Computer Science and Artificial Intelligence Laboratory

Technical Report

MIT-CSAIL-TR-2016-011 August 15, 2016

Flowtune: Flowlet Control for Datacenter Networks

Jonathan Perry, Hari Balakrishnan, and Devavrat Shah

massachusetts institute of technology, cambridge, ma 02139 usa — www.csail.mit.edu

Flowtune: Flowlet Control for Datacenter Networks

Jonathan Perry, Hari Balakrishnan and Devavrat Shah
M.I.T. Computer Science and Artificial Intelligence Laboratory
{yonch,hari,devavrat}@mit.edu

CSAIL Tech Report, 2016-08-15

Abstract

Rapid convergence to a desired allocation of network resources to endpoint traffic has been a long-standing challenge for packet-switched networks. The reason for this is that congestion control decisions are distributed across the endpoints, which vary their offered load in response to changes in application demand and network feedback on a packet-by-packet basis. We propose a different approach for datacenter networks, *flowlet control*, in which congestion control decisions are made at the granularity of a flowlet, not a packet. With flowlet control, allocations have to change only when flowlets arrive or leave.

We have implemented this idea in a system called Flowtune using a centralized allocator that receives flowlet start and end notifications from endpoints. The allocator computes optimal rates using a new, fast method for network utility maximization, and updates endpoint congestion-control parameters. Experiments show that Flowtune outperforms DCTCP, pFabric, sfqCoDel, and XCP on tail packet delays in various settings, converging to optimal rates within a few packets rather than over several RTTs. Our implementation of Flowtune handles 10.4× more throughput per core and scales to 8× more cores than Fastpass, for an 83-fold throughput gain.

1 Introduction

Over the past thirty years, network congestion control schemes—whether distributed [25, 9, 22, 20, 40] or centralized [33], whether end-to-end or with switch support [13, 17, 18, 36, 27, 39, 32], and whether in the wide-area Internet [14, 43] or in low-latency datacenters [2, 3, 4, 23, 31]—have operated at the granularity of individual packets. Endpoints transmit data at a rate (window) that changes from packet to packet.

Packet-level network resource allocation has become the de facto standard approach to the problem of determining the rates of each flow in a network. By contrast, *if* it were possible to somehow determine optimal rates for a set of flows sharing a network, then those rates would have to change *only* when new flows arrive or flows leave the system. Avoiding packet-level rate fluctuations could help achieve *fast convergence* to optimal rates.

For this reason, in this paper, we adopt the position that a *flowlet*, and not a packet, is a better granularity for congestion control. By "flowlet", we mean a batch of packets that are backlogged at a sender; a flowlet ends when there is a threshold amount of time during which a sender's queue is empty. Our idea is to compute optimal rates for a set of active flowlets and to update those rates dynamically as flowlets enter and leave the network.[1]

We have developed these ideas in a system called *Flowtune*. It is targeted at datacenter environments, although it may also be used in enterprise and carrier networks, but is not intended for use in the wide-area Internet.

In datacenters, fast convergence of allocations is critical, as flowlets tend to be short (one study shows that the majority of flows are under 10 packets [11]) and link capacities are large (40 Gbits/s and increasing); if it takes more than, say, 40 μs to converge to the right rate, then most flowlets will have already finished. Most current approaches use distributed, end-to-end congestion control, and generally take multiple RTTs to converge. By contrast, Flowtune uses a centralized rate allocator, aiming for fast convergence to optimal rates for all flows in the network.

Computing the optimal rates is a difficult problem because even one flowlet arriving or leaving could, in general, cause updates to the rates of many existing flows, which in turn could cause updates to more flows, and so on in a cascading manner. To solve this problem in a scalable way, Flowtune uses the *network utility maximization* (NUM) framework, previously developed by Kelly et al. to analyze distributed congestion control protocols [28]. In Flowtune, the centralized allocator optimizes an objective like proportional fairness, i.e., $\max \sum_i U(x_i)$, where $U(x_i) = \log x_i$ (for example), and x_i is the throughput of

[1] long lived flows that send intermittently generate multiple flowlets.

flowlet *i*. We introduce a new method, termed *Newton-Exact-Diagonal* (NED), to perform this computation in a fast and scalable way in Flowtune's centralized allocator.

A scalable implementation of the optimization algorithm on CPUs would run in parallel on multiple cores. Unfortunately, straightforward implementations are slowed by expensive cache coherence traffic. We identify a partitioning of flows to cores where each core only interacts with a small set of links. Each core has copies of link state it needs. Before manipulating link state, the algorithm aggregates all modified copies of link state to authoritative copies. Afterwards, the algorithm distributes copies back to the cores. This scheme allows our implementation to allocate 15.36 Tbits/s in 8.29 µs (on 4 Nehalem cores, 40 Gbits/s links), up to 184 Tbits/s in 30.71 µs (64 Nehalem cores, 40 Gbits/s links).

Simulation results[2] show that Flowtune out-performs distributed congestion control methods like DCTCP, pFabric, Cubic-over-sfqCoDel, and XCP on metrics of interest like the convergence time and the 99%ile ("p99") of the flow completion time (FCT):

Versus	
DCTCP	8.6×-10.9× and 2.1×-2.9× lower p99 FCT for 1-packet and 1-10 packet flows. 12× lower p99 network queuing delay. Converges to a fair allocation within 20 µs vs. multiple milliseconds.
pFabric	Drop rate reduced from 6% to near-zero. 1.7×-2.4× lower p99 FCT on 1-packet and large flows. Shares network fairly without starving long flows.
sfqCoDel	Reduces p99 FCT by 3.5×-3.8× on 10-100 packet flows and 1.6×-1.85× for flows under 10 packets in high loads, and 2.1×-2.4× on 100-1000 packet flows on lower loads. Drop rate reduced from 8% to near-zero.
XCP	4000× faster convergence to correct allocation. 2.35× lower p99 FCT on 1-packet flows, 1.8×-4.1× on large flows, and 1.2×-3.2× on other flows. 3.5× lower p99 network queuing delay.

Compared with the centralized arbitration in Fastpass [33], Flowtune offers similar fast convergence, but is able to handle 10.4× traffic per core and utilize 8× more cores, for an improvement in throughput by a factor of 83. Another advantage over Fastpass is better fault tolerance because in Fastpass flows share fate with the arbiter: when the arbiter fails, the network has no idea who should transmit next. Fastpass must replicate the arbiter and implement a failover scheme to tolerate arbiter failures, increasing system complexity. By contrast, in Flowtune, if the allocator fails, traditional TCP congestion control takes over, and rate allocations remain close to optimal for a while. A network with Flowtune can recover from failure quickly even without any replication.

[2]we use simulation to compare network performance because pFabric, sfqCoDel, and XCP don't have usable implementations

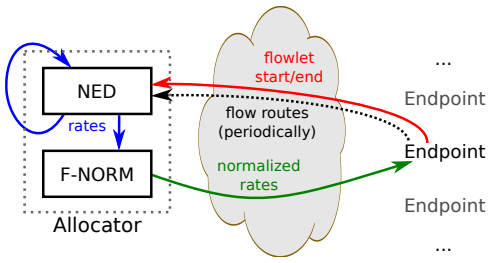


Figure 1: Flowtune components. Endpoints send notifications of new flowlets starting and current flowlets finishing to the allocator. The NED optimizer computes rates, which F-NORM then normalizes and sends back to Endpoints; endpoints adjust sending rates accordingly.

These results show that Flowtune achieves fast convergence of rates (within a few packets) to a desired network objective while avoiding network congestion, scaling favorably with link speeds, and without requiring any allocator replication for fault-tolerance.

2 Architecture

Endpoints report to the allocator when a flowlet starts or ends. On each such notification, Flowtune runs its optimizer, which computes a set of rates for each flow in the network. The optimization may cause some of these rates to temporarily exceed the capacity of some links, causing queuing delays. To reduce queuing, Flowtune uses a *rate normalizer* (F-NORM) to scale-down the computed values. The normalizer's results are sent to the endpoints. Endpoints transmit according to these rates (i.e., they are trusted, similar to trust in TCP transmissions today).

Flowtune does not select flow paths, but rather works given the paths the network selects for each flow (§7).

Figure 1 shows these components and how they interact with each other.

