Centralized Performance Control for Datacenter Networks

by

Jonathan Perry

Submitted to the Department of Electrical Engineering and Computer Science in partial fulfillment of the requirements for the degree of

Doctor of Philosophy in Computer Science

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

June 2017

© Massachusetts Institute of Technology 2017. All rights reserved.
Centralized Performance Control for Datacenter Networks

by

Jonathan Perry

Submitted to the Department of Electrical Engineering and Computer Science on May 19, 2017, in partial fulfillment of the requirements for the degree of Doctor of Philosophy in Computer Science

Abstract

An ideal datacenter network should allow operators to specify policy for resource allocation between users or applications, while providing several properties, including low median and tail latency, high utilization (throughput), and congestion (loss) avoidance. Current datacenter networks inherit the principles that went into the design of the Internet, where packet transmission and path selection decisions are distributed among the endpoints and routers, which impede obtaining the desired properties. Instead, we propose that a centralized controller should tightly regulate senders' use of the network according to operator policy, and evaluate two architectures: Fastpass and Flowtune.

In Fastpass, the controller decides when each packet should be transmitted and what path it should follow. Fastpass incorporates two fast algorithms: the first determines the time at which each packet should be transmitted, while the second determines the path to use for that packet. We deployed and evaluated Fastpass in a portion of Facebook's datacenter network. Our results show that Fastpass achieves high throughput comparable to current networks at a 240× reduction in queue lengths, achieves much fairer and consistent flow throughputs than the baseline TCP, scales to schedule 2.21 Terabits/s of traffic in software on eight cores, and achieves a 2.5× reduction in the number of TCP retransmissions in a latency-sensitive service at Facebook.

In Flowtune, congestion control decisions are made at the granularity of a flowlet, not a packet, so allocations change only when flowlets arrive or leave. The centralized allocator receives flowlet start and end notifications from endpoints, and computes optimal rates using a new, fast method for network utility maximization. A normalization algorithm ensures allocations do not exceed link capacities. Flowtune updates rate allocations for 4600 servers in 31 μs regardless of link capacities. Experiments show that Flowtune outperforms DCTCP, pFabric, sfqCoDel, and XCP on tail packet delays in various settings, and converges to optimal rates within a few packets rather than over several RTTs. EC2 benchmarks show a fairer rate allocation than Linux's Cubic.

Thesis Supervisor: Hari Balakrishnan
Title: Professor

Thesis Supervisor: Devavrat Shah
Title: Professor
Acknowledgments

By far the most substantial contribution to this work came from my family. My parents and grandparents have shown me an ingrained appreciation of education and the pursuit of knowledge since I was an infant, spent immense amounts of time and effort in raising and teaching me, and gave me freedom to set out on academic endeavors. I am eternally grateful to my wife, Noa, for joining in this adventure halfway across the planet; for being a true friend and confidant; for together building a family and raising our daughter, Ayala during her infant years. Noa, without your love and support over the years, this work would not have existed. I look forward to our next adventures together!

To my advisors, Hari and Devavrat, thank you for taking me in. Thank you for your patience and your insight, for openly sharing your views on a wide range of subjects from the metric system to US politics, for opening your homes. May I have the strength and ability to mentor and collaborate with others like you mentored me, with a gentle hand and a smile.

To our admin, Sheila Marian, thank you for taking care of me and the group, and for treating Noa, Ayala and me like family!

To my friends from CSAIL, from Eastgate, Westgate, and Walden St., from back home and those who scattered around the globe – you know who you are – and to my office-mates who brightened my days at the office, Fadel Adib, Eugene Wu, Holger Pirk, Raul Castro Fernandez, and Oscar Moll – thank you for your friendship. I hope to enjoy your company and good spirits for many more years to come!

I would like to thank my co-authors Amy Ousterhout, Hans Fugal, Petr Lapukhov, Peter Iannucci, and Kermin Elliott Fleming, and of course Hari and Devavrat. You have been exceptional collaborators, and each added your special color and depth to our work.

Omid Aryan and Brendan Chang accepted me as their M.Eng mentor, Mohana Prasad as his summer internship mentor, and Ambika Krishnamachar and Deepti Raghavan as a SuperUROP mentor. Thank you for your trust. I enjoyed working with you!

Relly, Yigal, Taly and Tomer were our adopting family away from home. Thank you for welcoming us into your lives!
The collaboration with Facebook had enriched my PhD tremendously, and I am grateful to Omar Baldonado and Sanjeev Kumar of Facebook for their enthusiastic support of this collaboration.

To Frans Kaashoek and PDOS, thank you for welcoming me into PDOS and inviting me to TA 6.824. I always looked forward to the group meetings and lunches. Frans, I thank you for your open door policy and dedicating time over the years.

To Nick McKeown, thank you for accepting me into your Stanford group during my last PhD year, and for your generous help and advice.

I thank my EECS academic advisor, Prof. Berthold K.P. Horn for having my back in ensuring I achieved the PhD program’s milestones.

I thank Srikanth Kandula for his mentorship during the summer of 2011 at MSR, and for a lot of my early knowledge of datacenter networks.

Many people provided support and advice over the years. I am grateful to Doug Weimer for his generous assistance with the Facebook infrastructure, Garrett Wollman and Jon Proulx at MIT CSAIL for their help and efforts in setting up environments for our early experiments, and David Oran of Cisco for his help. I thank Mohammad Alizadeh, Rodrigo Fonseca, John Ousterhout, George Varghese, Chuck Thacker, Amin Vahdat, Steve Hand, Prabhakaran Ganesan, Songqiao Su, Kirtesh Patil, Neda Beheshti, James Zeng, Sandeep Hebbani, Jasmeet Bagga, Dinesh Bharadia, Chris Davies, Andreas Nowatzyk, Tom Rodeheffer.

I would like to thank the SIGCOMM, NSDI, OSDI and HotNets reviewers for their insightful feedback.

This work was supported in part by National Science Foundation grants, a Jacobs Presidential Fellowship, Claude E. Shannon Research Assistantship, the Elie Shaio Memorial Award, Andrew and Erna Viterbi Fellowship, and a Facebook Graduate Fellowship. I thank the people and organizations who made these sources available.
Contents

1 Introduction 15
  1.1 Fastpass .................................................. 17
  1.2 Flowtune .................................................. 23
  1.3 The bigger picture ....................................... 27
  1.4 Contributions ........................................... 28

2 Fastpass: Packet-based Allocation 31
  2.1 Fastpass Architecture ..................................... 31
  2.2 Timeslot Allocation ....................................... 33
  2.3 Theoretical Properties of the Timeslot Allocator ........ 37
  2.4 Path Selection ............................................ 40
  2.5 Handling Faults ........................................... 43
  2.6 Implementation .......................................... 45
  2.7 Evaluation ................................................. 48
  2.8 Discussion ................................................. 58
  2.9 Summary .................................................. 61

