Flowtune: Flowlet Control for Datacenter Networks

Jonathan Perry, Hari Balakrishnan, and Devavrat Shah,
Massachusetts Institute of Technology

https://www.usenix.org/conference/nsdi17/technical-sessions/presentation/perry

This paper is included in the Proceedings of the 14th USENIX Symposium on Networked Systems Design and Implementation (NSDI '17).

March 27–29, 2017 • Boston, MA, USA

ISBN 978-1-931971-37-9

Open access to the Proceedings of the 14th USENIX Symposium on Networked Systems Design and Implementation is sponsored by USENIX.

Flowtune: Flowlet Control for Datacenter Networks

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

Abstract

Rapid convergence to a desired allocation of network resources to endpoint traffic is a difficult problem. The reason 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. Benchmarks on an EC2 deployment show a fairer rate allocation than Linux's Cubic. A data aggregation benchmark shows 1.61× lower p95 coflow completion time.

1 Introduction

Over the past thirty years, network congestion control schemes—whether distributed [24, 8, 21, 20, 39] or centralized [33], whether end-to-end or with switch support [12, 16, 17, 35, 26, 38, 32], and whether in the wide-area Internet [13, 42] or in low-latency datacenters [2, 3, 4, 22, 30]—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.

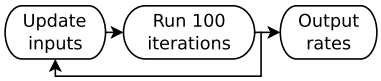
In datacenters, fast convergence of allocations is critical, as flowlets tend to be short (one study shows that the majority of flows are under 10 packets [9]) and link capacities are large (40 Gbits/s and increasing). If it takes longer than, say, 40 μs to converge to the right rate, then most flowlets will have already finished. Most current approaches use distributed congestion control, and generally take multiple RTTs to converge. By contrast, Flowtune uses a centralized rate allocator.

Computing the optimal rates is difficult because even one flowlet arriving or leaving could, in general, cause updates to the rates of many existing flows. Flows that share a bottleneck with the new or ending flow would change their rates. But, in addition, if some of these flows slow down, other flows elsewhere in the network might be able to speed up, and so on. The effects can cascade.

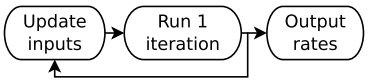
To solve this problem in a scalable way, Flowtune uses the *network utility maximization* (NUM) framework, previously developed to analyze distributed congestion control protocols [27, 29]. In Flowtune, network operators specify an explicit objective. We introduce a new method, termed *Newton-Exact-Diagonal* (NED), that converges quickly to the allocation that maximizes the specified utility (§2).

Flowtune can achieve a variety of desirable objectives. In this paper, we focus on proportional fairness, i.e.,

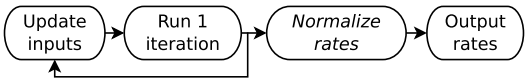
[1] Long-lived flows that send intermittently generate multiple flowlets.



(a) The strawman solution runs the optimizer until it converges, but then system is slow to react to updates of flowlet start/end.



(b) Performing just one iteration reduces reaction time, but results in over-allocation while the algorithm converges.



(c) Flowtune adds a normalization step that eliminates link over-allocation while achieving 99.7% of optimal throughput.

Figure 1: Motivation for the allocator architecture.

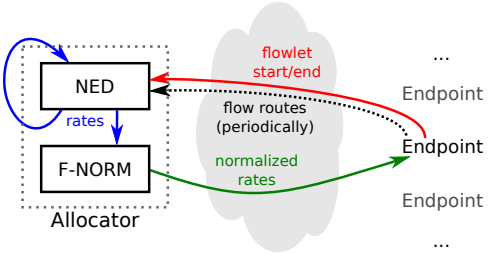


Figure 2: Flowtune components. Endpoints send notifications of new flowlets starting and current flowlets finishing to the allocator. The NED optimizer computes rates, which F-NORM then normalizes and sends back to Endpoints; endpoints adjust sending rates accordingly.

$\max \sum_i U(x_i)$, where $U(x_i) = \log x_i$, and x_i is the throughput of flowlet i. In general, however, NED supports any objective where the utility is a function of the flow's allocated rate.[2]

Endpoints report to the allocator whenever a flowlet starts or ends. Rather than waiting for the optimization to converge before setting rates (Figure 1a), the allocator continuously updates its set of active flowlets and sets intermediate rates between iterations of the optimizer (Figure 1b). These intermediate rates may temporarily exceed the capacity of some links, causing queuing delays. To reduce queuing, Flowtune uses a *rate normalizer* (F-NORM, §3) to scale-down the computed values (Figure 1c).

The normalizer's results are sent to the endpoints. Endpoints transmit according to these rates (they are trusted, similar to trust in TCP transmissions today). Figure 2 shows the system components and their interactions.

[2]Under some requirements of utility functions, discussed in §2.

Flowtune does not select flow paths, but rather works given the paths the network selects for each flow (§6).

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 propose 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 (§4). On 40 Gbits/s links, this scheme allows our implementation to allocate 15.36 Tbit/s in 8.29 on 4 Nehalem cores, up to 184 Tbit/s in 30.71 µs on 64 Nehalem cores (§5.2).

We implemented Flowtune in a Linux kernel module and a C++ allocator that implements the multi-core NED algorithm and uses kernel-bypass for NIC access. The system enforces rate allocations on unmodified Linux applications. We deployed Flowtune on Amazon Web Services instances; experiments show the servers are able to achieve their fair share of available network resources, with much better fairness than the Linux baseline (which uses Cubic). Flowtune reduced the 95th percentile (p95) of coflow completion times [10] by 1.61× on a data aggregation benchmark.