Optimizer intuition. Consider a network with a set of flows. Suppose a new flow starts or a flow ends. If we could set each flow's rate explicitly, what would we set it to?

The problem is that even one flow arriving or leaving can cause changes in the rates of an arbitrary subset of flows, depending on the topology and bottleneck structure. Certainly flows that share a bottleneck with the new or ending flow would change their rates. But if some of these flows slow down, the other flows elsewhere in the

network might be able to speed up, and so on. The effects can cascade.

To solve the problem of determining flow rates under flow churn, we turn to the *network utility maximization* (NUM) framework, first introduced by Kelly et al. [28, 30]. NUM offers three potential benefits. First, it allows network operators to specify an explicit objective and allocate rates that optimize that objective. Second, because previous work has shown that traditional congestion-control protocols often map to the NUM framework, NUM may be a reasonable approach for centralized allocation. Third, we show that it is possible to develop a fast, centralized method for rate allocation in this framework, which produces rates that outperform prior distributed schemes.

Our contribution here is a method to perform the optimization quickly; we term this method *Newton-Exact-Diagonal* (NED), because it is based on a "Newton-like" method [6] but takes advantage of the unique properties of the centralized context to speed up its execution.

Fault-tolerance. Flowtune has a more attractive fault-tolerance plan than Fastpass and centralized SDN controllers, both of which rely on replication and, in the case of SDN, maintaining state consistency. In Flowtune, the allocated rates have a temporary lifespan, and new allocated rates must arrive every few tens of microseconds. If the allocator fails, the rates expire and endpoint congestion control (e.g., TCP) takes over, using the previously allocated rates as a starting point. If the allocator only experiences a short failure, network rates will still be close to optimal when operation resumes.

Objective function. Flowtune uses a different objective function than Fastpass. The reason is that Fastpass is limited to methods that map to a weighted maximal matching to determine packet transmission times, such as max-min fairness or (approximately) minimum mean flow completion times. By contrast, Flowtune can achieve a variety of other desirable objectives such as weighted proportional fairness, which may be more appropriate for multi-bottleneck settings observed in datacenters. In this paper we focus on weighted proportional fairness, but note that the method supports any objective where flow utility is a function of the flow's allocated rate, and that different flows can have different utility functions.[3]

Rest of the paper. We focus on two key mechanisms of Flowtune: the NED flow optimizer (§3), and reducing queuing delays (§4). We also show how a parallel multicore implementation of the optimizer and normalizer can determine flow rates with low latency (§5), and present results on the performance of these methods and compare the results with other schemes (§6).

[3]Under some requirements of utility functions, discussed in §3.

3 Rate Allocation in Flowtune

This section presents Flowtune's rate allocation algorithm using the NUM framework. Solving an explicit optimization problem allows Flowtune to converge rapidly to the desired optimal solution. To our knowledge, it is the first NUM scheme designed specifically for fast convergence in the centralized setting.

The following table shows notation used in this section.

L	Set of all links	L(s)	Links traversed by flow s
S	Set of all flows	$S(\ell)$	Flows that traverse link ℓ
p_ℓ	Price of link ℓ	c_ℓ	Capacity of link ℓ
x_s	Rate of flow s	$U_s(x)$	Utility of flow s
G_ℓ	By how much link ℓ is over-allocated		
$H_{\ell\ell}$	How much flow rates on ℓ react to a change in p_ℓ		

The NUM framework. The goal is to allocate rates to all flows subject to network resource constraints: for each link $\ell \in L$,

$$\sum_{s \in S(\ell)} x_s \leq c_\ell. \tag{1}$$

Note that in general many allocations satisfy this constraint. The question is, which amongst these feasible options should be chosen. NUM proposes that it should be the one that maximizes the overall network utility, $\sum_{s \in S} U_s(x_s)$. Thus, the rate allocation should be the solution of the following optimization problem:

$$\begin{aligned} &\max \sum_s U_s(x_s) \qquad (2)\\ &\text{over } x_s \geq 0, \text{ for all } s \in S,\\ &\text{subject to (1).} \end{aligned}$$

Solving NUM using prices. The capacity constraints in (1) make it hard to solve the optimization problem directly. Kelly's approach to solving NUM [28] is to use Lagrange multipliers, which replace the hard capacity constraints with a "utility penalty" for exceeding capacities. This is done by introducing *prices* for links.

With prices, each flow selfishly optimizes its own profit, i.e., chooses a rate such that its utility, minus the price it pays per unit bandwidth on the links it traverses, is maximized. Although each flow is selfish, the system still converges to a global optimum because prices force flows to make globally responsible rate selections.[4]

The way prices are adjusted is the key differentiator between different algorithms to solve NUM. Simplistic methods can adjust prices too gently and be slow to converge, or adjust prices too aggressively and cause wild fluctuations in rates, or not even converge.

[4]We discuss the requirements for convergence further below.

Adjusting prices. An important quantity to consider when adjusting prices is by how much each link is over-allocated, i.e., $G_\ell = (\sum_{s \in S(\ell)} x_s) - c_\ell$. If $G_\ell > 0$, the link price should increase; if $G_\ell < 0$ it should decrease.

Gradient. Arguably the simplest algorithm for adjusting prices is Gradient projection [30], which adjusts prices directly from the amount of over-allocation:

$$p_\ell \leftarrow p_\ell + \gamma G_\ell.$$

Gradient's shortcoming is that it doesn't know how sensitive flows are to a price change, so it must update prices very gently (i.e., γ must be small). This is because depending on flow utility functions, large price updates might cause flows to react very strongly and change rates dramatically, causing oscillations in rates and failure to converge. This results in very timid price updates that make Gradient slow to converge.

Newton's method. Unlike the gradient method, Newton's method takes into account second-order effects of price updates. It adjusts the price on link ℓ based not only on how flows on ℓ will react, but also based on how price changes to all other links impact flows on ℓ:

$$\mathbf{p} \leftarrow \mathbf{p} - \gamma \mathbf{G}\, H^{-1},$$

where H is the Hessian matrix. This holistic price update makes Newton's method converge quickly, but also makes computing new prices expensive: inverting the Hessian on CPUs is impractical under Flowtune's time constraints.

The Newton-like method. An approximation to the Newton method was proposed in [6]. The Newton-like method estimates how sensitive flows are to price changes, by observing how price changes impact network throughput. Prices are then updated accordingly: inversely proportional to the estimate of price-sensitivity. The disadvantage is that network throughput must be averaged over relatively large time intervals, so estimating the diagonal is slow.

The NED algorithm. The key observation in NED that enables its fast convergence is that in the datacenter, it is possible to directly compute how flows on a link will react to a change in that link's price. In other words, NED computes the diagonal of the Hessian, $H_{\ell\ell}$ for all links. This eliminates the need to measure the network, and in contrast to the full Newton's method, can be computed quickly enough on CPUs for sizeable topologies. This results in the update rule:

$$p_\ell \leftarrow p_\ell + \gamma G_\ell H_{\ell\ell}^{-1}.$$

We note that the ability to directly compute $H_{\ell\ell}$ originates from trust that exists in the datacenter, not the centralization of the allocator.

Algorithm 1 Single iteration of Newton-Exact-Diagonal

NED updates rates $\mathbf{x} = (x_s)$ given prices $\mathbf{p} = (p_\ell)$ ("rate update" step). Then, in the next step of the iteration ("price update"), it uses the updated rates to update the prices.

Rate update. Given prices $\mathbf{p} = (p_\ell)$, for each flow $s \in S$, update the rate:

$$x_s = x_s(\mathbf{p}) = \left(U_s'\right)^{-1}\Big(\sum_{\ell \in L(s)} p_\ell\Big). \tag{3}$$

For example, if $U_s(x) = w \log x$, then $x_s = \frac{w}{\sum_{\ell \in L(s)} p_\ell}$.

Price update. Given updated rates $\mathbf{x} = \mathbf{x}(\mathbf{p}) = (x_s(\mathbf{p}))$ as described above, update the price of each link $\ell \in L$:

$$p_\ell \leftarrow \max\left(0, p_\ell - \gamma H_{\ell\ell}^{-1} G_\ell\right), \tag{4}$$

where $\gamma > 0$ is a fixed algorithm parameter (e.g. $\gamma = 1$), $G_\ell = (\sum_{s \in S(\ell)} x_s) - c_\ell$, $H_{\ell\ell} = \sum_{s \in S(\ell)} \frac{\partial x_s(\mathbf{p})}{\partial p_\ell}$.
From (3), $\frac{\partial x_s(\mathbf{p})}{\partial p_\ell} = \left((U_s')^{-1}\right)'\left(\sum_{m \in L(s)} p_m\right)$.

