (Figure 1) could evolve to improve convergence speed or simplify the implementation. For instance, an interesting line of future work could consider alternatives to Swift (§4.1) for achieving weighted max-min, for example, practical approximations of WFQ such as a small set of queues with different weights; or replacing WFQ altogether with a fast proactive rate control scheme such as PERC [27].

NUMFabric's theoretical analysis also warrants more attention. While we can show that NUMFabric's fixed point is unique and optimal [48], we have yet to prove that it always converges or analytically characterize any constraints on its parameters for convergence.

Finally, it is still an open question as to how an operator can take advantage of NUMFabric's capabilities to optimize application layer objectives. This is the focus of our current work. We are extending NUMFabric to support more general definitions of flows such as co-flows [15, 17], VM-level and tenant-level aggregates [7, 55, 26, 35].

Acknowledgements. We thank our shepherd, David Wetherall, and the anonymous SIGCOMM reviewers for their valuable feedback which helped improve the presentation of the paper. This work was partly supported by a Google Faculty Research Award.

9. REFERENCES

[1] M. Alizadeh et al. Data Center TCP (DCTCP). In *SIGCOMM*, 2010.
[2] M. Alizadeh et al. Less is more: trading a little bandwidth for ultra-low latency in the data center. In *NSDI*, 2012.
[3] M. Alizadeh et al. pFabric: Minimal Near-optimal Datacenter Transport. In *SIGCOMM*, 2013.
[4] M. Alizadeh et al. CONGA: Distributed congestion-aware load balancing for datacenters. In *SIGCOMM*, 2014.
[5] S. Athuraliya and S. Low. Optimization flow control with Newton-like algorithm. In *GLOBECOM*, 1999.
[6] S. Athuraliya, S. H. Low, V. H. Li, and Q. Yin. REM: active queue management. *IEEE Network*, 15(3):48–53, 2001.
[7] H. Ballani, P. Costa, T. Karagiannis, and A. Rowstron. Towards Predictable Datacenter Networks. In *SIGCOMM*, 2011.
[8] J. C. R. Bennett and H. Zhang. WF2Q: worst-case fair weighted fair queueing. In *INFOCOM*, 1996.
[9] D. P. Bertsekas, R. G. Gallager, and P. Humblet. *Data networks*, volume 2. Prentice-Hall International New Jersey, 1992.
[10] D. Bickson, Y. Tock, A. Zymnis, S. P. Boyd, and D. Dolev. Distributed large scale network utility maximization. In *ISIT*, 2009.
[11] P. Bosshart, G. Gibb, H.-S. Kim, G. Varghese, N. McKeown, M. Izzard, F. Mujica, and M. Horowitz. Forwarding Metamorphosis: Fast Programmable Match-action Processing in Hardware for SDN. In *SIGCOMM*, 2013.
[12] S. Boyd and L. Vandenberghe. *Convex optimization*. Cambridge university press, 2009.

[13] R. L. Carter and M. E. Crovella. Measuring Bottleneck Link Speed in Packet-switched Networks. *Perform. Eval.*, 27-28:297–318, Oct. 1996.

[14] M. Chowdhury and I. Stoica. Efficient coflow scheduling without prior knowledge. In *SIGCOMM*, 2015.

[15] M. Chowdhury, Y. Zhong, and I. Stoica. Efficient Coflow Scheduling with Varys. In *SIGCOMM*, 2014.

[16] A. Demers, S. Keshav, and S. Shenker. Analysis and Simulation of a Fair Queueing Algorithm. In *SIGCOMM*, 1989.

[17] F. R. Dogar, T. Karagiannis, H. Ballani, and A. Rowstron. Decentralized Task-aware Scheduling for Data Center Networks. *SIGCOMM*, 2014.

[18] N. Dukkipati, M. Kobayashi, R. Zhang-Shen, and N. McKeown. Processor sharing flows in the internet. In *IWQoS*, 2005.

[19] S. J. Golestani. A self-clocked fair queueing scheme for broadband applications. In *INFOCOM*, 1994.