Simulation results 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 p99 of the flow completion time (FCT).

Compared with the centralized arbitration in Fastpass [33], Flowtune offers similar fast convergence, but handles 10.4× traffic per core and utilizes 8× more cores, for an improvement in throughput by a factor of 83.2.

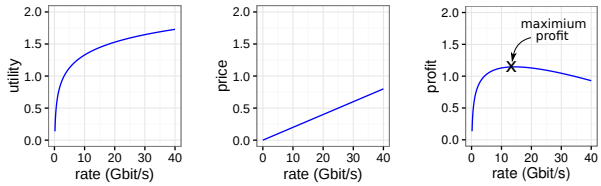
2 Rate Allocation in Flowtune

Solving an explicit optimization problem allows Flowtune to converge rapidly to the desired optimal solution. To our knowledge, NED is the first NUM scheme designed specifically for fast convergence in the centralized setting.

2.1 Intuition

The Flowtune rate allocator chooses flow rates by introducing link prices. The allocator adjusts link prices based on demand, increasing the price when the demand is high and decreasing it when demand is low.

Figure 3 shows how the allocator chooses a flow's rate given link prices. It takes the flow's utility function (3a). Then, it determines the flow's price per unit bandwidth, which is the sum of prices on links it traverses (3b). The utility minus price determines the profit (3c); the allocator chooses the rate that maximizes the flow's profit.



(a) Each flow has a utility function.

(b) Link prices determine the cost per unit rate.

(c) The rate maximizes profit, i.e., utility minus price.

Figure 3: Illustration of how NUM chooses a flow rate.

Intuitively, prices should be adjusted strongly when demand is far from capacity, and gently when it is close. But by exactly how much should an algorithm adjust prices in a given setting?

The exact recipe for adjusting prices is the key differentiator among 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.

NED converges quickly because it adjusts prices not only using the difference between demand and capacity, but also using an estimate of how strongly rates will change for a given change in price.

2.2 The NUM framework

The following table summarizes the notation used in this paper:

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 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. Among these, NUM proposes that we should choose the one that maximizes the overall network utility, $\sum_{s \in S} U_s(x_s)$. Thus, the rate allocation should be the solution to the following optimization problem:

$$\max \sum_s U_s(x_s) \tag{2}$$

$$\text{over } x_s \geq 0, \text{ for all } s \in S,$$

$$\text{subject to (1).}$$

Solving NUM using prices. The capacity constraints in (1) make it hard to solve the optimization problem directly. Kelly's approach to solving NUM [27] 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.[3]

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.

Appendix A outlines why price duality works. Related NUM algorithms are discussed in §7 and Appendix B.

2.3 The NED algorithm

The key observation in NED that enables its fast convergence is that given the utility functions, it is possible to directly compute (1) flow rates given prices, and (2) how strongly flows on a link ℓ will react to a change in that link's price, which we denote $H_{\ell\ell}$.[4]

Direct computation of values eliminates the need to measure the network, and thus greatly speeds up algorithm iterations. In contrast to the full Newton's method, prices updates based on the diagonal 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 the ability to reliably obtain the above values, not the centralization of the allocator. When the endpoints are trusted, a distributed implementation of NED can use the endpoints to compute and supply these values.

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 [27].

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

[4] H is in fact the Hessian; NED computes the Hessian's diagonal, $H_{\ell\ell}$. The Hessian's diagnoal is where NED gets its name.

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 $\ell \in L$:

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

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), $\frac{\partial x_s(\mathbf{p})}{\partial p_\ell} = \left((U_s')^{-1}\right)'\left(\sum_{m \in L(s)} p_m\right)$.

3 Rate normalization

The optimizer works in an online setting: when the set of flows changes, the optimizer does not start afresh, but instead 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 §3).

Normally, allocating rates above link capacity results in queuing. The 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.

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:

$$\bar{x}_s = \frac{x_s}{r^\star}. \tag{5}$$

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 §3, uniform scaling tends to scale down flows too much, reducing total network throughput.

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

$$\bar{x}_s = \frac{x_s}{\max_{\ell \in L(s)} r_\ell}. \tag{6}$$

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.

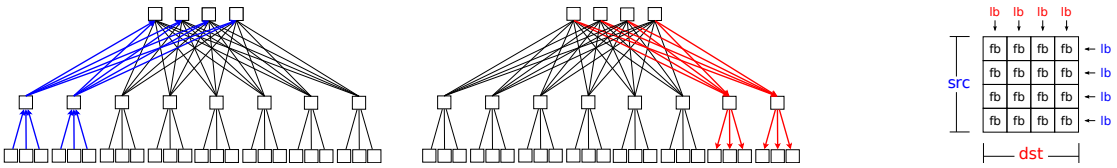
We note that the normalization of flow rates follows a similar structure to NED but instead of prices, the algorithm computes normalization factors. This allows F-NORM to reuse the multi-core design of NED, as described in §4.

4 Scalability

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 [36], 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.

A strawman multiprocessor algorithm, which arbitrarily distributes flows to different processors, will perform poorly 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.

Reducing concurrent updates. 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.