3 Flowtune: Flowlet-based Allocation 63
  3.1 Rate Allocation in Flowtune ............................. 63
  3.2 Rate normalization ....................................... 67
  3.3 Scalability ................................................ 69
  3.4 Evaluation ................................................. 71
  3.5 Discussion ............................................... 83
<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.6  Summary</td>
<td>87</td>
</tr>
<tr>
<td>3.A  Appendix: Why price duality works</td>
<td>89</td>
</tr>
<tr>
<td>3.B  Appendix: Simulation CDFs</td>
<td>90</td>
</tr>
<tr>
<td>4  Related work</td>
<td>91</td>
</tr>
<tr>
<td>5  Conclusion</td>
<td>95</td>
</tr>
<tr>
<td>5.1  Future work</td>
<td>96</td>
</tr>
<tr>
<td>5.2  A personal note</td>
<td>97</td>
</tr>
</tbody>
</table>
List of Figures

1-1 Example allocation of packet transmission from host A to host B in Fastpass. . . . 17
1-2 Datacenter topologies frequently sport multiple possible paths between endpoints. 18
1-3 Structure of the arbiter, showing the timeslot allocator, path selector, and the client-
arbiter communication. ................................................................. 19
1-4 Maximal matchings are good matchings: summary of this work’s theoretical contri-
butions in context of prior results. .......................................................... 21
1-5 Example of allocation in Flowtune. .................................................. 24
1-6 Iterative solvers to NUM adjust rates via prices. Too small adjustment result in slow
convergence; too aggressive updates result in oscillation and failure to converge. . . 25
1-7 NED uses the slope of the explicit utility functions to choose the size of incremental
updates, similar to the Newton method, requiring fewer updates to converge. . . . 25
1-8 Motivation for the allocator architecture. .......................................... 26
1-9 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. ................................................................. 27
1-10 Responsibilities for networking activities in the traditional, SDN, Flowtune and
Fastpass architectures. ................................................................. 27
2-1 Pipelined timeslot allocation. The allocator for timeslot $t$ processes the demands
not fully satisfied by the allocator for $t - 1$ ........................................ 35
2-2 Timeslot allocation for the max-min fairness allocation objective. ............... 36
2-3 Path selection. (a) input matching (b) ToR graph (c) edge-colored ToR graph (d) edge-colored matching. .......................................................... 41

2-4 Multicore allocation: (1) allocation cores assign packets to timeslots, (2) path selection cores assign paths, and (3) communication cores send allocations to endpoints. .......................................................... 47

2-5 Switch queue lengths sampled at 100ms intervals on the top-of-rack switch. The diagram shows measurements from two different 20 minute experiments: baseline (red) and Fastpass (blue). Baseline TCP tends to fill switch queues, whereas Fastpass keeps queue occupancy low. .......................................................... 51

2-6 Histogram of ping RTTs with background load using Fastpass (blue) and baseline (red). Fastpass’s RTT is $15.5 \times$ smaller, even with the added overhead of contacting the arbiter. .......................................................... 52

2-7 Each connection’s throughput, with a varying number of senders. Even with 1s averaging intervals, baseline TCP flows achieve widely varying rates. In contrast, for Fastpass (bottom), with 3, 4, or 5 connections, the throughput curves are on top of one another. The Fastpass max-min fair timeslot allocator maintains fairness at fine granularity. The lower one- and two-sender Fastpass throughput is due to Fastpass qdisc overheads (§2.7.2). .......................................................... 53

2-8 As more requests are handled, the NIC polling rate decreases. The resulting queueing delay can be bounded by distributing request-handling across multiple comm-cores. .......................................................... 54

2-9 The arbiter requires 0.5 Gbits/s TX and 0.3 Gbits/s RX bandwidth to schedule 150 Gbits/s: around 0.3% of network traffic. .......................................................... 55

2-10 Path selection routes traffic from 16 racks of 32 endpoints in <12μs. Consequently, 10 pathsel-cores would output a routing every <1.2μs, fast enough to support 10 Gbits/s endpoint links. Error bars show one standard deviation above and below the mean. .......................................................... 56

2-11 Distribution of the sending and receiving rates of one production server per 100 microsecond interval over a 60 second trace. .......................................................... 57
2-12 99th percentile web request service time vs. server load in production traffic. Fastpass shows a similar latency profile as baseline.

2-13 Live traffic server load as a function of time. Fastpass is shown in the middle with baseline before and after. The offered load oscillates gently with time.

2-14 Median server TCP retransmission rate during the live experiment. Fastpass (middle) maintains a $2.5 \times$ lower rate of retransmissions than baseline (left and right).

3-1 Illustration of how NUM chooses a flow rate.

3-2 Partitioning of network state.

3-3 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.

3-4 8-to-1 experiment on Amazon EC2. Flowtune shares available throughput more fairly than baseline Linux.

3-5 AWS EC2 data aggregation benchmarks. Flowtune coflows are more predictable and generally faster.

3-6 Flowtune achieves a fair allocation within 100 $\mu$s 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.

3-7 Overhead with Hadoop, cache, and Web workloads is $< 0.17\%$, $0.57\%$, and $1.13\%$ of network capacity.

3-8 Notifying servers when rates change by more than a threshold substantially cuts control traffic volume.

3-9 The fraction of rate-update traffic remains constant as the network grows from 128 to 2048 servers.
3-10 Improvement in 99th percentile flow completion time with Flowtune. Note the different scales of the y axis. .................................................. 80

3-11 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. ......................................................................... 81

3-12 pFabric and sfqCoDel have a significant drop rate (1-in-13 for sfqCoDel). Flowtune, DCTCP, and XCP drop negligible amounts. ......................................................... 82

3-13 Comparison of proportional fairness of different schemes, i.e., $\sum \log_2(\text{rate})$. Flowtune allocates flows closer to their proportional-fair share. ................................. 82

3-14 Normalization is necessary; without it, optimization algorithms allocate more than link capacities. ................................................................. 83

3-15 Normalizing with F-NORM achieves close to optimal throughput, while avoiding over-capacity allocations. U-NORM's throughput is low in comparison. ...... 84

3.B.1 CDF of flow size distributions (reproduced from [57]) used in §3.4. ................. 90
List of Tables

2.1 Fastpass: Summary of Results ........................................ 49
2.2 Distribution of switch queue lengths in KB. ....................... 50
2.3 Latency comparison, Baseline vs. Fastpass. ....................... 51
2.4 Throughput variability, Baseline vs. Fastpass. ................... 52
2.5 Throughput of different alloc-core pipelines lengths. .......... 55

3.1 NED Notation .............................................................. 64
3.2 Flowtune: Summary of Results. ...................................... 72
3.3 Comparison of coflow completion time, Baseline vs. Flowtune. .. 74
3.4 NED iteration time for different numbers of cores, nodes, and flows. 75
3.B.1 Statistics of the different flow size distributions (bytes). ....... 90
Chapter 1

Introduction

Many applications today are offered as a service: the application is deployed on compute and storage infrastructure controlled by the serving organization, allowing developers to quickly deploy new features and updates, and to better assure the availability of the service and its associated data in case of failure.

The datacenters hosting these applications are frequently shared infrastructure, hosting many tenants and applications. In the public cloud, applications belong to multiple companies; in the private cloud, different teams and business units. Furthermore, developers partition applications into many components (“microservices”). Microservices allow each component to be developed, deployed and scaled independently, greatly accelerating innovation.

The networks supporting datacenters are also shared, and applications compete for the ability to communicate through the network. When the network fails to allocate the right resources to applications, user experience suffers due to high response times and failure rates. Impaired performance damages user productivity, hurts revenue for companies that sell products online or show advertising, and for any organization, harms the organization’s reputation and its ability to retain users.