Algorithm 1 shows Flowtune's Newton-Exact-Diagonal (NED) rate allocation algorithm. In Flowtune, the initialization of prices happens only once, when the system first starts. The allocator starts without any flows, and link prices are all set to 1. When flows arrive, their initial rates are computed using current prices.

Choice of utility function. NED admits any utility function U_s that is strictly concave, differentiable, and monotonically increasing. For example, the logarithmic utility function, $U(x) = w \log x$ (for some weight $w > 0$), will optimize weighted proportional fairness [28].

Why price duality works. The utility function U_s for each $s \in S$ is a strictly concave function and hence the overall objective $\sum_s U_s$ in (2) is strictly concave. The constraints in (2) are linear. The capacity of each link is strictly positive and finite. Each flow passes through at least one link, i.e. $L(s) \neq \emptyset$ for each $s \in S$. Therefore, the set of feasible solutions for (2) is non-empty, bounded and convex. The Lagrangian of (2) is

$$L(\mathbf{x}, \mathbf{p}) = \sum_{s \in S} U_s(x_s) - \sum_{\ell \in L} p_\ell \Big(\sum_{s \in S(\ell)} x_s - c_\ell\Big). \tag{5}$$

with dual variables p_ℓ, and the dual function is defined as

$$D(\mathbf{p}) = \max\ L(\mathbf{x}, \mathbf{p}) \text{ over } x_s \geq 0, \text{ for all } s \in S. \tag{6}$$

The dual optimization problem is given by

$$\min\ D(\mathbf{p}) \text{ over } p_\ell \geq 0, \text{ for all } \ell \in L. \tag{7}$$

From Slater's condition in classical optimization theory, the utility of the solution of (2) is equal to its Lagrangian

dual's (7), and given the optimal solution $\mathbf{p}^\star$ of (7) it is possible to find the optimal solution for (2) from (6), i.e., using the rate update step. More details on solving NUM using Lagrange multipliers appear in [28, 6].

4 Rate normalization

The optimizer works in an online setting: when the set of flows changes, the optimizer does not start afresh, but rather updates the previous prices with the new flow configuration. While the prices re-converge, there are momentary spikes in throughput on some links.

Spikes occur because when one link price drops, flows on the link increase their rates and cause higher, over-allocated demand on other links (shown in §6.6).

Normally, allocating rates above link capacity results in queuing, and indeed REM [7], which implements a variant of gradient projection [30], added a mechanism for draining queues by raising prices when queue lengths increase.

By contrast, Flowtune's centralized optimizer can avoid queuing and its added latency by normalizing allocated rates to link capacities. We propose two schemes for normalization: *uniform normalization* and *flow normalization*.

For simplicity, the remainder of this section assumes all links are allocated non-zero throughput; it is straightforward to avoid division by zero in the general case.

4.1 Uniform normalization (U-NORM)

U-NORM scales the rates of all flows by a factor such that the most congested link will operate at its capacity. U-NORM first computes for each link the ratio of the link's allocation to its capacity $r_\ell = \sum_{s \in S(\ell)} x_s / c_\ell$. The most over-congested link has the ratio $r^\star = \max_{\ell \in L} r_\ell$; all flows are scaled using this ratio:

$$\bar{x}_s = \frac{x_s}{r^\star}. \tag{8}$$

The benefits of uniform scaling of all flows by the same constant are the scheme's simplicity, and that it preserves the relative sizes of flows; for utility functions of the form $w \log x_s$, this preserves the fairness of allocation.

However, as shown in §6.6, uniform scaling tends to scale down flows too much, reducing total network throughput.

4.2 Flow normalization (F-NORM)

Per-flow normalization scales each flow by the factor of its most congested link. This scales down all flows passing through a link ℓ by *at least* a factor of r_ℓ, which guarantees the rates through the link are at most the link capacity. Formally, F-NORM sets

$$\bar{x}_s = \frac{x_s}{\max_{\ell \in L(s)} r_\ell}. \tag{9}$$

F-NORM requires per-flow work to calculate normalization factors, and does not preserve relative flow rates, but a few over-allocated links do not hurt the entire network's throughput. Instead, only the flows traversing congested links are scaled down.

We note that the normalization of flow rates follows a similar structure to NED but instead of prices, the algorithm computes normalization factors. This allows F-NORM to reuse the multi-core designof NED, as described in §5.

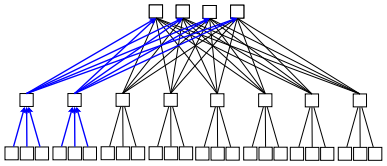
5 Implementation

Parallelizing the allocator. The allocator scales by working on multiple cores on one of more machines. Our design and implementation focuses on optimizing 2-stage Clos networks such as a Facebook fabric pod [5] or a Google Jupiter aggregation block [37], the latter consisting of 6,144 servers in 128 racks. We believe the techniques could be generalized to 3-stage topologies, but demonstrating that is outside the scope of this paper.

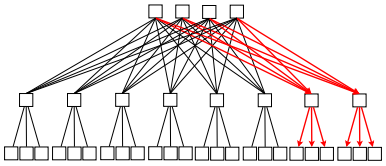
On a single multi-core machine. A strawman multiprocessor algorithm, which arbitrarily distributes flows to different processors, will result in poor performance because NED uses flow state to update link state when it computes aggregate link rates from flow rates: updates to a link from flows on different processors will cause significant cache-coherence traffic, slowing down the computation.

Now consider an algorithm that distributes flows to processors based on source rack. This algorithm is still likely to be sub-optimal: flows from many source racks can all update links to the same destination, again resulting in expensive coherence traffic. However, this grouping has the property that all updates to links connecting servers→ToR switches and ToR→aggregation switches (i.e., going *up* the topology) are only performed by the processor responsible for the source rack. A similar grouping by destination rack has locality in links going *down* the topology. Flowtune uses this observation for its multi-processor implementation.

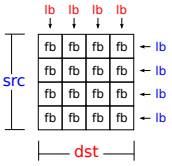
Figure 2 shows the partitioning of flows and links into FlowBlocks and LinkBlocks. Groups of network racks form *blocks* (two racks per block in the figure). All links going upwards from a block form an *upward LinkBlock*, and all links going downward towards a block form a *downward LinkBlock*. Flows are partitioned by both their source and destination blocks into *FlowBlocks*.



(a) Upward links from a set of racks form an *upward LinkBlock*. Only flows originating from these racks update this LinkBlock.

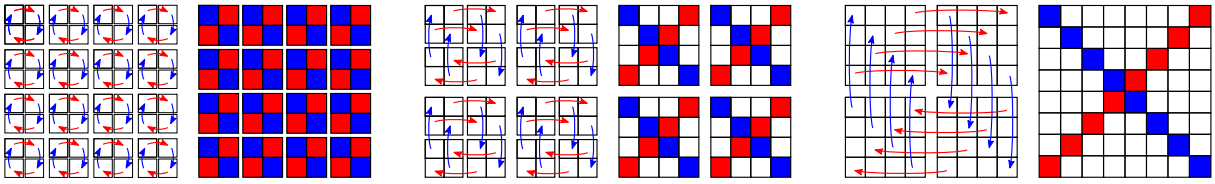


(b) Downward links towards a set of racks form a *downward LinkBlock*. Only flows destined to these racks update this LinkBlock.



(c) Flows are partitioned by source and destination into FlowBlocks, each updating an upward (blue) and a downward (red) LinkBlock.

Figure 2: Partitioning of network state.



(a) Step 1: 2x2 processors.

(b) Step 2: 4x4 processors.

(c) Step 3: 8x8 processors.

Figure 3: Aggregation of per-processor LinkBlock state in a 64-processor setup. At the end of step m, blocks of $2m \times 2m$ processors have aggregated upward LinkBlocks on the main diagonal, and downward LinkBlocks on the secondary diagonal.