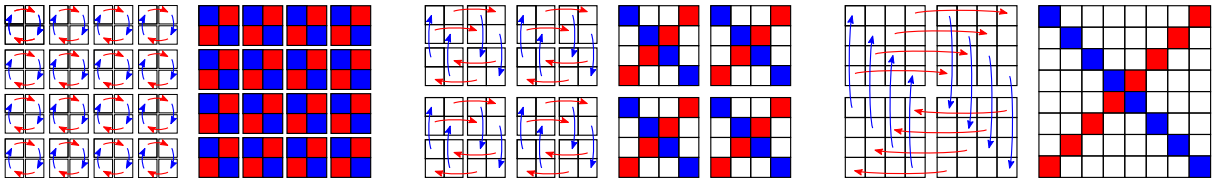


(a) Upward links from a set of racks form an *upward LinkBlock*. Only flows originating from these racks update this LinkBlock.

(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 4: Partitioning of network state.



(a) Step 1: 2x2 processors.

(b) Step 2: 4x4 processors.

(c) Step 3: 8x8 processors.

Figure 5: 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.

Figure 4 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*. 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.

Eliminating concurrent updates. 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 5 shows the 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 well 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 only every quadrupling of processors.

The aggregation pattern has uniform bandwidth requirements: when aggregating $2m \times 2m$ processors, each $m \times m$ sub-group sends and receives the same amount 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 [18, 2], while LinkBlocks have a small constant number of links per server (usually between one and three).

Multi-machine allocator. The LinkBlock-FlowBlock partitioning distributes the allocator on multiple machines. Figure 5 shows 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 sends to another. This arrangement scales to any $2^m \times 2^m$ collection of machines.

5 Evaluation

We evaluate Flowtune using a cluster deployment, microbenchmarks, and ns-2 and numeric simulations. ns-2 simulations allows comparison with state-of-the art schemes whose implementations are only readily avail-

able in the ns-2 simulator: pFabric [4], sfqCoDel [32], and XCP [26].

EC2 Experiments (§5.1)
(A) On Amazon EC2, Flowtune's sharing of available throughput is more fair than the baseline Linux implementation running Cubic.
(B) Flowtune makes transfers on EC2 more predictable: Many-to-One Coflow Completion Time was sped up by 1.61× in p95 and 1.24× in p90.

Multicore micro-benchmarks (§5.2)
(C) 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.

ns-2 Simulations (§5.3)
(D) Flowtune converges quickly to a fair allocation within 100 µs, orders of magnitude faster than other schemes.
(E) 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.
(F) 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.
(G) As the network size increases, allocator traffic takes the same fraction of network capacity.
(H) 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.
(I) Flowtune keeps p99 network queuing delay under 8.9 µs, 12× less than DCTCP.
(J) Flowtune maintains a negligible rate of drops. sfqCoDel drops up to 8% of bytes, pFabric 6%.
(K) Flowtune achieves higher proportional-fairness score than DCTCP, pFabric, sfqCoDel, and XCP.

Numeric simulation (§5.4)
(L) Normalization is important; without it, NED over-allocates links by up to 140 Gbits/s.
(M) F-NORM achieves over 99.7% of optimal throughput. U-NORM is not competitive.

5.1 Amazon EC2 deployment

We deployed Flowtune on 10 Amazon EC2 `c4.8xlarge` instances running Ubuntu 16.04 with 4.4.0 Linux kernels. One of the instances ran the allocator and had direct access to the NIC queues using SR-IOV. The other instances ran the workload.

Server module. We implemented the Flowtune client side using a kernel module, requiring no modification to applications. The module reports to the allocator when socket buffers transition between empty and non-empty, and enforces allocated rates by delaying packets when the rate limit is exceeded. An implementation could also change TCP slow-start and loss/marking behavior, but our implementation keeps those unchanged.

Protocol. Communication uses the Flowtune protocol over a variant of the Fastpass Control Protocol (FCP) for transport. The Flowtune protocol allows endpoints to process payloads without head-of-line blocking, so a dropped packet does not increase latency for non-dropped packets. The Flowtune protocol synchronizes state between the allocator and endpoints; when reacting to loss, instead of retransmitting old state, participants send the most recent state, and that only if the acknowledged state differs.

Allocator. The allocator is written in C++ and accesses NIC queues directly using the DPDK library. A hash table maps endpoints to their flow state, which the protocol maintains in synchronization with the endpoints. When allocated flow rates differ from the allocations acknowledged by the endpoints, the allocator triggers rate update messages.

Measurement. The experiment harness achieves accurate workload timing by measuring the clock offset of each instance using ntpdate. Before starting/stopping the workload, processes on the measured instances call nanosleep with appropriate amounts to compensate.

(A) Fairness. In an 8-to-1 experiment, eight senders start every 50 ms in sequence, and then finish similarly. Figure 6 shows the rates of each flow as the experiment progresses. Flowtune shares the throughput much more fairly than the baseline: the rates of the different flows overlap at equal division of throughput. The baseline rates oscillate, even with only 3 competing flows.

(B) Coflow completion time. Here, 8 senders each make 25 parallel transfers of 10 Mbytes to a single receiver. This transfer pattern models Spark aggregating data from worker cores, or a slice of a larger MapReduce shuffle stage. Figure 7 shows results from 100 runs with Flowtune vs. the baseline. Flowtune achieves more predictable results. The reduction in different percentiles are summarized in the following table.