Each application independently requests the network to transmit its data with unpredictable timing, destinations, and transfer sizes. This makes it hard for a network architecture to effectively manage the way its resources are divided among applications.

Current network architectures are highly distributed: endpoints decide when and how many packets to send (“congestion control”), and switches select packet paths (“routing”) and decide
how to mark, drop, and queue packets ("scheduling and queue management"). Because of the strong fault-tolerance and scalability of these distributed architectures, these architectures have been widely adopted as the foundation of the Internet and practically all enterprise networks, and their distributed principles have been engrained into networking research and practice.

However, current architectures also have significant disadvantages. First, with any non-trivial resource allocation policy, configuring the distributed mechanisms at endpoints and switches to achieve desired application outcomes becomes practically impossible, as components interact in subtle ways. Next, the different components interact through the network, so it frequently takes many round-trip times (RTTs) for the network to achieve the desired resource allocation. Finally, changing the network behavior could require reconfiguring every component or in worse cases to replace the deployed mechanisms, which is expensive.

In this thesis we set out to address these disadvantages.

Is it possible to design a network in which: (1) the operator can explicitly specify the network's resource allocation policies, (2) the network quickly and consistently enforces these policies, (3) it is easy to deploy, configure, and update network policy, preferably in software, and (4) the network supports high utilization and low latency?

Such a network would be useful in many contexts, but especially in datacenters where the network is administered by one organization, link rates are at technology’s bleeding edge, and system operators have to contend with multiple users and a rich mix of workloads. Meeting complex service-level objectives and application-specific goals would be much easier in a network that delivered these three ideals.

We advocate what may seem like a rather extreme approach: to centrally exercise (very) tight control over when endpoints send packets.

Besides better apportioning resources according to policy, centralized transmission control also solves long standing architectural problems in current networks: queueing latency and packet drops. In current networks, many endpoints can simultaneously burst packets that pass through a single link. To absorb these bursts, network use queues leading to delays that rise and fall and to packet loss when bursts exceed queue capacity.

With tight control, flow rates can match available network capacity over the time-scale of individual packets. Not only will persistent congestion be eliminated, but packet latencies will not
Figure 1-1: Example allocation of packet transmission from host A to host B in Fastpass.

rise and fall, queues will never vary in size, tail latencies will remain small, and packets will never be dropped due to buffer overflow. Further, networks traditionally control queueing and packet drops by over-provisioning link capacity, leaving a share of capacity unused. Solving queueing and packet drops allows higher utilizations, improving networks cost-efficiency.

Three main concerns arise when considering centralized allocation:

1. **Latency.** Will allocation excessively delay packets?
2. **Scale.** How large can centrally allocated clusters be?
3. **Fault-tolerance.** Can the system gracefully handle controller and switch failure?

This thesis proposes two architectures, Fastpass and Flowtune, and specifies and evaluates solutions that address these concerns.

### 1.1 Fastpass

Considering the spectrum of network control approaches from the most distributed to the most centralized, Fastpass takes the extreme centralized end: it chooses the exact transmission time and network path of every single packet in the network.

**Example.** Figure 1-1 shows an example allocation in Fastpass. Say server A wants to send a packet to server B. A first queues the packet and sends a request to the arbiter (shown in red). The arbiter knows about all the packets servers want to send, so it can choose a time and path for the packet such that the packet causes no queueing inside the network: the packet arrives at each switch exactly when its designated output becomes available (yellow). The arbiter informs the server of the allocation (green), and at the designated time, A sends the packet on the allocated path (blue).
Network latencies. The figure also shows example timing for the different steps: transmitting requests and allocations should take 5 μs and 15 μs respectively, and computation at the arbiter should take 1-20 μs, depending on network size. While allocation adds 20-40 μs for every packet, the major benefit is that once a packet is allocated, it has the ultimate network traversal experience: no queueing and no packet drops. Compared to current networks, the best possible one-way delay in Fastpass is therefore 20-40 μs larger, however under load the average and tail latencies improve dramatically (§2.7.1). The idea is analogous to a hypothetical road traffic control system in which a central entity tells every vehicle when to depart and which path to take. Then, instead of waiting in traffic, cars can zoom by all the way to their destinations.

But how could an arbiter allocate traffic on a datacenter network? Network transfers start and terminate extremely frequently, and in any non-trivial deployment, it is practically impossible to predict all the transfers’ source, destination, transfer size and timing. Further, common topologies sport multiple paths between most endpoint pairs (figure 1-2); allocation should load-balance traffic across these paths without creating congestion on any particular link.

Timeslots. The first step we took to simplify the problem statement was to divide time into timeslots, in which each source sends to at most one destination, and each destination receives from at most one source. Timeslots discretize the allocator’s decisions, as the allocator doesn’t need to allocate fractions of a sender’s throughput among multiple receivers. Instead, it allocates the entire timeslot to one receiver. The sender can send multiple packets to the receiver – as many as fit in the timeslot. In the prototype, we set timeslot duration to the transmission time of the largest possible packet (an MTU, Maximum Transmission Unit), e.g., 1.2 μs on a 10 Gbits/s network. Timeslots can cause waste due to internal fragmentation; we discuss this in §2.8.4.
Untangling timeslot allocation from path selection. Packet-level control over timing and paths allows the arbiter in the Fastpass architecture to perfectly balance load across the multiple paths available in today's datacenters. Our first implementation jointly chose which endpoints will transmit and the path that transmissions will take: for a given sender-receiver pair, the arbiter traverses all paths between the sender and receiver, and allocates traffic through a path if all links on a path are yet unallocated. This was expensive and hard to scale.

Luckily, a circuit-switching result [36] enables a clean separation of concerns: timeslot allocation can match senders and receivers as if the network is a big switch; then a good path assignment to this allocation is guaranteed to exist. We further found that this path assignment not only exists but can also be efficiently computed: a series of algorithmic advances in edge coloring between 2001-2005 [18, 7, 67] paved the way for the fast edge-coloring implementation in Fastpass (§2.4).

Fastpass architecture. Figure 1-3 shows the Fastpass architecture. The endpoints queue outgoing packets and send allocation requests to the arbiter. The arbiter performs timeslot allocation and path selection, and notifies endpoints of their allocated timeslots and paths.

Fastpass includes three key components:

1. A timeslot allocation algorithm at the arbiter to determine when each endpoint’s packets should be sent (§2.2).
2. A path assignment algorithm at the arbiter to assign a path to each packet (§2.4).
3. A reliable control protocol between endpoints and the arbiter and a replication strategy to handle network and arbiter failures (§2.5).
**Timeslot allocation.** The arbiter runs a heuristic algorithm for maximal matching. To allocate a timeslot, the allocator iterates over endpoint requests, and allocates a source-destination pair if the source and destination are both unallocated in the timeslot. The implementation keeps a bit for every source and for every destination, so checking if a request is possible is an AND of the source and destination bits. To allocate a pair, the code sets both bits to zero and adds an allocation record to an array. These can be implemented very efficiently on commodity CPUs.