This partitioning reduces concurrent updates, but does not eliminate them, as each upward LinkBlock is still updated by all FlowBlocks in the same source block. Similarly, downward LinkBlocks are updated by all FlowBlocks in the same destination block.

Aggregation. To eliminate concurrent updates completely, each FlowBlock works on private, local copies of its upward and downward LinkBlocks. The local copies are then *aggregated* into global copies. The algorithm then proceeds to update link prices on the global copies, and *distributes* the results back to FlowBlocks, so they again have local copies of the prices. Distribution follows the reverse of the aggregation pattern.

Figure 3 shows Flowtune's LinkBlock aggregation pattern. Each aggregation step m combines LinkBlocks within each $2m \times 2m$ group of processes to the group diagonals, with the main diagonal aggregating upward LinkBlocks, and the secondary diagonal downward LinkBlocks.

The aggregation scheme scales favorably with the number of cores. n^2 processors require only $\log_2 n$ steps rather than $\log_2 n^2$—the number of steps increases every quadrupling of processors, not doubling.

The aggregation pattern has uniform bandwidth requirements: when aggregating a $2m \times 2m$ group of processors, each $m \times m$ sub-group sends and receives the same number of LinkBlocks state to/from its neighbor sub-groups. Unlike FlowBlocks, whose size depends on the traffic pattern, each LinkBlock contains exactly the same number of links, making transfer latency more predictable.

Sending LinkBlocks is also much cheaper than sending FlowBlocks: datacenter measurements show average flow count per server at tens to hundreds of flows [19, 2], while LinkBlocks have a small constant of links per server (usually between one and three).

Multi-machine allocator. The LinkBlock–FlowBlock partitioning can be directly applied to distributing the allocator on multiple machines. Figure 3 is consistent with a setup with four machines with 16 cores each. In steps (a) and (b), each machine aggregates LinkBlocks internally, then in (c), aggregation is performed across machines; each machine receives from one machine and send to another. This arrangement scales to any $2^m \times 2^m$ collection of machines.

6 Evaluation

We evaluate Flowtune using benchmarks and simulation.

Benchmarks. The benchmarks measure the allocator's latency as a function of network size and the number of available cores, showing how the system scales.

Simulation. Simulations evaluate the system's overhead and convergence speed, and the emergent p99 FCT, packet drops, queueing delay, and fairness.

Why simulation? Simulation allows us to compare Flowtune to state-of-the art schemes whose implementations are only readily available in the ns2 simulator: pFabric [4], sfqCoDel [32], and XCP [27]. In addition, simulation

provides controlled, repeatable experiments, and large topologies – we simulated up to 2048 servers.

ns2 provides clean abstractions that force all communication to traverse the network, subject to queueing and packet drops. However, care must be taken to model important aspects of the network and of Flowtune to ensure that the results are trustworthy.

We took great care to accurately model Flowtune's control and data traffic in ns2. All control traffic shares the network with data traffic and experiences queuing and packet drops. The allocator adheres to ns2's separation of Agent vs. Application, ensuring all communication traverses the network. Control payloads are transmitted using TCP, and are only processed after all payload bytes arrive at the servers/allocator. The simulations use the same multicore implementation of NED described in §5 and benchmarked in §6.1.

The following table summarizes the experimental results.

Summary of Results	
§6.1	**(A)** A multi-core implementation optimizes traffic from 384 servers on 4 cores in 8.29 µs. 64 cores schedule 4608 servers' traffic in 30.71 µs – around 2 network RTTs.
§6.3	**(B)** Flowtune converges quickly to a fair allocation within 100 µs, orders of magnitude faster than other schemes.
§6.4	**(C)** The amount of traffic to and from the allocator depends on the workload; it is $< 0.17\%$, 0.57%, and 1.13% of network capacity for the Hadoop, cache, and web workloads. **(D)** Rate update traffic can be reduced by 69%, 64%, and 33% when allocating 0.95 of link capacities on the Hadoop, cache, and web workloads. **(E)** As the network size increases, allocator traffic takes the same fraction of network capacity.
§6.5	**(F)** Flowtune achieves low p99 flow completion time: 8.6×-10.9× and 1.7×-2.4× lower than DCTCP and pFabric on 1-packet flowlets, and 3.5×-3.8× than sfqCoDel on 10-100 packets. **(G)** Flowtune keeps p99 network queuing delay under 8.9 µs, 12× less than DCTCP. **(H)** Flowtune maintains a negligible rate of drops. sfqCoDel drops up to 8% of bytes, pFabric 6%.
§6.6	**(I)** Normalization is important; without it, NED over-allocates links by up to 140 Gbits/s. **(J)** F-NORM achieves over 99.7% of optimal throughput. U-NORM is not competitive.

6.1 Multicore implementation

We benchmarked NED's multi-core implementation on a machine with 8 Intel E7-8870 CPUs, each with 10 physical cores running at 2.4 GHz. We divided the network into 2, 4 and 8 blocks, giving runs with 4, 16, and 64 FlowBlocks. In the 4-core run, we mapped all FlowBlocks to the same CPU. With higher number of cores, we divided all FlowBlocks into groups of 2-by-2, and put two adjacent groups on each CPU.

The following table shows the number of cycles taken for different choices of network sizes and loads:

Cores	Nodes	Flows	Cycles	Time
4	384	3072	19896.6	8.29 µs
16	768	6144	21267.8	8.86 µs
64	1536	12288	30317.6	12.63 µs
64	1536	24576	33576.2	13.99 µs
64	1536	49152	40628.5	16.93 µs
64	3072	49152	57035.9	23.76 µs
64	4608	49152	73703.2	30.71 µs

Rows 1-3 show run-times with increasing number of cores, rows 3-5 with increasing number of flows, and rows 5-7 with increasing number of endpoints. These results show general-purpose CPUs are able to optimize network allocations on hundred of nodes within microseconds.

Rate allocation for 49K flows from 4608 endpoints takes 30.71 µs, around 2 network RTTs, or 3 RTTs considering an RTT for control messages to obtain the rate. TCP takes tens of RTTs to converge – significantly slower.

Communication between CPUs in the aggregate and distribute steps took more than half of the runtime in all experiments, e.g., 20 µs with 4068 nodes. This result implies it should be straightforward to perform the aggregate and distribute steps on multiple servers in a cluster using commodity hardware and kernel-bypass libraries.

Throughput scaling and comparison to Fastpass. Flowtune scales to larger networks than Fastpass, which reported 2.2 Tbits/s on 8 cores. Fastpass performs per-packet work, so its scalability declines with increases in link speed. Flowtune schedules flowlets, so allocated rates scale proportionally with the network links. The benchmarks results above show that on 40 Gbits/s links, 4 cores allocate 15.36 Tbits/s, and 64 cores allocate 184 Tbits/s on 64 cores in under 31 µs, 10.4× more throughput per core on 8× more cores – an 83× throughput increase over Fastpass.

6.2 Simulation setup

Topology. The topology is a two-tier full-bisection topology with 4 spine switches connected to 9 racks of 16 servers each, where server are connected with a 10 Gbits/s link. It is the same topology used in [4]. Links and servers have 1.5 and 2 microsecond delays respectively, for a total of 14 µs 2-hop RTT and 22 µs 4-hop RTT, commensurate with measurements we conducted in a large datacenter.

Workload. To model micro-bursts, flowlets follow a Poisson arrival process. Flowlet size distributions are according to the Web, Cache, and Hadoop workloads published by Facebook [34]. The Poisson rate at which flows enter the system is chosen to reach a specific average server load, where 100% load is when the rate equals server link capacity divided by the mean flow size. Unless otherwise specified, experiments use the Web workload, which has the highest rate of changes and hence stresses Flowtune the most among the three workloads. Sources and destinations are chosen uniformly at random.

Flowtune servers. When opening a new connection, servers start a regular TCP connection, and in parallel send a notification to the allocator. Whenever a server receives a rate update for a flow from the allocator, it opens the flow's TCP window and paces packets on that flow according to the allocated rate.

Flowtune allocator. The allocator performs an iteration every 10 µs. We found that for NED parameter γ in the range $[0.2, 1.5]$, the network exhibits similar performance; experiments have $\gamma = 0.4$.