Metric	Baseline	Flowtune	Speedup
median	1.859249	1.787622	1.04×
p90	2.341086	1.881433	1.24×
p95	3.050718	1.894544	1.61×

5.2 Multicore micro-benchmarks

We benchmarked NED's multi-core implementation on a machine with 8 Intel E7-8870 CPUs, each with 10 physi-

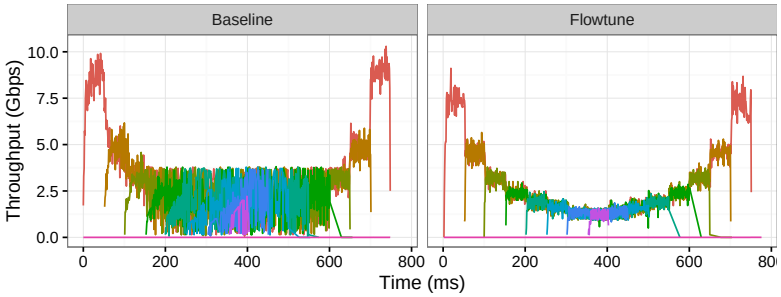


Figure 6: 8-to-1 experiment on Amazon EC2. Flowtune shares available throughput more fairly than baseline Linux.

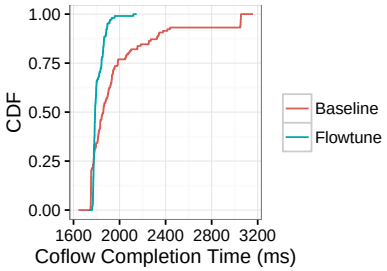


Figure 7: AWS EC2 data aggregation benchmarks. Flowtune coflows are more predictable and generally faster.

cal 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.

(C) Iteration time. This micro-benchmark measures the average NED iteration time, i.e., the "Run 1 iteration" in Figure 1c. 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, for example > 20 µs with 4068 nodes. This 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.

Note that this benchmark only captures the computation required to optimize flows; communication between the servers and the allocator is evaluated in §5.3.2, and discussed in §6.

Throughput scaling and comparison to Fastpass. Flowtune scales to larger networks than Fastpass, which reported 2.2 Tbit/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 benchmark results above show that on 40 Gbits/s links, 4 cores allocate 15.36 Tbit/s, and 64 cores allocate 184 Tbit/s on 64 cores in under 31 µs, 10.4× more throughput per core on 8× more cores – an 83× throughput increase over Fastpass.

5.3 ns-2 simulations

Model. All control traffic shares the network with data traffic and experiences queuing and packet drops. Control payloads are transmitted using TCP, and are only processed after all payload bytes arrive at their destinations.

Topology. The topology is a two-tier full-bisection topology with 4 spine switches connected to 9 racks of 16 servers each, where servers 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]. Appendix C has more information on the CDFs used. 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.

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 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.

5.3.1 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.

(D) Convergence comparison. Figure 8 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 an ideal sharing between flows: N flows each get $1/N$ of bandwidth. This changes happens within one averaging interval (100 µs). 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.

5.3.2 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 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 to 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.

(E) Amount of update traffic. Figure 9 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%. Scaling the rate updates to large networks is discussed in §6.

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.

(F) Reducing update traffic. Increasing the update threshold reduces the volume of update traffic and the processing required at servers. Figure 10 shows the measured reduction in update traffic for different thresholds compared to the 0.01 threshold in Figure 9. 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.

(G) 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 11 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.

5.3.3 Comparison to prior schemes

We compare Flowtune to DCTCP [2], pFabric [4], XCP [26], and Cubic+sfqCoDel [32].

(H) 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 [11].

Figure 12 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

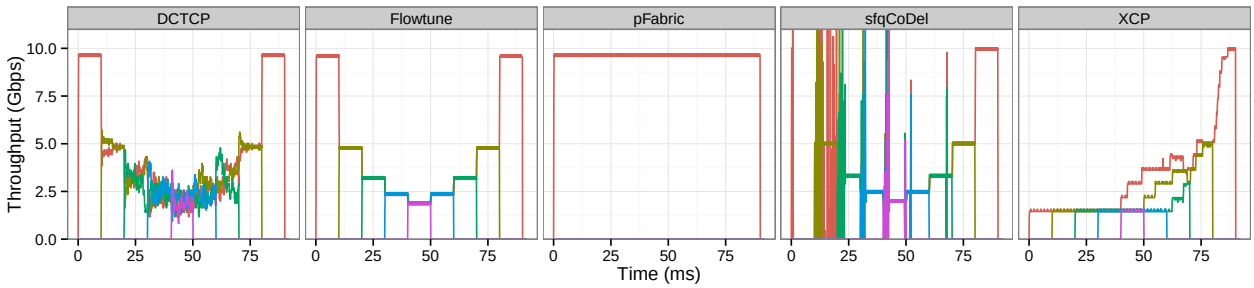


Figure 8: Flowtune achieves a fair allocation within 100 µs of a new flow arriving or leaving. In the benchmark, every 10 ms a new flow is added up to 5 flows, then flows finish one by one. DCTCP approaches a fair allocation after several milliseconds. pFabric, as designed, doesn't share the network 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.

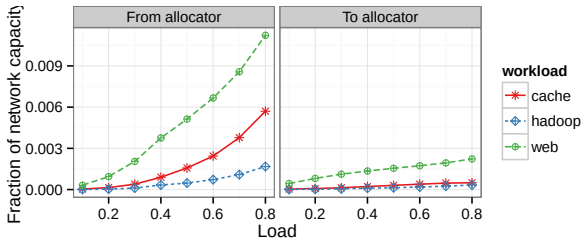


Figure 9: Overhead with Hadoop, cache, and Web workloads is $< 0.17\%$, 0.57%, and 1.13% of network capacity.

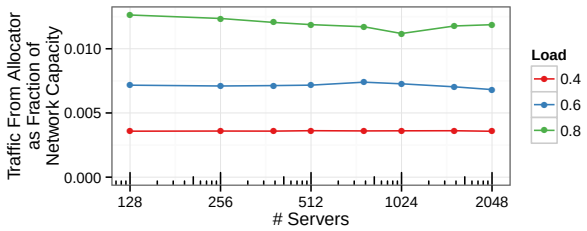


Figure 11: The fraction of rate-update traffic remains constant as the network grows from 128 to 2048 servers.