**Enforcing operator policies.** The timeslot allocator iterates over requests to produce a matching. How can it be modified to enforce different operator policies? It turns out that *request processing order matters*; different objectives can be achieved with different orderings. For example, Shortest Remaining Time First (SRTF) can be achieved by first processing requests with the lowest number of remaining timeslots, or max-min fair allocation by first processing the least-recently allocated requests. A re-ordering component sorts requests into priority bins (64 bins in our implementation), and releases these bins in-order to the timeslot allocator. This causes some request delay while requests wait in bins, however policies tend to have limited re-ordering (true for the SRTF and max-min fair policies), so delays are small in practice.

**Scaling timeslot allocation.** To scale the timeslot allocator, we set out to extend the algorithm to leverage multiple CPU cores. Unfortunately, parallel maximal matching algorithms [8, 47] tend to focus on hardware implementations and are extremely inefficient on CPUs. Worse, parallel implementation do not preserve the request ordering required for policy enforcement. We realized it is not necessary to speed up a single timeslot allocation. Instead, cores can allocate a set of consecutive timeslots, with each core responsible for a single timeslot. The remaining demand from one timeslot is the input to the next timeslot, forming a pipeline of cores. Cores only need to communicate along the pipeline, and demands are transferred in small batches between cores to reduce overhead.

**Are maximal matchings good matchings?** The Fastpass timeslot allocator uses a heuristic to find matchings. Before entrusting a datacenter network to such a heuristic, it is important to understand how good (or bad) the produced matchings are. In this work, we bound the maximal matching’s average latency by $2 \times$ the optimal scheduler’s latency on its worse-case workload.\(^1\) Figure 1-4

---

\(^1\) under a standard $2 \times$ speedup assumption, see §2.3 for details.
**Maximal Matching**
with 2C network capacity

<table>
<thead>
<tr>
<th>Dai-Prabhakar ’00:</th>
<th>Finite average latency ≤ Finite average latency</th>
</tr>
</thead>
<tbody>
<tr>
<td>This work:</td>
<td>Average latency ≤ 2× Average latency</td>
</tr>
</tbody>
</table>

Figure 1-4: Maximal matchings are good matchings: summary of this work’s theoretical contributions in context of prior results.

shows this in the context of related work. Note that this is an upper bound, and in practice, maximal matching tends to perform very well: queueing remains low even at high utilization.

**Clock synchronization.** The arbiter allocates timeslots such that there is no queueing in the network. To avoid queueing, the endpoints must transmit packets precisely at allocated times. We implemented Fastpass in the Linux kernel using high-precision timers (hrtimers) to time transmitted packets and deployed and configured the IEEE1588 Precision Time Protocol (PTP) to achieve sub-microsecond network-wide time synchronization. When endpoints are not perfectly synchronized, some queueing results. Fastpass uses switch queues to absorb errors in synchronization. We found queues to be proportional to clock discrepancy and are small in practice (see §2.6.3 and §2.7.1).

**Relaxing synchronization.** While PTP achieves good synchronization results, it was hard to deploy: it requires a carefully tuned configuration and takes many minutes to converge after synchronization disruptions. In later versions of Fastpass, we used arbiter-to-endpoint allocation packets to clock transmissions. Allocation packets experience some jitter, so endpoints do not send precisely at their allocated times. However, the added timing noise is small, and as mentioned above results only in a small amount of queueing (§2.7.1).

**Experimental results.** We conducted several experiments with Fastpass running in a portion of Facebook’s datacenter network. Our main findings are:

1. High throughput with nearly-zero queues: On a multi-machine (n-to-1) bulk transfer workload, Fastpass achieved throughput only 1.6% lower than baseline TCP, while reducing the switch queue size from a median of 4.35 Mbytes to under 18 Kbytes. The resulting RTT reduced from 3.56 ms to 230 μs.

21
2. Consistent (fair) throughput allocation and fast convergence: In a controlled experiment with multiple concurrent flows starting and ending at different times, Fastpass **reduced the standard deviation of per-flow throughput by a factor over 5200x** compared to the baseline with five concurrent TCP connections.

3. Scalability: Our implementation of the arbiter shows nearly linear scaling of the allocation algorithm from one to eight cores, with the **8-core allocator handling 184.3M timeslots per second (2.21 Terabits/s on 10 Gbits/s links)**. The arbiter responds to requests within tens of microseconds even at high load.

4. Reduced retransmissions: On a real-world latency-sensitive service located on the response path for user requests, Fastpass **reduced the occurrence of TCP retransmissions by 2.5x**, from between 4 and 5 per second to between 1 and 2 per second.

**Added benefit: eliminating incast.** Incast occurs when concurrent requests to many servers triggers concurrent responses. With small router queues, response packets are lost, triggering delay-inducing retransmission timeouts [51], whereas large queues cause delays due to queueing. Current solutions are approximations of full control, based on estimates and assumptions about request RTTs, and solve the problem only partially [54, 75]. Fastpass schedules concurrent responses in different timeslots, thereby avoiding loss and retransmission timeouts.

**Limitation: scaling.** The Fastpass architecture provides very tight control of application network performance, however at the cost of performing per-packet work. Link speeds have been increasing: from 10 Gbits/s networks to 25 Gbits/s, 40 Gbits/s, 100 Gbits/s and beyond. Fastpass is at a disadvantage with increasing link speeds. First, Fastpass could increase its compute power by adding more cores to the pipeline, however this will increase latency, while the networking hardware latencies decrease. Second, Fastpass’s pipelining architecture cannot scale indefinitely, as short requests tend to get fulfilled by the head of the pipeline, leaving its tail underutilized.

**Limitation: algorithmic design.** Fastpass enforces operator policy by re-ordering requests, which requires the policy to be expressed as a algorithm that assigns each request to one of a set of bins; bins are then passed to the timeslot allocator in order. Some policies can be expressed very naturally this way, such as SRTF and max-min fair, however others are harder, especially those pertaining to longer lived flows, where operators would like to control the relative allocation of throughput in different settings, and care less about per-packet latency.
These two limitations motivated us to develop Flowtune.

1.2 Flowtune

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. Over the past thirty years, network congestion control schemes—whether distributed [39, 14, 33, 31, 70] or centralized [56], whether end-to-end or with switch support [21, 27, 28, 62, 43, 66, 52], and whether in the wide-area Internet [22, 74] or in low-latency datacenters [4, 5, 6, 34, 49]—have operated at the granularity of individual packets.

Flowlet-based allocation. In Flowtune, we adopt the position that a flowlet, and not a packet, is a better granularity for congestion control. By “flowlet”, we mean the application’s transmission granularity: a flowlet starts when a socket has no backlog and an application calls the send() socket call, and ends when there is a threshold amount of time during which the sender’s queue is empty. Because application developers care about the network’s performance over these send() calls, it is natural to both express and enforce policy over flowlets. 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.²

Example. Figure 1-5 illustrates an allocation in Flowtune. Whenever a flowlet starts or ends, the endpoint notifies a centralized allocator. Upon receiving each such message, the allocator computes the best values of flow rates for each endpoint according to policy, and updates the endpoints.

NUM. Flowtune adopts the network utility maximization (NUM) framework, previously developed to analyze distributed congestion control protocols [44, 46]. In Flowtune, network operators specify explicit utility functions, and the allocator maximizes the total network utility.

Allocation complexity. With flowlet-based allocation, the network can avoid per-packet work, as rates would have to change only when new flows arrive or flows leave the system. However, 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.