[20] P. Goyal, H. M. Vin, and H. Cheng. Start-time fair queueing: a scheduling algorithm for integrated services packet switching networks. *IEEE/ACM Transactions on Networking (TON)*, 5(5):690–704, 1997.

[21] D. Han, R. Grandl, A. Akella, and S. Seshan. Fcp: a flexible transport framework for accommodating diversity. In *SIGCOMM*, 2013.

[22] P. Hande, Z. Shengyu, and M. Chiang. Distributed rate allocation for inelastic flows. *IEEE/ACM Transactions on Networking (TON)*, 15(6):1240–1253, 2007.

[23] C.-Y. Hong, M. Caesar, and P. B. Godfrey. Finishing Flows Quickly with Preemptive Scheduling. In *SIGCOMM*, 2012.

[24] S. Ishikawa. Fixed points by a new iteration method. *Proceedings of the American Mathematical Society*, 44(1):147–150, 1974.

[25] S. Jain et al. B4: Experience with a Globally-deployed Software Defined Wan. In *SIGCOMM*, 2013.

[26] V. Jeyakumar et al. EyeQ: Practical Network Performance Isolation at the Edge. In *NSDI*, 2013.

[27] L. Jose et al. High Speed Networks Need Proactive Congestion Control. In *HotNets*, 2015.

[28] D. Katabi, M. Handley, and C. Rohrs. Congestion control for high bandwidth-delay product networks. In *SIGCOMM*, 2002.

[29] F. Kelly. Charging and rate control for elastic traffic. *European transactions on Telecommunications*, 8(1):33–37, 1997.

[30] F. Kelly, G. Raina, and T. Voice. Stability and fairness of explicit congestion control with small buffers. *ACM SIGCOMM Computer Communication Review*, 2008.

[31] F. Kelly and T. Voice. Stability of end-to-end algorithms for joint routing and rate control. *SIGCOMM*, 2005.

[32] F. P. Kelly. Mathematical modeling of the internet. *Mathematics unlimited-2001 and beyond*, pages 685–702, 2001.

[33] F. P. Kelly, A. K. Maulloo, and D. K. H. Tan. Rate control for communication networks: Shadow prices, proportional fairness and stability. *The Journal of the Operational Research Society*, 49(3):pp. 237–252, 1998.

[34] S. Keshav. A Control-theoretic Approach to Flow Control. In *SIGCOMM*, 1991.

[35] A. Kumar et al. BwE: Flexible, Hierarchical Bandwidth Allocation for WAN Distributed Computing. In *SIGCOMM*, 2015.

[36] S. Kunniyur and R. Srikant. End-to-end congestion control schemes: Utility functions, random losses and ecn marks. *IEEE/ACM Transactions on Networking*, 11(5):689–702, 2003.

[37] K. Lee, J. C. Eidson, H. Weibel, and D. Mohl. Ieee 1588-standard for a precision clock synchronization protocol for networked measurement and control systems. In *Conference on IEEE*, volume 1588, page 2, 2005.

[38] L.-S. Liu. Ishikawa and mann iterative process with errors for nonlinear strongly accretive mappings in banach spaces. *Journal of Mathematical Analysis and Applications*, 194(1):114–125, 1995.

[39] S. H. Low. A duality model of tcp and queue management algorithms. *IEEE/ACM Transactions on Networking (TON)*, 11(4):525–536, 2003.

[40] S. H. Low and D. E. Lapsley. Optimization flow control i: basic algorithm and convergence. *IEEE/ACM Transactions on Networking (TON)*, 7(6):861–874, 1999.

[41] S. H. Low, F. Paganini, and J. C. Doyle. Internet congestion control. *Control Systems, IEEE*, 22(1):28–43, 2002.

[42] S. H. Low, L. L. Peterson, and L. Wang. Understanding tcp vegas: a duality model. *Journal of the ACM*, 49(2):207–235, 2002.

[43] D. J. MacKay. *Information theory, inference and learning algorithms*. Cambridge university press, 2003.

[44] L. Massoulié and J. Roberts. Bandwidth sharing: objectives and algorithms. In *INFOCOM*, 1999.