Flowtune control connections. The allocator is connected using a 40 Gbits/s link to the each of the spine switches. Allocator–server communication uses TCP with a 20 µs minRTO and 30 µs maxRTO. Notifications of flowlet start, end, and rate updates are encoded in 16, 4, and 6 bytes plus the standard TCP/IP overheads. Updates to the allocator and servers are only applied when the corresponding bytes arrive, as in ns2's TcpApp.

6.3 Fast convergence

To show how fast the different schemes converge to a fair allocation, we ran five senders and one receiver. Starting with an empty network, every 10 ms one of the senders would start a flow to the receiver. Thereafter, every 10 ms one of the senders stops.

Figure 4 shows the rates of each of the flows as a function of time. Throughput is computed at 100 µs intervals; smaller intervals make very noisy results for most schemes. Flowtune achieves the ideal sharing between flows: N flows each get $1/N$ of bandwidth quickly (within 20 µs, not shown in the figure). DCTCP takes several milliseconds to approach the fair allocation, and even then traffic allocations fluctuate. pFabric doesn't share fairly; it prioritizes the flow with least remaining bytes and starves the other flows. sfqCoDel reaches a fair allocation quickly, but packet drops cause the application-observed throughput to be extremely bursty: the application sometime receives nothing for a while, then a large amount of data when holes in the window are successfully received. XCP is slow to allocate bandwidth, which results in low throughputs during most of the experiment.

6.4 Rate-update traffic

Flowtune only changes allocations on flowlet start and stop events, so when these events are relatively infrequent, the allocator could send relatively few updates every second. On the other hand, since the allocator optimizes utility across the entire network, a change to a single flow could potentially change the rates of all flows in the network. This section explores how much traffic is generated by traffic to and from the allocator.

The allocator notifies servers when the rates assigned to flows change by a factor larger than a threshold. For example, with a threshold of 0.01, a flow allocated 1 Gbit/s will only be notified when its rate changes above 1.01 or below 0.99 Gbits/s. To make sure links are not over-utilized, the allocator adjusts the available link capacities by the threshold; with a 0.01 threshold, the allocator would allocate 99% of link capacities.

Amount of update traffic. Figure 5 shows the amount of traffic sent to and from the allocator as a fraction of total network capacity, with a notification threshold of 0.01. The Web workload, which has the smallest mean flow size, also incurs the most update traffic: 1.13% of network capacity. At 0.8 load, the network will be 80% utilized, with 20% unused, so update traffic is well below the available headroom. Hadoop and Cache workloads need even less update traffic: 0.17% and 0.57%.

Traffic from servers to the allocator is substantially lower than from the allocator to servers: servers only communicate flowlet arrival and departures, while the allocator can potentially send many updates per flowlet.

Reducing update traffic. Increasing the update threshold reduces the volume of update traffic and the processing required at servers. Figure 6 shows the measured reduction in update traffic for different thresholds compared to the 0.01 threshold in Figure 5. Notifying servers of changes of 0.05 or more of previous allocations saves up to 69%, 64% and 33% of update traffic for the Hadoop, Cache, and Web workloads.

Effect of network size on update traffic. An addition or removal of a flow in one part of the network potentially changes allocations on the entire network. As the network grows, does update traffic also grow, or are updates contained? Figure 7 shows that as the network grows from 128 servers up to 2048 servers, update traffic takes the same fraction of network capacity — there is no debilitating cascading of updates that increases update traffic. This result shows that the threshold is effective at limiting the cascading of updates to the entire network.

6.5 Comparison to prior schemes

We compare Flowtune to DCTCP [2], pFabric [4], XCP [27], and Cubic+sfqCoDel [32].

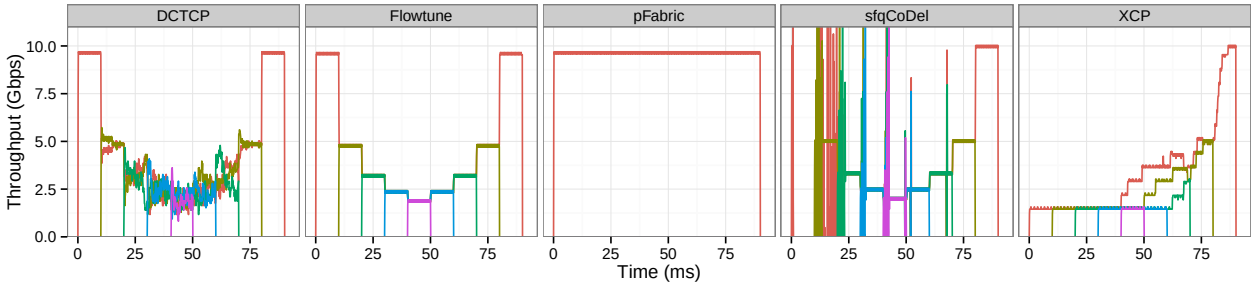


Figure 4: Flowtune achieves a fair allocation within 100 µs of a new flow arriving or leaving. In the benchmark, every 10 ms a new flow is added up to 5 flows, then flows finish one by one. DCTCP approaches a fair allocation after several milliseconds. pFabric, as designed, doesn't share the network among flows. sfqCoDel gets a fair allocation quickly, but retransmissions cause the application to observe bursty rates. XCP is conservative in handing out bandwidth and so converges slowly.

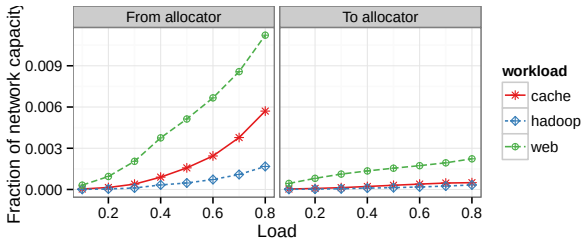


Figure 5: Overhead with Hadoop, cache, and Web workloads is $< 0.17\%$, 0.57%, and 1.13% of network capacity.

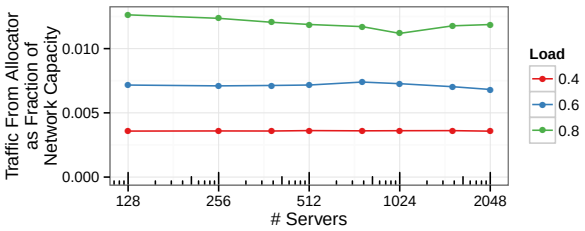


Figure 7: The fraction of rate-update traffic remains constant as the network grows from 128 to 2048 servers.

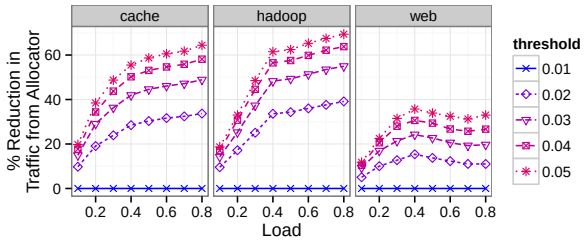


Figure 6: Notifying servers when rates change by more than a threshold substantially cuts control traffic volume.

99th percentile FCT. For datacenters to provide faster, more predictable service, tail latencies must be controlled. Further, when a user request must gather results from tens or hundreds of servers, p99 server latency quickly dominates user experience [12].

Figure 8 shows the improvement in 99th percentile flow completion time achieved by switching from different schemes to Flowtune. To summarize flows of different lengths to the different size ranges ("1-10 packets", etc.), we normalize each flow's completion time by the time it would take to send out and receive all its bytes on an empty network.

Flowtune preforms better than DCTCP on short flows: 8.6×-10.9× lower p99 FCT on 1-packet flows and 2.1×-2.9× on 1-10 packet flows. This happens because DCTCP has high p99 queuing delay, as shown in the next experiment.

Overall, pFabric and Flowtune have comparable performance, with Flowtune winning on some flow sizes, pFabric on others. Note, however, that Flowtune achieves this performance without requiring any changes to networking hardware. Flowtune achieves 1.7×-2.4× lower p99 FCT on 1-packet flows, and up to 2.4× on large flows. pFabric performs well on flows 1-100 packets long, with similar ratios. pFabric is designed to prioritize short flows, which explains its performance.

sfqCoDel has comparable performance on large flows, but is 3.5×-3.8× slower on 10-100 packets at high load and 2.1×-2.4× slower on 100-1000 packet flows at low load. This is due to sfqCoDel's high packet loss rate. Cubic handles most drops using SACKs, except at the end of the flow, where drops cause timeouts. These timeouts are most apparent in the medium-sized flows. We present packet-drop measurements below.