²Long-lived flows that send intermittently generate multiple flowlets.
The allocator assigns rates to flowlets. When links are free, flowlets are instructed to immediately ramp up to line rate.

(b) When a new flowlet arrives, the allocator re-assigns rates according to policy.

Figure 1-5: Example of allocation in Flowtune.

**NED.** To converge quickly to the allocation that maximizes the specified utility, we introduce a new method, termed *Newton-Exact-Diagonal* (NED). Iterative solvers to NUM make incremental adjustments to rates (Figure 1-6). NED leverages the explicit knowledge of the operator’s utility functions to adjust how aggressive each iteration’s updates should be (Figure 1-7, §3.1).

Flowtune can achieve a variety of desirable objectives. In this work, we focus on proportional fairness, i.e., \( \max \sum U(x_i) \), 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.\(^3\)

**A responsive architecture.** Traditionally, an optimizer would run iterations until the solution converges before setting rates (Figure 1-8a), causing large delays in rate updates when flowlets start or end. To increase system responsiveness, the allocator continuously updates its set of active flowlets and publishes intermediate rates between iterations of the optimizer (Figure 1-8b). These intermediate rates may temporarily exceed the capacity of some links, causing queuing delays. To reduce queuing, Flowtune uses a *rate normalizer* (F-NORM, §3.2) to scale-down the computed values (Figure 1-8c).

\(^3\)Under some requirements of utility functions, discussed in §3.1.
Figure 1-6: Iterative solvers to NUM adjust rates via prices. Too small adjustment result in slow convergence; too aggressive updates result in oscillation and failure to converge.

Figure 1-7: NED uses the slope of the explicit utility functions to choose the size of incremental updates, similar to the Newton method, requiring fewer updates to converge.

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 1-9 shows the system components and their interactions.

Flowtune does not select flow paths, but rather works given the paths the network selects for each flow (§3.5).

**Cache-aware multicore architecture.** 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 (§3.3). On 40 Gbits/s links,
(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 1-8: Motivation for the allocator architecture.

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 (§3.4.2).

**Implementation and Evaluation.** 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 [17] 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 [56], 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.
Figure 1-9: 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.

Figure 1-10: Responsibilities for networking activities in the traditional, SDN, Flowtune and Fastpass architectures.

1.3 The bigger picture

Figure 1-10 compares Flowtune and Fastpass to the two major network architectures: traditional networking and Software Defined Networks (SDN). In traditional networks, the endpoint chooses when and how many packets to send (flow control and congestion control), and switches choose the routes packet take, decide how to schedule queued packets, and perform the actual packet forwarding.
SDN centralizes the control plane: the choice of routes is delegated to a centralized controller. The controller makes decisions when flows start or in some proposals at coarse granularity periodically (every few minutes or seconds).

Flowtune centralizes endpoint congestion control and switch scheduling and queue management, giving the allocator control over resource allocation with data-plane granularity. Control over routing is left to other mechanisms, whether distributed or centralized.

Fastpass further centralizes routing, and provides packet-granularity scheduling and routing control, further extracting functionality from switches.

**Switches can become much simpler.** With Fastpass and Flowtune, switches need to implement less functionality, so switches for these architectures can be simpler and cheaper. Further, with simplified functionality, it should be easier to scale switches to the next generations of link speeds.

**Better developer productivity.** In the Flowtune and Fastpass architectures, developers need not worry about packet drops, high tail latency, and hotspots delaying traffic, and in Fastpass, do not need to avoid bursts. These stronger network-provided semantics simplify application building and debugging, improving developer productivity.

**Better datacenter operations.** Flowtune and Fastpass give operators tools to better balance the needs of the different applications and ensure applications’ peaceful co-existence. Further, networks can run at higher utilizations without excessive queueing and packet loss, which reduces the cost/performance ratio.

**Guiding principal: explicit policy.** Both architectures presented in this work let operators explicitly specify policy of how network resources should be allocated. We believe explicit policy will help application developers and network engineers reason about and enforce performance goals, thus bridge the communication chasm between the two disciplines. We hope the explicit policy enforcement proposed by this work will help service providers run their datacenter networks – and improve online service quality for all.

### 1.4 Contributions

This thesis makes the following contributions.
1.4.1 Fastpass

1. A timeslot-based networking architecture for datacenters, with per-timeslot control of allowed sender-receiver pairs and packet paths.

2. A method to use maximal matching to achieve operator objectives by re-ordering input requests. An O(1) insertion and pop data structure for re-ordering requests. Algorithms to achieve SRTF and max-min fair allocations using re-ordering. Fast implementations of the re-ordering, maximal matching, SRTF, and max-min fair algorithms on CPU.

3. A proof that bounds the maximal-matching average latency by 2× that of the optimal allocation’s (with the standard 2x speedup common in such proofs). Best previous result only shows that maximal-matching latencies are bounded if optimal latencies are bounded.

4. A technique to scale timeslot allocation by arranging timeslot allocation cores into a pipeline and passing requests between cores in batches to reduce overhead. A multicore architecture which allows scaling communication, timeslot allocation, and path selection independently (comm-, alloc- and pathsel-cores), and a specification of how the core types communicate. A C++ implementation using DPDK cores and rings.

5. Adaptation of a circuit-switching technique to allocate paths given allocated matchings [36] to packet-switched networks, where updates are expected to be 4-5 orders of magnitude more frequent. A fast implementation of edge-coloring for path selection on CPUs.

6. A reliable, low-latency protocol for requests and allocations using idempotent updates (the Fastpass Control Protocol, FCP). A low-overhead implementation of FCP.

7. A hot backup failover technique for fault tolerance in case of Arbiter failure, and its implementation.

8. Evaluation using microbenchmarks, synthetic experiments in a datacenter, and Facebook production traffic.

1.4.2 Flowtune

1. Flowlet-based allocation, where the network allocates resources to application-layer transmission batches (flowlets), and an architecture for centrally assigning rates to flowlets.

2. The Newton-Exact-Diagonal (NED) algorithm for solving the Network Utility Maximization (NUM) problem with fast convergence times.

3. An architecture that allows the network allocator to output rates before full convergence while avoiding queueing and packet loss by introducing a normalizer. Two normalizer algorithms: U-NORM and F-NORM, their implementation and evaluation.

4. A multicore algorithm required to efficiently move between computation on flows to computation on links on modern CPUs, replacing random-access memory patterns with sequential scans. Fast multicore implementations of NED and F-NORM using this algorithm.

5. A Linux kernel implementation of flowlet scheduling and an idempotent protocol on top of FCP. A C++ controller using DPDK.

6. Evaluation using ns-2 simulation, microbenchmarks, numeric simulation, and a deployment on Amazon EC2.
Chapter 2

Fastpass: Packet-based Allocation

2.1 Fastpass Architecture

In Fastpass, a logically centralized arbiter controls all network transfers (Fig. 1-3). When an application calls send() or sendto() on a socket, the operating system sends this demand in a request message to the Fastpass arbiter, specifying the destination and the number of bytes. The arbiter processes each request, performing two functions:

1. **Timeslot allocation**: Assign the requester a set of timeslots in which to transmit this data.