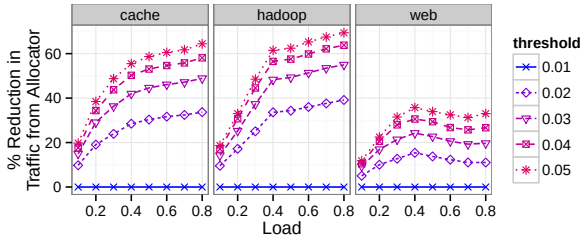


Figure 10: Notifying servers when rates change by more than a threshold substantially cuts control traffic volume.

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 better 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. XCP is conservative in allocating bandwidth (§5.3.1), which causes flows to finish slowly.

(I) Queuing delay. The following experiments collected queue lengths, drops, and throughput from each queue every 1 ms. Figure 13 shows the 99th percentile queuing delay on network paths, obtained by examining queue lengths. This queuing delay has a major contribution 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 12. XCP's conservative allocation causes its queues to remain shorter. pFabric and sfqCoDel maintain relatively long queues, but the comparison is not apples-to-apples because packets do not traverse their queues in FIFO order.

(J) Packet drops. Figure 14 shows the rate at which the network drops data, in Gigabits per second. At 0.8 load,

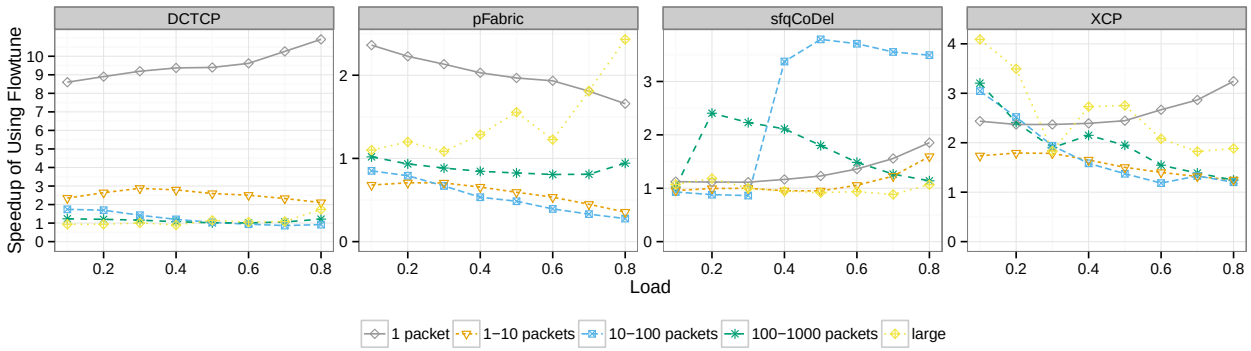


Figure 12: Improvement in 99th percentile flow completion time with Flowtune. Note the different scales of the y axis.

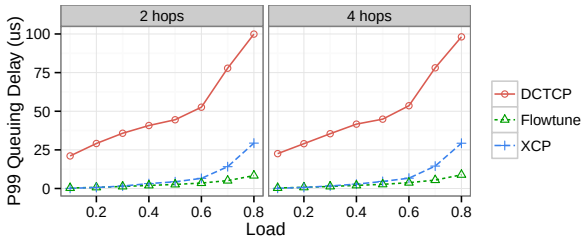


Figure 13: Flowtune keeps 2-hop and 4-hop 99th-percentile queuing delays below 8.9 µs. At 0.8 load, XCP has 3.5× longer queues, DCTCP 12×. pFabric and sfqCodel do not maintain FIFO ordering so their p99 queuing delay could not be inferred from sampled queue lengths.

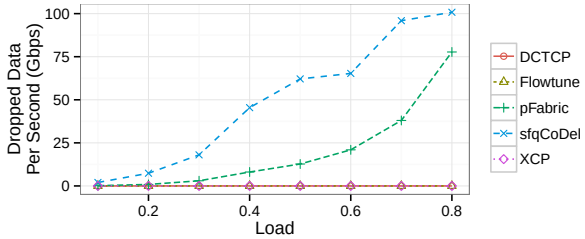


Figure 14: pFabric and sfqCoDel have a significant drop rate (1-in-13 for sfqCoDel). Flowtune, DCTCP, and XCP drop negligible amounts.

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, but timeouts due to these drops could result in high p99 FCT, which affects medium-sized flows (figure 12). 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.

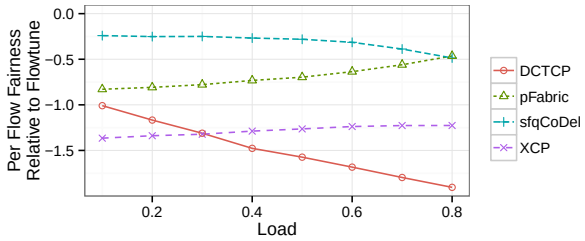


Figure 15: Comparison of proportional fairness of different schemes, i.e., $\sum \log_2(\text{rate})$. Flowtune allocates flows closer to their proportional-fair share.

(K) Fairness. Figure 15 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.

5.4 Numerical simulations

Experiments in this section compared different NUM optimizers using numerical simulations. Simulations ran the web flow size distribution described in §5.3.