XCP is conservative in handing out bandwidth (§6.3), which causes flows to finish slowly.

Queuing delay. The following experiments collected queue lengths, drops, and throughput from each queue every 1 ms. Figure 9 shows the 99th percentile queuing delay on network paths, obtained by examining queue lengths. This queuing delay has a major contribution

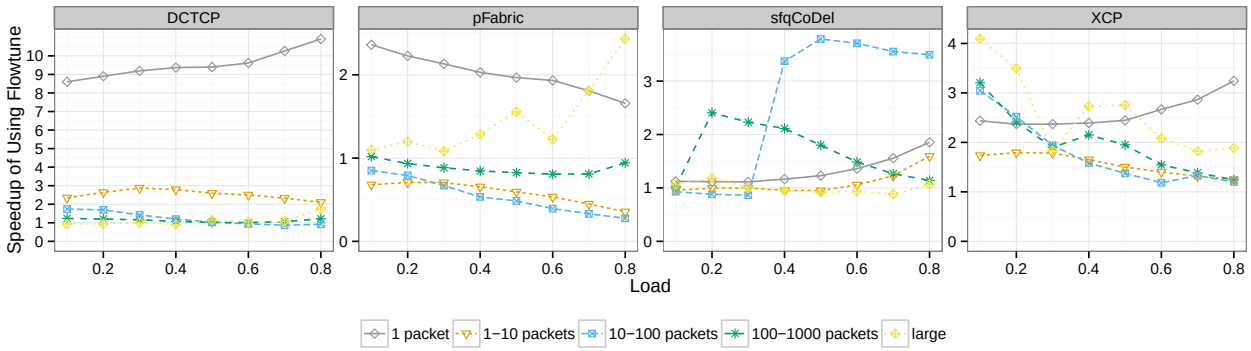


Figure 8: Improvement in 99th percentile flow completion time with Flowtune. Note the different scales of the y axis.

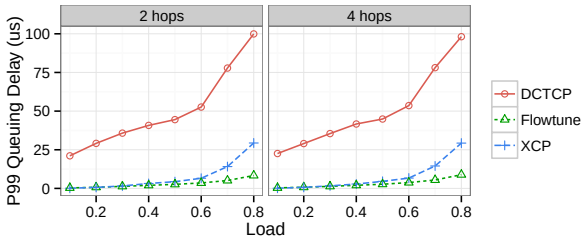


Figure 9: Flowtune keeps 2-hop and 4-hop 99th-percentile queuing delays below 8.9 µs. At 0.8 load, XCP has 3.5× longer queues, DCTCP 12×. pFabric and sfqCodel do not maintain FIFO ordering so their p99 queuing delay could not be inferred from sampled queue lengths.

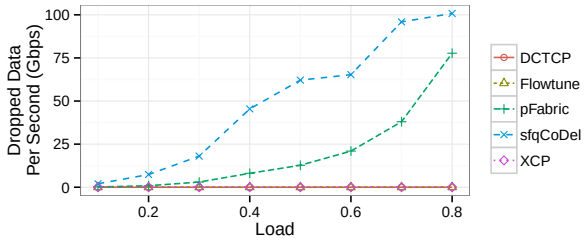


Figure 10: pFabric and sfqCoDel have a significant drop rate (1-in-13 for sfqCoDel). Flowtune, DCTCP, and XCP drop negligible amounts.

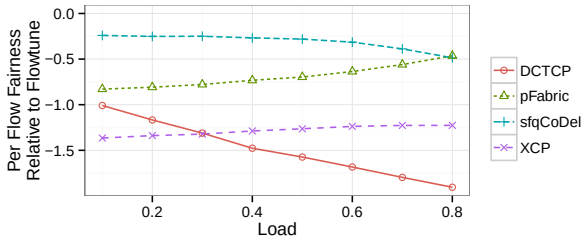


Figure 11: Comparison of proportional fairness of different schemes, i.e., $\sum \log_2(\text{rate})$. Flowtune allocates flows closer to their proportional-fair share.

to 1-packet and 1-10 packet flows. Flowtune has near-empty queues, whereas DCTCP's queues are 12× longer, contributing to the significant speedup shown in figure 8. XCP is conservative in handing out throughput, so its queues remain relatively shorter. pFabric and sfqCoDel maintain relatively long queues, but the comparison is not apples-to-apples since packets do not traverse their queues in FIFO order.

Packet drops. Figure 10 shows the rate at which the network drops data, in Gigabits per second. At 0.8 load, sfqCoDel servers transmit at 1279 Gbits/s (not shown), and the network drops over 100 Gbits/s, close to 8%. These drops in themselves are not harmful, however timeouts due to these drops could result in high p99 FCT, which appears to affect medium-sized flows (figure 8). Further, in a datacenter deployment of sfqCoDel, servers would spend many CPU cycles in slow-path retransmission code. pFabric's high drop rate would also make it prone to higher server CPU usage, but its probing and retransmission schemes mitigate high p99 FCT. Flowtune, DCTCP, and XCP drop negligible amounts.

Fairness. Figure 11 shows the proportional-fairness per-flow score of the different schemes normalized to Flowtune's score. A network where flows are assigned rates r_i gets score $\sum_i \log_2(r_i)$. This translates to gaining a point when a flow gets 2× higher rate, losing a point when a flow gets 2× lower rate. Flowtune has better fairness than the compared schemes: a flow's fairness score has on average 1.0-1.9 points more in Flowtune than DCTCP, 0.45-0.83 than pFabric, 1.3 than XCP, and 0.25 than CoDel.

6.6 Normalization

We now examine the performance of three algorithms: NED (§3), the Gradient method [30], and FGM (Fast Weighted Gradient Method [8]). For NED and Gradient, we compare the double-precision floating point reference implementations with real-time implementations NED-

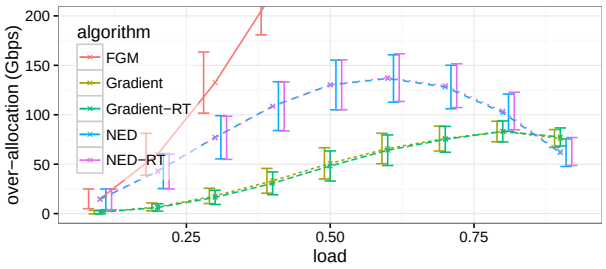


Figure 12: Normalization is necessary; without it, optimization algorithms allocate more than link capacities.

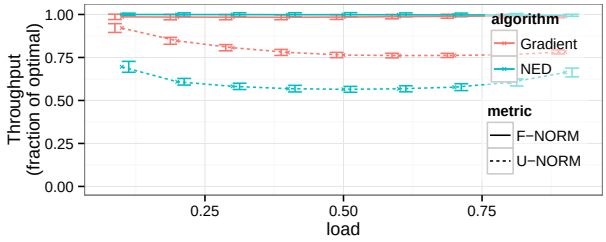


Figure 13: Normalizing with F-NORM achieves close to optimal throughput, while avoiding over-capacity allocations. U-NORM's throughput is low in comparison.

RT and Gradient-RT, which use single-point floating point operations and some numeric approximations for speed.

Need for normalization. As discussed in §4, without normalization, the churn of arriving and finishing flowlets causes momentary spikes in allocation, exceeding the capacity of some links. Left unhandled, this over-allocation causes queuing and drops in the network. Figure 12 shows the total amount of over-capacity allocations when there is no normalization. NED over-allocates more than Gradient because it is more aggressive at adjusting prices when flowlets arrive and leave. FGM does not handle the stream of updates well, and its allocations become unrealistic at even moderate loads.

U-NORM vs F-NORM. We ran Gradient and NED on the same workload and recorded their throughput in the steady state. After each iteration, we ran a separate instance of NED until it converged to the optimal allocation. Figure 13 shows U-NORM and F-NORM throughputs as a fraction of the optimal. F-NORM scales each flow based on the over-capacity allocations of links it traverses, achieving over 99.7% of optimal throughput with NED (98.4% with Gradient). In contrast, U-NORM scales flow throughput too aggressively, hurting overall performance. Gradient suffers less from U-NORM's scaling, because it adjusts rates slowly and does not over-allocate as much as NED.