   The granularity of a timeslot is the time taken to transmit a single MTU-sized packet over the fastest link connecting an endpoint to the network. The arbiter keeps track of the source-destination pairs assigned each timeslot (§2.2).

2. **Path selection**: The arbiter also chooses a path through the network for each packet and communicates this information to the requesting source (§2.4).

   Because the arbiter knows about all current and scheduled transfers, it can choose timeslots and paths that yield the “zero-queue” property: the arbiter arranges for each packet to arrive at a switch on the path just as the next link to the destination becomes available.

   The arbiter must achieve high throughput and low latency for both these functions; a single arbiter must be able to allocate traffic for a network with thousands of endpoints within a few timeslots.

   Endpoints communicate with the arbiter using the Fastpass Control Protocol (FCP) (§2.5.3). FCP is a reliable protocol that conveys the demands of a sending endpoint to the arbiter and the
allocated timeslot and paths back to the sender. FCP must balance conflicting requirements: it must consume only a small fraction of network bandwidth, achieve low latency, and handle packet drops and arbiter failure without interrupting endpoint communication. FCP provides reliability using timeouts and ACKs of aggregate demands and allocations.

Endpoint applications can send many megabytes with one send() socket call, and can make many send()s in quick succession. Endpoints aggregate allocation demands over a few microseconds into each request packet sent to the arbiter. This aggregation reduces the overhead of requests, and limits queuing at the arbiter.

Fastpass can recover from faults with little disruption to the network (§2.5). Because switch buffer occupancy is small, packet loss is rare and can be used as an indication of component failure. Endpoints report packet losses to the arbiter, which uses these reports to isolate faulty links or switches and compute fault-free paths. The arbiter itself maintains only soft state and endpoints retransmit aggressively if the arbiter does not respond (10s of microseconds), so that a secondary arbiter can take over within a few milliseconds if the primary arbiter fails.

To achieve the ideal of zero queueing, Fastpass requires precise timing across the endpoints and switches in the network. When endpoint transmissions occur outside their allocated timeslots, packets from multiple allocated timeslots might arrive at a switch at the same time, resulting in queueing. Queueing is linear in the clock discrepancy and is small in practice (see §2.6.3, §2.7.1)

Fastpass requires no switch modifications, nor the use of any advanced switch features. Endpoints require some hardware support in NICs that is currently available in commodity hardware (§2.6.3), with protocol support in the operating system. Arbiters can be ordinary server-class machines, but to handle very large clusters, a number of high-speed ports would be required.

2.1.1 Latency experienced by packets in Fastpass

In an ideal version of Fastpass, endpoints receive allocations as soon as they request them: the latency of communication with the arbiter and the time to compute timeslots and paths would be zero. In this ideal case, the end-to-end latency of a packet transmission would be the time until the allocated timeslot plus the time needed for the packet to traverse the path to the receiver with empty queues at all egress ports.
In moderately-loaded to heavily-loaded networks, packets are delayed at their endpoints while previously-allocated packets are transmitted. This causes the ideal allocations mentioned above to typically be several timeslots in the future. As long as the Fastpass arbiter returns results in less than the several timeslots, Fastpass would achieve the ideal minimum packet latency in practice too.

In lightly-loaded networks, Fastpass trades off a slight degradation in the mean packet latency (due to communication with the arbiter) for a significant reduction in the tail packet latency.

### 2.1.2 Deploying Fastpass

Fastpass is deployable incrementally in a datacenter network. Communication to endpoints outside the Fastpass boundary (e.g., to hosts in a non-Fastpass subnet or on the Internet) uses Fastpass to reach the boundary, and is then carried by the external network. Incoming traffic either passes through gateways or travels in a lower priority class. Gateways receive packets from outside the boundary, and use Fastpass to send them within the boundary. Alternatively, incoming packets may use a lower-priority class to avoid inflating network queues for Fastpass traffic.

This chapter focuses on deployments where a single arbiter is responsible for all traffic within the deployment boundary. We discuss larger deployments in §2.8.1.

### 2.2 Timeslot Allocation

The goal of the arbiter’s timeslot allocation algorithm is to choose a matching of endpoints in each timeslot, i.e., a set of sender-receiver endpoint pairs that can communicate in a timeslot. For a simpler exposition, we assume here that all endpoint links run at the same rate. The demand for any given link in the network can exceed its capacity; the arbiter selects sender-receiver pairs and assigns a path to packets (described in §2.4) to ensure that traffic issued in a given timeslot will not exceed any link’s bandwidth.

Networks are often organized into tiers, with each tier providing network transport to components below it: top-of-rack switches connect servers in a rack, aggregation switches connect racks into clusters, core routers connect clusters (Figure 1-2). Fastpass requires that tiers be rearrangeably non blocking (RNB) [23], networks where any traffic that satisfies the input and output bandwidth constraints of the network can be routed such that no queuing occurs.
The RNB property allows the arbiter to perform timeslot allocation separately from path selection: by definition, as long as the allocated matching satisfies the bandwidth constraints in and out of each tier, path selection is guaranteed to successfully assign paths on the physical topology. Consequently, each tier can be abstracted as a single switch for the purpose of timeslot allocation.\footnote{A switch where port capacity reflects the aggregate bandwidth in and out of the tier to that component.} The result is a tree topology on which it is easy for timeslot allocation to check bandwidth constraints, even when the physical network is oversubscribed and has many paths between endpoints. Non-oversubscribed (full-bisection bandwidth) topologies \cite{2,29,53,76} can be abstracted further: we can view the entire network as a single switch.

Because the arbiter has knowledge of all endpoint demands, it can allocate traffic according to global policies that would be harder to enforce in a distributed setting. For instance, the arbiter can allocate timeslots to achieve max-min fairness, to minimize flow completion time, or to limit the aggregate throughput of certain classes of traffic. When conditions change, the network does not need to converge to a good allocation – the arbiter can change the allocation from one timeslot to the next. As a result, the policy (e.g., fairness) can be achieved even over short time scales.

How fast must a viable allocation algorithm be? At first glance, endpoint link speeds determine the allowed allocator runtime, since the arbiter’s processing rate must match endpoint link speed. On 10 Gbits/s links with 1500-byte timeslots, a timeslot is 1.2 μs; the arbiter must allocate a timeslot every 1.2 μs, leaving very few compute cycles for allocating each timeslot. However, this can be improved by allocating multiple timeslots in parallel: say 10 cores allocate 10 timeslots in parallel. If each core takes 12 μs to compute its allocation ($10 \times$ more cycles), in aggregate the arbiter can still finish one allocation per 1.2 μs, maintaining the required processing rate.

A long runtime per timeslot (compared to the minimum RTT between the endpoints) is acceptable with some workloads, but not others: on heavily-loaded endpoints, the time until the first available timeslot can be tens to hundreds of microseconds, so traffic will observe the ideal end-to-end latency (§2.1.1), even if allocation takes many microseconds. On the other hand, traffic on lightly-loaded networks doesn’t enjoy this masking of allocation latency; the algorithm must finish promptly if a small end-to-end latency increase is desired.