(L) Over-allocation in NUM. Figure 16 shows the total amount of over-capacity allocations when there is no normalization. FGM is the Fast Weighted Gradient Method [7]. The -RT variants are optimized implementations which use single-point floating point operations and some numeric approximations for speed. 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.

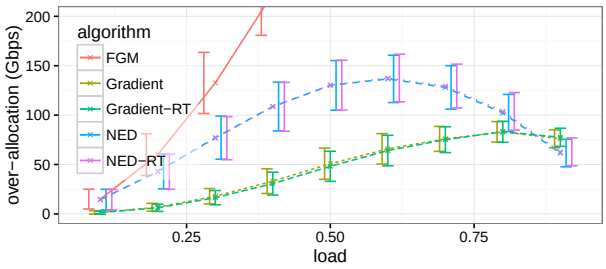


Figure 16: Normalization is necessary; without it, optimization algorithms allocate more than link capacities.

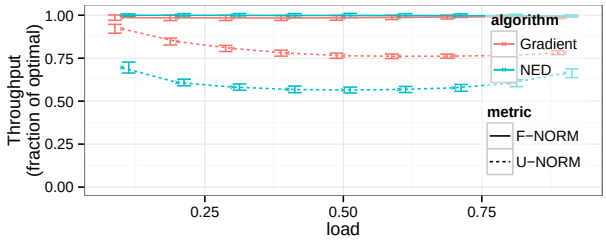


Figure 17: Normalizing with F-NORM achieves close to optimal throughput, while avoiding over-capacity allocations. U-NORM's throughput is low in comparison.

(M) Normalizer performance. We ran Gradient and NED on the same workload and recorded their throughput. After each iteration, we ran a separate instance of NED until it converged to the optimal allocation. Figure 17 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, but not the link capacities. Rather, the allocation gets more throughput than the optimal at the cost of being a little unfair to some flows.

6 Discussion

Fault-tolerance: 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.

This is a more attractive plan than in Fastpass. When the Fastpass arbiter fails, the network has no idea who should transmit next. Falling back to TCP requires the endpoints to go through slow-start before finding a good allocation. In Flowtune, the network continues to operate with close-to-optimal rates during allocator fail-over.

Path discovery: The allocator knows each flow's path through the network. 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 [14] networks, given the MPLS configuration stream. In VL2 [18]-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, endpoints can send chosen routes to the allocator.

Using Flowtune with TCP: In some settings, it is advantageous to use Flowtune to rate-limit traffic, while still using TCP. One such setting is when path information is not available. Flowtune can model the network as the servers connected to one big switch. Then, Flowtune would manages rates on the edge links between servers and the top-of-rack switches, while TCP congestion control would handle contention in the core of the network.

Flow startup; mice and elephants: When using Flowtune with TCP, the system can allow servers to start transmitting their flowlets before an allocation has arrived; the allocator would reserve a small fraction of link capacity to accomodate these flows. This allows short mice flows to finish quickly, without paying for the RTT to the allocator, and simplifies fault tolerance: upon allocator failure, the endpoints automatically use TCP on new flowlets without incurring timeouts.

Handling network failure: When links and switches fail, the network re-routes flows via alternate paths. For example, ECMP would hash flows onto the set of available links. Re-routing flows without notifying the allocator can cause queuing and packet loss on their new paths, since the allocator computes flow rates according to their old paths.

When timely notifications of re-routing events are not possible, the system can run traffic over TCP as discussed above. While re-routing information is not available, TCP gracefully handles over-allocations. An alternative is detecting failure using an external system like Pingmesh [19], and then triggering path re-discovery or re-routing for affected flows. While correct path information is being obtained, the allocator can mitigate link over-allocation by zeroing the capacity of failed links.

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. As in Fastpass,

Flowtune could prioritize or separately schedule external traffic, or adopt a different approach. 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 FlowBlock/LinkBlock abstraction can generalize to 3-tier Clos networks, or if a new method is needed.

Another approach to scaling would be running a seperate Flowtune allocator per pod, each controller treating incoming inter-pod traffic as external traffic (as discussed above). This would allow each pod to optimize its objective function on its egress inter-pod flows, but the network will not be able to globally optimize inter-pod traffic.

More scalable rate update schemes: Experiments in §5.3.2 show that rate updates have a throughput overhead of 1.12% at 0.8 load, so each allocator NIC can update 89 servers. Note that 0.8 load is on the extreme high end in some datacenters: one study reports 99% of links are less than 10% loaded, and heavily-loaded links utilize roughly 5× the lightly loaded ones [34]. Lower load translates directly to reduced update traffic (Figure 9).

In small deployments of a few hundred endpoints, it might be feasible to install a few NICs in the allocator. Figure 10 shows how increasing the update threshold reduces update traffic, which can help scale a little further, but as deployments grow to thousands of endpoints, even the reduced updates can overwhelm allocator NICs.

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 eight byte rate updates are sent. 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 would send an MTU to each intermediary with all updates to the intermediary's endpoints. The intermediary would in turn forward rate updates to each endpoint.

Hypervisors: A Flowtune endpoint needs to send flowlet start/stop notifications to the allocator, and rate-limit flows based on received allocations. A hypervisor can accomplish this without VM support by interposing itself on VM network I/O (e.g., using a vSwitch), maintaining per-flow queues, and scheduling outgoing packets. However, this approach precludes direct VM access to NIC queues (e.g., using SR-IOV) and its associated performance advantages. A potential direction could be adding hardware support for flow notification and pacing to NICs.