Note that NED with F-NORM allocations occasionally slightly exceed the optimal allocation. This allocation does not exceed link capacities. Rather, the allocation gets more throughput than the optimal at the cost of being a little unfair to some flows.

7 Discussion

Path discovery: An important requirement that Flowtune makes of the underlying network is that the allocator know each flow's path through the network. Many common architectures support this requirement:

Routing information can be computed from the network state: in ECMP-based networks, given the ECMP hash function and switch failure notifications; in SDN-based networks, given controller decisions; and in MPLS-based [15] networks, given the MPLS configuration stream.

When endpoints choose routes, the allocator can gather routes from endpoints: in VL2 [19]-like networks where endpoints tunnel packets to a core switch for forwarding to the destination; and in static-routed network where endpoints have multiple subnets for different paths and the choice of subnet dictates a packet's path.

In other cases, endpoints could use a traceroute-like scheme to discover flow paths.

External traffic: Most datacenters do not run in isolation; they communicate with other datacenters and users on the Internet. A Flowtune cluster must be able to accept flows that are not scheduled by the allocator. Fastpass has a similar problem, and proposes to run external traffic at a lower priority so it doesn't interfere with scheduled traffic, or add gateways at the border of the network to schedule traffic up to the boundary. Flowtune could use either of these suggestions, but another possibility exists.

Both previous suggestions allow the allocator to work in an "open loop": the allocator can assume it knows everything about the state of the network. However, with NED, it is straightforward to dynamically adjust link capacities or add dummy flows for external traffic; a "closed loop" version of the allocator would gather network feedback observed by endpoints, and adjust its operation based on this feedback. The challenge here is what feedback to gather, and how to react to it in a way that provides some guarantees on the external traffic performance.

Scaling to larger networks: Although the allocator scales to multiple servers, the current implementation is limited to two-tier topologies. Beyond a few thousand endpoints, some networks add a third tier of spine switches to their topology that connects two-tier *pods*. Assigning a full pod to one block would create huge blocks, limiting allocator parallelism. On the other hand, the links going into and out of a pod are used by all servers in a pod, so splitting a pod to multiple blocks creates expensive updates. An open question is whether the Flow-

Block/LinkBlock abstraction can generalize to 3-tier clos networks, or if a new method is needed.

More scalable rate update schemes: Experiments in §6.4 show rate updates have a throughput overhead of 1.12%, so each allocator NIC can update 89 servers. In small deployments of a few hundred endpoints, it might be feasible to install a few NICs in the allocator to scale. Figure 5 shows how increasing the update threshold reduces update traffic, which can help scale a little farther, but as deployments grow to thousands of endpoints, even the reduced updates can overwhelm allocator NICs.

One relevant observation is that sending tiny rate updates of a few bytes has huge overhead: Ethernet has 64-byte minimum frames and preamble and interframe gaps, which cost 84-bytes, even if only one byte is sent. When sending an 8-byte rate update there is a 10× overhead. A straightforward solution to scale the allocator 10× would be to employ a group of intermediary servers that handle communication to a subset of individual endpoints. The allocator sends an MTU to each intermediary with all updates to the intermediary's endpoints. The intermediary would in turn forward rate updates to each endpoint, scaling up to a few thousand endpoints.

8 Related work

Rate allocation. Several systems control datacenter routes and rates, but are geared for inter-datacenter traffic. BwE [29] groups flows hierarchically and assigns a max-min fair allocation at each level of the hierarchy every 5-10 seconds on WAN links (similar time-scale to B4 [26]), and SWAN [24] receives demands from non-interactive services, computes rates, and reconfigures OpenFlow switches every 5 minutes. Flowtune supports a richer set of utility functions, with orders of magnitude smaller update times.

Hedera [1] gathers switch statistics to find elephant flows and reroutes those to avoid network hotspots. It is complementary to Flowtune: integrating the two systems can give Hedera its required information with very low latency. Mordia [16] and Datacenter TDMA [41] compute matchings between sources and destinations using gathered statistics, and at any given time, only flows of a single matching can send. While matchings are changed relatively frequently, the set of matchings is updated infrequently (seconds). In contrast, Flowtune updates allocations within tens of microseconds.

NED. The first-order methods [28, 30, 38] do not estimate $H_{\ell\ell}$ or use crude proxies. Gradient projection [30] adjusts prices with no weighting. Fast Weighted Gradient [8] uses a crude upper bound on the convexity of the utility function as a proxy for $H_{\ell\ell}$.

The Newton-like method [6], like NED, strives to use $H_{\ell\ell}$ to normalize price updates, but it uses network measurements to estimate its value. These measurements increase convergence time and have associated error; we have found the algorithm is unstable in several settings. Flowtune, in contrast, computes $H_{\ell\ell}$ explicitly from flow utilities, saving the time required to obtain estimates, and getting an error-free result.

Recent work [42] has a different formulation of the problem, with equality constraints rather than inequalities. While the scheme holds promise for faster convergence, iterations are much more involved and hence slower to compute, making the improvement questionable. Accelerated Dual Descent [44] does not use the flow model: it doesn't care what destination data arrives at, only that all data arrives at *some* destination. However, the method is notable for updating a link's price p_ℓ based not only on the link's current and desired throughput, but also on how price changes to other links p_k affect it. Adapting the method to the flow setting could reduce the number of required iterations to convergence (again at the cost of perhaps increasing iteration runtimes).

Parallel architectures. Conflict-free Replicated Data Types [35] (CRDTs) allow distributed data structure updates without synchronization, and then achieve eventual consistency through an arbitrary sequence of state merges. Flowtune's LinkBlock aggregation scheme allows distributed updates, but guarantees consistency after a fixed number of merges, and bounds communication throughput.

In the delegation parallel design pattern [10], all updates to a data structure are sent to a designated processor which then has exclusive access. Flowtune processors, however, perform the large bulk of updates to link state locally.

In flat-combining [21], concurrent users of a data structure write their requests in local buffers, and then the first user to obtain a global lock services requests of all waiting users. Flowtune's LinkBlock aggregation assigns responsibility for aggregation in a regular pattern, and does not incur the cost of competition between processors for global locks.

9 Conclusion

This paper made the case for *flowlet control* for datacenter networks. We developed Flowtune using this idea and demonstrated that it converges to an optimal allocation of rates within a few packet-times, rather than several RTTs. Our experiments show that Flowtune outperforms DCTCP, pFabric, Cubic-over-sfqCoDel, and XCP in various datacenter settings; for example, it achieves 8.6×-10.9× and 2.1×-2.9× lower p99 FCT for 1-packet and 1-10 packet flows compared to DCTCP.

Compared to Fastpass, Flowtune scales to 8× more cores and achieves 10.4× higher throughput per core,

does not require allocator replication for fault-tolerance, and achieves weighted proportional-fair rate allocations quickly in between 8.29 μs and 30.71 μs ($\leq$2 RTTs) for networks that have between 384 and 4608 nodes.

Acknowledgements

We thank Omar Baldonado, Chuck Thacker, Prabhakaran Ganesan, Songqiao Su, Petr Lapukhov, Neda Beheshti, James Zeng, Sandeep Hebbani and Chris Davies for helpful discussions. We are grateful for Facebook's support of Perry through a Facebook Fellowship. We thank the industrial members of the MIT Center for Wireless Networks and Mobile Computing for their support and encouragement.

References

[1] M. Al-Fares, S. Radhakrishnan, B. Raghavan, N. Huang, and A. Vahdat. Hedera: Dynamic Flow Scheduling for Data Center Networks. In *NSDI*, 2010.

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

[3] M. Alizadeh, A. Kabbani, T. Edsall, B. Prabhakar, A. Vahdat, and M. Yasuda. Less is More: Trading a Little Bandwidth for Ultra-Low Latency in the Data Center. In *NSDI*, 2012.

[4] M. Alizadeh, S. Yang, M. Sharif, S. Katti, N. McKeown, B. Prabhakar, and S. Shenker. pFabric: Minimal near-optimal datacenter transport. In *Proceedings of the ACM SIGCOMM 2013 conference on SIGCOMM*, pages 435–446. ACM, 2013.

[5] A. Andreyev. Introducing data center fabric, the next-generation Facebook data center network.