Complete knowledge of all network demands thus becomes a double-edged sword; in order to meet these latency and throughput requirements, the timeslot allocator requires very fast algorithms.
Finding an allocation with the largest possible number of pairs (a maximum matching) is expensive; switch algorithms (e.g., [69, 9, 47]) generally use heuristics to find good, but not maximum, matchings. Fastpass uses a similar approach: as the arbiter processes demands, it greedily allocates a source-destination pair if allocating the pair does not violate bandwidth constraints. When the arbiter finishes processing all demands, it has a maximal matching, a matching in which none of the unallocated demands can be allocated while maintaining the bandwidth constraints.

### 2.2.1 A pipelined allocator

The allocator takes a list of all network demands (how many timeslots are waiting to be sent between each pair of endpoints), and computes the allocated matching and the remaining demands after the allocation. Figure 2-1 shows how Fastpass allocators are arranged into a pipeline: the input to the allocator processing timeslot $t$ is the remaining demand after allocating timeslot $t-1$. Cores do not wait to process all input demands before outputting remaining demands. Instead, cores output each remaining demand soon after processing it. Demands stream between the running cores, allowing multiple cores to perform useful work concurrently.

The arbiter implements different allocation policies by changing the order in which demands are processed. For max-min fairness, the arbiter orders demands by the last timeslot that was allocated to the source-destination pair, “least recently allocated first”; for minimizing flow completion time

---

2In a non-oversubscribed network, the arbiter checks that neither the source nor the destination have already been allocated to a different pair. Oversubscribed topologies require the arbiter to additionally check bandwidth constraints in and out of each network tier.
Figure 2-2 demonstrates the allocation of one timeslot in a simple network with four endpoints. The allocator orders the demands by the last timeslot allocated to each pair, and processes them in that order. On the right is the state used to track bandwidth constraints: one bit for each source and for each destination. The first two demands can be allocated because both the source and destination are available, but the third demand cannot be allocated because destination 3 has already been allocated. The remaining two demands can be allocated, yielding a maximal matching.

Each allocator in the pipeline receives a stream of demands. Ideally, an allocator could process each demand as soon as it is produced by the previous allocator. If demands can appear out of the desired order, however, the allocator must reorder them first. In a worst-case scenario, the last demand from the previous allocator should be processed first. The allocator would have to wait until the previous allocator produced all demands in the stream before it could start processing, eliminating the concurrency gained by pipelining.

Fortunately, with both max-min fairness and min-FCT (and other such objectives), demands can be kept in roughly the correct order with only limited reordering. For example, in max-min fairness, the allocator of timeslot $t$ only changes the last allocated timeslot of a source-destination pair if that pair is allocated, and will only change it to $t$. Therefore, an allocator for timeslot $t + 1$ can process all demands with last allocated timeslot strictly less than $t$ immediately. Only demands with last allocated timeslot equal to $t$ need to be kept until all demands are received.
To reduce the overhead of processing demands, the allocator allocates a batch of 8 timeslots in one shot using a data structure, the bitmap table. This table maintains a bitmap for each sender-receiver pair in the network, with one bit per timeslot. A “1” in the bitmap signifies that the pair is not scheduled to communicate in that timeslot, while a “0” indicates otherwise. To find the first available timeslot for a given packet, the allocator computes the bitwise AND of the source and destination bitmaps, and then uses the “find first set” operation (the `bsf` instruction on x86). Modern processors perform this operation quickly [38]. Pairs that have been allocated and have remaining demand are kept, and the arbiter will attempt to allocate more batch timeslots to them; pairs that cannot be allocated in this batch are sent to the next allocator in the pipeline.

2.3 Theoretical Properties of the Timeslot Allocator

The timeslot allocator uses a heuristic to find matchings. Will a network scheduled with these matchings provide good service to its endpoints? Faced with a workload that does not oversubscribe the network’s inputs and outputs, a good allocator will not let endpoint source-destination queues grow too much, i.e., it will have small average queueing latency.

In this section, we prove that in those topologies where timeslot allocation and path selection can be separated, the average latency with Fastpass is no worse than $2 \times$ the average latency of an optimal scheduler with half as much network capacity on its worst-case workload. It follows that the worst-case Fastpass behavior is no worse than $2 \times$ the optimal scheduler’s with half the capacity. The result upper-bounds the latency cost of using Fastpass over any other solution.

Note that the requirement for Fastpass to have twice the network capacity as the optimal scheduler is a side effect of the proof method we used. The bound, however, is not tight: Fastpass achieves low latency even at much higher network loads (§2.7).

System model. We consider a network with $N$ endpoints denoted by $1, \ldots, N$, with each endpoint potentially having requests for any other endpoint. Time is slotted; in a unit timeslot each endpoint can transmit at most one MTU-sized packet to any other endpoint and receive at most one packet from any other endpoint. New requests arrive at an endpoint $i$ for endpoint $j$ with probability $p_{ij}$ in each timeslot. An arriving request brings a random number of packets, distributed according to a distribution $G_{ij}$ and independent of everything else.
Let $E[G_{ij}] = g_{ij}$. Thus, on average $\lambda_{ij} = p_{ij}g_{ij}$ packets arrive at endpoint $i$ for endpoint $j$ per timeslot. The matrix $\Lambda = [\lambda_{ij}]$ denotes the net average data traffic between endpoints arriving in each timeslot. We assume a non-oversubscribed (also called full bisection bandwidth) network: the network can support any traffic where each node is paired to at most one other node per timeslot.

In the above setup, all traffic matrices, $\Lambda$, that can be served by any system architecture, must be doubly sub-stochastic. That is,

$$\sum_{k=1}^{N} \lambda_{ik} < 1, \text{ for all } i \text{ and } \sum_{k=1}^{N} \lambda_{kj} < 1, \text{ for all } j.$$  \hfill (2.1)

By the celebrated result of Birkhoff and von Neumann, all doubly stochastic matrices can be decomposed as a weighted sum of permutation matrices (i.e., matchings) with the sum of the weights being at most 1. Therefore, non-oversubscribed networks can support all doubly stochastic traffic matrices. A traffic matrix $\Lambda$ is called admissible if and only if $\rho(\Lambda) < 1$ where, the system load $\rho(\Lambda)$ is defined as

$$\rho(\Lambda) = \max_{i,j} \left( \sum_{k=1}^{N} \lambda_{ik}, \sum_{k=1}^{N} \lambda_{kj} \right).$$  \hfill (2.2)

Finally, let $Q_{ij}(t)$ denote the total number of packets (potentially across different requests) waiting to be transferred from endpoint $i$ to endpoint $j$ at time $t$. This setup is similar to that used in literature on input-queued switches [48], enabling us to view the network as a big input-queued switch with $Q_{ij}(t)$ the Virtual Output Queue sizes.

**Main result.** The arbiter’s timeslot allocation algorithm of §2.2 is equivalent to the following: each queue $(i, j)$ has a “priority score” associated with it. In the beginning of each timeslot, the arbiter starts processing queues in non-decreasing order of these priority scores. While processing, the arbiter allocates a timeslot to queue $(i, j)$ if the arbiter has not already allocated another packet starting from $i$ or destined to $j$ in this timeslot. Therefore, at the end of processing the timeslot, the allocations correspond to a maximal matching between endpoints in the bipartite graph between endpoints, where an edge is present between $(i, j)$ if there are packets waiting at endpoint $i$ destined for $j$. From the literature on input-queued switches, it is well-known that any maximal matching...
provides 50% throughput guarantees [19, 1]. Building upon these results as well as [58], we state the following property of our algorithm.