Detecting flowlets: Detection of when flowlets start and end can be done in the operating system, in the hypervisor (as discussed above), in a network appliance/switch (similar to a hypervisor implementation), or in some implementations the applications could participate in Flowtune directly. With OS and application flowlet detection, flowlets are clearly delineated by different `send()` socket calls, and timers are not required to detect a flowlet's end. This accurate detection is more economical to the system, since a rate is not allocated in vain to an empty flowlet while waiting for its timer to expire. Moreover, knowing the exact flowlet size allows the OS and applications to provide the allocator with advance notification of flowlet endings, further reducing wasted allocations.

When a new `send()` socket call arrives in the middle of a flowlet, an implementation can choose to coalesce the new data into the existing flowlet, and notify the allocator only when all data has finished. This is beneficial when the utility function is the same for both socket calls: the allocator will output the same rate if there are two back-to-back flowlets or one large flowlet, and coalescing helps reduce communication overhead.

7 Related work

Rate allocation. NUMFabric [31] also uses NUM to assign network rates, however switches must be modified to support its xWI protocol. Unlike Flowtune, it is distributed, so an iteration time is coupled with network RTT and the system cannot apply global normalization to make all traffic admissible.

Several systems control datacenter routes and rates, but are geared for inter-datacenter traffic. BwE [28] 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 [25]), and SWAN [23] 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 [15] and Datacenter TDMA [40] 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 [27, 29, 37] do not estimate $H_{\ell\ell}$ or use crude proxies. Gradient projection [29] adjusts prices with no weighting. Fast Weighted Gradient [7] 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. Appendix B discusses the Gradient, Newton and Newton-like methods in more detail.

Recent work [41] 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 [43] 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 runtime).

8 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 (≤2 RTTs) for networks that have between 384 and 4608 nodes.

Acknowledgements

We thank Omar Baldonado, Chuck Thacker, Prabhakaran Ganesan, Songqiao Su, Kirtesh Patil, Petr Lapukhov, Neda Beheshti, Mohana Prasad, Mohammad Alizadeh, James Zeng, Sandeep Hebbani, Jasmeet Bagga, Dinesh Bharadia, Chris Davies, and Doug Weimer for helpful discussions. We are grateful for Facebook's support of Perry through a Facebook Fellowship. Balakrishnan was supported in part by NSF grants 1526791 and 1407470, and Shah by NSF grant 1523546. We thank the industrial members of the MIT Center for Wireless Networks and Mobile Computing for their support and encouragement.

A 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

$$\mathscr{L}(\mathbf{x},\mathbf{p}) = \sum_{s\in S} U_s(x_s) - \sum_{\ell\in L} p_\ell\Big(\sum_{s\in S(\ell)} x_s - c_\ell\Big). \quad (7)$$

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

$$D(\mathbf{p}) = \max\ \mathscr{L}(\mathbf{x},\mathbf{p}) \text{ over } x_s \geq 0, \text{ for all } s \in S. \quad (8)$$

The dual optimization problem is given by

$$\min\ D(\mathbf{p}) \text{ over } p_\ell \geq 0, \text{ for all } \ell \in L. \quad (9)$$

From Slater's condition in classical optimization theory, the utility of the solution of (2) is equal to its Lagrangian dual's (9), and given the optimal solution $\mathbf{p}^\star$ of (9) it is possible to find the optimal solution for (2) from (8), i.e., using the rate update step. More details on solving NUM using Lagrange multipliers appear in [27, 6].

B Related NUM algorithms

This appendix surveys three related NUM algorithms.

Gradient. Arguably the simplest algorithm for adjusting prices is Gradient projection [29], 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

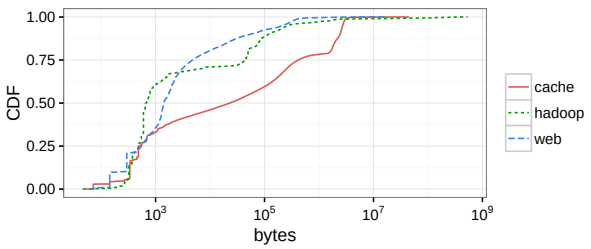


Figure 18: CDF of flow size distributions (reproduced from [34]) used in §5.

Metric	cache	hadoop	web
mean	567658.4	1296082.4	33105.3
median	22924	651	1419
p90	2432831	117471	55179
p95	2716140	266706	208966
p99	3131038	6405830	417147
p999	5663439	251359175	2560769

Table 1: Statistics of the different flow size distributions (bytes).

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 within 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 drawback is that network throughput must be averaged over relatively large time intervals, so estimating the diagonal is slow.

C Simulation CDFs

This section reproduces the flow size distribution graphs from [34], for completeness. Data from the paper has been open-sourced in the "Facebook Network Analytics Data Sharing" Facebook group. The distributions are based on the "all" category from the original publication.

Figure 18 shows the flow size CDF. Table 1 summarizes statistics of the different workloads.

References

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

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

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

[4] Alizadeh, M., Yang, S., Sharif, M., Katti, S., McKeown, N., Prabhakar, B., and Shenker, S. pFabric: Minimal near-optimal datacenter transport. In *SIGCOMM* (2013).

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

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

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

[8] Brakmo, L. S., O'Malley, S. W., and Peterson, L. L. TCP Vegas: New Techniques for Congestion Detection and Avoidance. In *SIGCOMM* (1994).

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

[10] Chowdhury, M., Zhong, Y., and Stoica, I. Efficient coflow scheduling with varys. In *SIGCOMM* (2014).

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

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

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

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

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

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

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

[18] GREENBERG, A., HAMILTON, J. R., JAIN, N., KANDULA, S., KIM, C., LAHIRI, P., MALTZ, D. A., PATEL, P., AND SENGUPTA, S. VL2: A Scalable and Flexible Data Center Network. In *SIGCOMM* (2009).