[45] R. Mittal, R. Agarwal, S. Ratnasamy, and S. Shenker. Universal packet scheduling. In *NSDI*, 2016.

[46] T. Mizrahi, E. Saat, and Y. Moses. Timed Consistent Network Updates. In *SIGCOMM*, 2015.

[47] J. Mo and J. Walrand. Fair end-to-end window-based congestion control. *IEEE/ACM Transactions on Networking (ToN)*, 8(5):556–567, 2000.

[48] K. Nagaraj, D. Bharadia, H. Mao, S. Chinchali, M. Alizadeh, and S. Katti. NUMFabric: Fast and Flexible Bandwidth Allocation in Datacenters. https://people.csail.mit.edu/alizadeh/papers/numfabric-techreport.pdf.

[49] ns3 Network Simulator. http://www.nsnam.org/.

[50] NUMFabric public release. https://knagaraj@bitbucket.org/knagaraj/numfabric.git.

[51] D. P. Palomar and M. Chiang. A tutorial on decomposition methods for network utility maximization. *IEEE Journal on Selected Areas in Communications*, 24(8):1439–1451, 2006.

[52] D. P. Palomar and M. Chiang. Alternative distributed algorithms for network utility maximization: Framework and applications. *IEEE Transactions on Automatic Control*, 52(12):2254–2269, 2007.

[53] A. K. Parekh and R. G. Gallager. A generalized processor sharing approach to flow control in integrated services networks: the single-node case. *IEEE/ACM Transactions on Networking (ToN)*, 1(3):344–357, 1993.

[54] J. Perry, A. Ousterhout, H. Balakrishnan, D. Shah, and H. Fugal. Fastpass: A Centralized "Zero-queue" Datacenter Network. In *SIGCOMM*, 2014.

[55] L. Popa et al. Faircloud: sharing the network in cloud computing. In *SIGCOMM*, 2012.

[56] C. Raiciu et al. Improving Datacenter Performance and Robustness with Multipath TCP. In *SIGCOMM*, 2011.

[57] J. Ros and W. K. Tsai. A theory of convergence order of maxmin rate allocation and an optimal protocol. In *INFOCOM*, 2001.

[58] A. Shieh, S. Kandula, A. G. Greenberg, C. Kim, and B. Saha. Sharing the data center network. In *NSDI*, 2011.

[59] A. Sivaraman et al. Towards Programmable Packet Scheduling. In *HotNets*, 2015.

[60] A. Sivaraman et al. Programmable Packet Scheduling at Line Rate. In *SIGCOMM*, 2016.

[61] R. Srikant. *The mathematics of Internet congestion control*. Springer, 2004.

[62] I. Stoica, S. Shenker, and H. Zhang. Core-stateless Fair Queueing: Achieving Approximately Fair Bandwidth Allocations in High Speed Networks. In *SIGCOMM*, 1998.

[63] K.-K. Tan and H. K. Xu. Approximating fixed points of nonexpansive mappings by the ishikawa iteration process. *Journal of Mathematical Analysis and Applications*, 178(2):301–308, 1993.

[64] B. Vamanan, J. Hasan, and T. N. Vijaykumar. Deadline-Aware Datacenter TCP (D2TCP). In *SIGCOMM*, 2012.

[65] D. X. Wei, C. Jin, S. H. Low, and S. Hegde. Fast tcp: motivation, architecture, algorithms, performance. *IEEE/ACM Transactions on Networking (ToN)*, 14(6):1246–1259, 2006.

[66] E. Wei, A. Ozdaglar, and A. Jadbabaie. A distributed newton method for network utility maximization. In *CDC*, 2010.

[67] C. Wilson, H. Ballani, T. Karagiannis, and A. Rowtron. Better never than late: meeting deadlines in datacenter networks. In *SIGCOMM*, 2011.

[68] D. Wischik, M. Handley, and M. B. Braun. The resource pooling principle. *SIGCOMM*, 2008.

[69] D. Wischik, C. Raiciu, A. Greenhalgh, and M. Handley. Design, Implementation and Evaluation of Congestion Control for Multipath TCP. In *NSDI*, 2011.