[6] S. Athuraliya and S. H. Low. Optimization Flow Control with Newton-like Algorithm. *Telecommunication Systems*, 15(3-4):345–358, 2000.

[7] S. Athuraliya, S. H. Low, V. H. Li, and Q. Yin. REM: Active Queue Management. *IEEE Network*, 15(3):48–53, 2001.

[8] A. Beck, A. Nedic, A. Ozdaglar, and M. Teboulle. A Gradient Method for Network Resource Allocation Problems. *IEEE Trans. on Control of Network Systems*, 1(1):64–73, 2014.

[9] L. S. Brakmo, S. W. O'Malley, and L. L. Peterson. TCP Vegas: New Techniques for Congestion Detection and Avoidance. In *Proceedings of the Conference on Communications Architectures, Protocols and Applications*, SIGCOMM '94, pages 24–35, New York, NY, USA, 1994. ACM.

[10] I. Calciu, J. E. Gottschlich, and M. Herlihy. Using elimination and delegation to implement a scalable numa-friendly stack. In *Proc. Usenix Workshop on Hot Topics in Parallelism (HotPar)*, 2013.

[11] Y. Chen, S. Alspaugh, and R. H. Katz. Design insights for mapreduce from diverse production workloads. In *Tech. Rep. EECS-2012-17*. UC Berkeley, 2012.

[12] J. Dean and L. A. Barroso. The tail at scale. *Comm. of the ACM*, 2013.

[13] A. Demers, S. Keshav, and S. Shenker. Analysis and Simulations of a Fair-Queueing Algorithm. *Internetworking: Research and Experience*, V(17):3–26, 1990.

[14] M. Dong, Q. Li, D. Zarchy, P. B. Godfrey, and M. Schapira. PCC: Re-architecting Congestion Control for Consistent High Performance. In *NSDI*, 2015.

[15] A. Elwalid, C. Jin, S. Low, and I. Widjaja. MATE: MPLS Adaptive Traffic Engineering. In *INFOCOM*, 2001.

[16] N. Farrington, G. Porter, Y. Fainman, G. Papen, and A. Vahdat. Hunting Mice with Microsecond Circuit Switches. In *HotNets*, 2012.

[17] S. Floyd. TCP and Explicit Congestion Notification. *CCR*, 24(5), Oct. 1994.

[18] S. Floyd and V. Jacobson. Random Early Detection Gateways for Congestion Avoidance. *IEEE ACM Trans. on Net.*, 1(4), Aug. 1993.

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

[20] S. Ha, I. Rhee, and L. Xu. CUBIC: A new TCP-friendly high-speed TCP variant. *ACM SIGOPS Operating Systems Review*, 42(5):64–74, 2008.

[21] D. Hendler, I. Incze, N. Shavit, and M. Tzafrir. Flat combining and the synchronization-parallelism tradeoff. In *Proceedings of the twenty-second annual ACM symposium on Parallelism in algorithms and architectures*, pages 355–364. ACM, 2010.

[22] J. C. Hoe. Improving the start-up behavior of a congestion control scheme for tcp. In *Conference Proceedings on Applications, Technologies, Architectures, and Protocols for Computer Communications*, SIGCOMM '96, pages 270–280, New York, NY, USA, 1996. ACM.

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

[24] C.-Y. Hong, S. Kandula, R. Mahajan, M. Zhang, V. Gill, M. Nanduri, and R. Wattenhofer. Achieving High Utilization with Software-Driven WAN. In *SIGCOMM*, 2013.

[25] V. Jacobson. Congestion Avoidance and Control. In *SIGCOMM*, 1988.

[26] S. Jain, A. Kumar, S. Mandal, J. Ong, L. Poutievski, A. Singh, S. Venkata, J. Wanderer, J. Zhou, M. Zhu, J. Zolla, U. Hölzle, S. Stuart, and A. Vahdat. B4: Experience with a Globally-deployed Software Defined Wan. In *Proceedings of the ACM SIGCOMM 2013 Conference on SIGCOMM*, SIGCOMM '13, pages 3–14, New York, NY, USA, 2013. ACM.

[27] D. Katabi, M. Handley, and C. Rohrs. Congestion Control for High Bandwidth-Delay Product Networks. In *SIGCOMM*, 2002.

[28] F. P. Kelly, A. K. Maulloo, and D. K. Tan. Rate Control for Communication Networks: Shadow prices, Proportional Fairness and Stability. *Journal of the Operational Research Society*, pages 237–252, 1998.

[29] A. Kumar, S. Jain, U. Naik, A. Raghuraman, N. Kasinadhuni, E. C. Zermeno, C. S. Gunn, J. Ai, B. Carlin, M. Amarandei-Stavila, M. Robin, A. Siganporia, S. Stuart, and A. Vahdat. Bwe: Flexible, hierarchical bandwidth allocation for wan distributed computing. In *Proceedings of the 2015 ACM Conference on Special Interest Group on Data Communication*, SIGCOMM '15, pages 1–14, New York, NY, USA, 2015. ACM.

[30] S. H. Low and D. E. Lapsley. Optimization Flow ControlĂŤ-I: Basic Algorithm and Convergence. *IEEE/ACM Trans. on Networking*, 7(6):861–874, 1999.

[31] R. Mittal, N. Dukkipati, E. Blem, H. Wassel, M. Ghobadi, A. Vahdat, Y. Wang, D. Wetherall, D. Zats, et al. TIMELY: RTT-based congestion control for the datacenter. In *SIGCOMM*, 2015.

[32] K. Nichols and V. Jacobson. Controlling queue delay. *Communications of the ACM*, 55(7):42–50, 2012.

[33] J. Perry, A. Ousterhout, H. Balakrishnan, D. Shah, and H. Fugal. Fastpass: A centralized "zero-queue" datacenter network. In *SIGCOMM*, 2014.

[34] A. Roy, H. Zeng, J. Bagga, G. Porter, and A. C. Snoeren. Inside the social network's (datacenter) network. In *Proceedings of the 2015 ACM Conference on Special Interest Group on Data Communication*, SIGCOMM '15, pages 123–137, New York, NY, USA, 2015. ACM.

[35] M. Shapiro, N. Preguiça, C. Baquero, and M. Zawirski. Conflict-free replicated data types. In *Proceedings of the 13th International Conference on Stabilization, Safety, and Security of Distributed Systems*, SSS'11, pages 386–400, Berlin, Heidelberg, 2011. Springer-Verlag.

[36] M. Shreedhar and G. Varghese. Efficient Fair Queueing Using Deficit Round Robin. In *SIGCOMM*, 1995.

[37] A. Singh, J. Ong, A. Agarwal, G. Anderson, A. Armistead, R. Bannon, S. Boving, G. Desai, B. Felderman, P. Germano, A. Kanagala, J. Provost, J. Simmons, E. Tanda, J. Wanderer, U. Hölzle, S. Stuart, and A. Vahdat. Jupiter rising: A decade of clos topologies and centralized control in google's datacenter network. In *Proceedings of the 2015 ACM Conference on Special Interest Group on Data Communication*, SIGCOMM '15, pages 183–197, New York, NY, USA, 2015. ACM.

[38] R. Srikant. *The mathematics of Internet congestion control*. Springer Science & Business Media, 2012.

[39] C. Tai, J. Zhu, and N. Dukkipati. Making Large Scale Deployment of RCP Practical for Real Networks. In *INFOCOM*, 2008.

[40] K. Tan, J. Song, Q. Zhang, and M. Sridharan. A compound TCP approach for high-speed and long distance networks. In *INFOCOM*, 2006.

[41] B. C. Vattikonda, G. Porter, A. Vahdat, and A. C. Snoeren. Practical TDMA for Datacenter Ethernet. In *EuroSys*, 2012.

[42] E. Wei, A. Ozdaglar, and A. Jadbabaie. A Distributed Newton Method for Network Utility Maximization–I: Algorithm. *IEEE Trans. on Automatic Control*, 58(9):2162–2175, 2013.

[43] K. Winstein, A. Sivaraman, and H. Balakrishnan. Stochastic Forecasts Achieve High Throughput and Low Delay over Cellular Networks. In *NSDI*, 2013.

[44] M. Zargham, A. Ribeiro, A. Ozdaglar, and A. Jadbabaie. Accelerated dual descent for network flow optimization. *IEEE Trans. on Automatic Control*, 59(4):905–920, 2014.