[19] GUO, C., YUAN, L., XIANG, D., DANG, Y., HUANG, R., MALTZ, D., LIU, Z., WANG, V., PANG, B., CHEN, H., ET AL. Pingmesh: A large-scale system for data center network latency measurement and analysis. *SIGCOMM* (2015).

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

[21] HOE, J. C. Improving the start-up behavior of a congestion control scheme for tcp. In *SIGCOMM* (1996).

[22] HONG, C. Y., CAESAR, M., AND GODFREY, P. Finishing Flows Quickly with Preemptive Scheduling. In *SIGCOMM* (2012).

[23] HONG, C.-Y., KANDULA, S., MAHAJAN, R., ZHANG, M., GILL, V., NANDURI, M., AND WATTENHOFER, R. Achieving High Utilization with Software-Driven WAN. In *SIGCOMM* (2013).

[24] JACOBSON, V. Congestion Avoidance and Control. In *SIGCOMM* (1988).

[25] JAIN, S., KUMAR, A., MANDAL, S., ONG, J., POUTIEVSKI, L., SINGH, A., VENKATA, S., WANDERER, J., ZHOU, J., ZHU, M., ZOLLA, J., HÖLZLE, U., STUART, S., AND VAHDAT, A. B4: Experience with a Globally-deployed Software Defined Wan. In *SIGCOMM* (2013).

[26] KATABI, D., HANDLEY, M., AND ROHRS, C. Congestion Control for High Bandwidth-Delay Product Networks. In *SIGCOMM* (2002).

[27] KELLY, F. P., MAULLOO, A. K., AND TAN, D. K. Rate Control for Communication Networks: Shadow prices, Proportional Fairness and Stability. *Journal of the Operational Research Society* (1998), 237–252.

[28] KUMAR, A., JAIN, S., NAIK, U., RAGHURAMAN, A., KASINADHUNI, N., ZERMENO, E. C., GUNN, C. S., AI, J., CARLIN, B., AMARANDEI-STAVILA, M., ROBIN, M., SIGANPORIA, A., STUART, S., AND VAHDAT, A. Bwe: Flexible, hierarchical bandwidth allocation for wan distributed computing. In *SIGCOMM* (2015).

[29] LOW, S. H., AND LAPSLEY, D. E. Optimization Flow Control–I: Basic Algorithm and Convergence. *IEEE/ACM Trans. on Networking 7*, 6 (1999), 861–874.

[30] MITTAL, R., DUKKIPATI, N., BLEM, E., WASSEL, H., GHOBADI, M., VAHDAT, A., WANG, Y., WETHERALL, D., ZATS, D., ET AL. TIMELY: RTT-based congestion control for the datacenter. In *SIGCOMM* (2015).

[31] NAGARAJ, K., BHARADIA, D., MAO, H., CHINCHALI, S., ALIZADEH, M., AND KATTI, S. NUMFabric: Fast and Flexible Bandwidth Allocation in Datacenters. In *SIGCOMM* (2016).

[32] NICHOLS, K., AND JACOBSON, V. Controlling queue delay. *Communications of the ACM 55*, 7 (2012), 42–50.

[33] PERRY, J., OUSTERHOUT, A., BALAKRISHNAN, H., SHAH, D., AND FUGAL, H. Fastpass: A centralized "zero-queue" datacenter network. In *SIGCOMM* (2014).

[34] ROY, A., ZENG, H., BAGGA, J., PORTER, G., AND SNOEREN, A. C. Inside the social network's (datacenter) network. In *SIGCOMM* (2015).

[35] SHREEDHAR, M., AND VARGHESE, G. Efficient Fair Queueing Using Deficit Round Robin. In *SIGCOMM* (1995).

[36] SINGH, A., ONG, J., AGARWAL, A., ANDERSON, G., ARMISTEAD, A., BANNON, R., BOVING, S., DESAI, G., FELDERMAN, B., GERMANO, P., KANAGALA, A., PROVOST, J., SIMMONS, J., TANDA, E., WANDERER, J., HÖLZLE, U., STUART, S., AND VAHDAT, A. Jupiter rising: A decade of clos topologies and centralized control in google's datacenter network. In *SIGCOMM* (2015).

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

[38] TAI, C., ZHU, J., AND DUKKIPATI, N. Making Large Scale Deployment of RCP Practical for Real Networks. In *INFOCOM* (2008).

[39] TAN, K., SONG, J., ZHANG, Q., AND SRIDHARAN, M. A compound TCP approach for high-speed and long distance networks. In *INFOCOM* (2006).

[40] VATTIKONDA, B. C., PORTER, G., VAHDAT, A., AND SNOEREN, A. C. Practical TDMA for Datacenter Ethernet. In *EuroSys* (2012).

[41] WEI, E., OZDAGLAR, A., AND JADBABAIE, A. A Distributed Newton Method for Network Utility Maximization–I: Algorithm. *IEEE Trans. on Automatic Control 58*, 9 (2013), 2162–2175.

[42] WINSTEIN, K., SIVARAMAN, A., AND BALAKRISHNAN, H. Stochastic Forecasts Achieve High Throughput and Low Delay over Cellular Networks. In *NSDI* (2013).

[43] ZARGHAM, M., RIBEIRO, A., OZDAGLAR, A., AND JADBABAIE, A. Accelerated dual descent for network flow optimization. *IEEE Trans. on Automatic Control 59*, 4 (2014), 905–920.