**Theorem 1** For any \( \rho < 1 \), there exists \( \Lambda \) with \( \rho(\Lambda) = \rho \) such that for any allocator,

\[
\liminf_{t} \mathbb{E}\left[\sum_{ij} Q_{ij}(t)\right] \geq \frac{N \rho}{2(1-\rho)}.
\]  

(2.3)

Further, let \( V \geq 1 \) be such that \( \mathbb{E}[G_{ij}^2] \leq V \mathbb{E}[G_{ij}] \) for all \( i, j \) (bounded \( G_{ij} \)); if we allow the Fastpass arbiter to schedule (as well as transmit through the network) twice per unit timeslot,\(^3\) then the induced average queue-size

\[
\limsup_{t} \mathbb{E}\left[\sum_{ij} Q_{ij}(t)\right] \leq \frac{N \rho (\rho + V)}{2(1-\rho)}.
\]  

(2.4)

**Proof Sketch.** To establish the lower bound (2.3) for any scheduling algorithm, it is sufficient to consider a specific scenario of our setup. Concretely, let the traffic matrix be uniform, i.e., \( \Lambda = [\lambda_{ij}] \) with \( \lambda_{ij} = \frac{\rho}{(N-1)} \) for all \( i \neq j \) and 0 when \( i = j \); \( p_{ij} = 1 \) for all \( i \neq j \); and let \( G_{ij} \) be Poisson variables with parameter \( \lambda_{ij} \). The network can be viewed as unit-sized packets arriving at each endpoint according to a Poisson arrival process of rate \( \rho \) and processed (transferred by the network) at unit rate. That is, the queue-size for each endpoint \( j \) is bounded below by that of an \( M/D/1 \) queue with load \( \rho \), which is known to be bounded below by \( \frac{\rho}{(2(1-\rho))} \) \([59]\). Therefore, the network-wide queue-size is bounded below by \( \frac{N \rho}{(2(1-\rho))} \).

To establish an upper bound, we use the fact that the algorithm effectively achieves a maximal matching in the weighted bipartite graph between endpoints in each timeslot. Given this fact, and under the assumption that Fastpass can schedule as well as transfer data at twice the speed, this is effectively a *speedup* of 2 in the classical terminology of switch scheduling. Therefore, for the Lyapunov function (cf. [19]),

\[
L(t) = \sum_{i,j} Q_{ij}(t)[Q_i(t) + Q_j(t)]
\]

\(^3\)Equivalent to having to double the network fabric bandwidth.
it can be shown using calculations similar to [58] that

$$\mathbb{E}[L(t+1) - L(t)|Q(t)] \leq 4(\rho - 1)(\sum_{i,j} Q_{ij}(t)) + 2Np^2 + 2VNp.$$ 

Telescoping this inequality for $t \geq 0$ and using the fact that the system reaches equilibrium due to ergodicity, we obtain the desired result. ■

**Implications.** Equation (2.3) says that there is some (worst case) input workload for which any allocator will have an expected aggregate queue length at least as large as $\frac{Np}{2(1-\rho)}$. Equation (2.4) says that with a speedup of 2 in the network fabric, for every workload, the expected aggregate queue length will be no larger than $\frac{Np(\rho + V)}{2(1-\rho)}$. Here $V$ is effectively a bound on burst size; if it is small, say 1, then it is within a factor of 2 of the lower bound! There is, however, a gap between theory and practice here, as in switch scheduling; many workloads observed in practice seem to require only small queues even with no speedup.

2.4 Path Selection

Path selection assigns packets that have been allocated timeslots to paths through the network that avoid queueing. Common datacenter topologies (e.g., multi-rooted trees) include redundant paths between endpoints. If the timeslot allocator admits traffic that utilizes the full network bandwidth, and more packets attempt to traverse a link than that link can sustain, queueing will result. To utilize the full network bandwidth without queueing, path selection must balance packet load across all available links.

Existing approaches for load balancing all have significant disadvantages. Equal-cost multi-path (ECMP, RFC 2992) routing can result in multiple flows hashing onto the same path, causing significant skew over short time scales. Hedera [3] is able to re-route “elephant” flows for better load balance, but focuses only on such flows; the load generated by small flows at finer scales is left unbalanced, and that could be substantial in practice.

The goal of path selection is to assign packets to paths such that no link is assigned multiple packets in a single timeslot; this property guarantees that there will be no queueing within the network. Timeslot allocation guarantees that this property holds for the links directly connected to
Figure 2-3: Path selection. (a) input matching (b) ToR graph (c) edge-colored ToR graph (d) edge-colored matching.

Each endpoint; path selection must provide this guarantee for the remainder of the network. In a network with two tiers, ToR and core, with each ToR connected directly to a subset of core switches (Fig. 1-2) each path between two ToRs can be uniquely specified by a core switch. Thus path selection entails assigning a core switch to each packet such that no two packets (all of which are to be sent in the same timeslot) with the same source ToR or destination ToR are assigned the same core switch.

**Edge-coloring.** This assignment can be performed with graph edge-coloring [36]. The edge-coloring algorithm takes as input a bipartite graph and assigns a color to each edge such that no two edges of the same color are incident on the same vertex. We model the network as a bipartite graph where the vertices are ToR switches, edges are the allocated packets, and colors represent core switches/paths. The edge-coloring of this graph provides an assignment of packets to paths such that no link is assigned multiple packets.

Figure 2-3 shows an example. The matching of packets to be transmitted in a given timeslot (a) is transformed into a bipartite multigraph of ToRs (b), where the source and destination ToRs of every packet are connected. Edge-coloring colors each edge ensuring that no two edges of the same color are incident on the same ToR (c). The assignment guarantees that at most one packet occupies the ingress, and one occupies the egress, of each port (d).

Edge-coloring requires uniform link capacities; in networks with heterogeneous link capacities, we can construct a virtual network with homogeneous link capacities on which to assign paths. Here, we replace each physical switch with high-capacity links with multiple switches with low capacity links that connect to the same components as the physical switch (e.g., one switch with
40 Gbits/s links would be replaced by four switches with 10 Gbits/s links). All packets assigned a path through the duplicate switches in the virtual topology would be routed through the single high-capacity switch on the physical topology.

Edge-coloring also generalizes to oversubscribed networks and networks with multiple tiers. Only traffic that passes through a higher network tier is edge-colored (e.g., in a two-tier network, only inter-rack traffic requires path selection). For a three-tier datacenter with ToR, Agg, and Core switches (and higher-tier ones), paths can be assigned hierarchically: the edge-coloring of the ToR graph assigns Agg switches to packets, then an edge-coloring of the Agg graph chooses Core switches [36, §IV].

**Fast edge-coloring.** A network with \( n \) racks and \( d \) nodes per rack can be edge-colored in \( O(nd \log d) \) time [18, 42]. Fast edge-coloring algorithms invariably use a simple and powerful building block, the *Euler-split*. An Euler-split partitions the edges of a regular graph where each node has the same degree, \( 2d \), into two regular graphs of degree \( d \). The algorithm is simple: (1) find an Eulerian cycle (a cycle that starts and ends at the same node, and contains every edge exactly once, though nodes may repeat) of the original graph, (2) assign alternate edges of the cycle to the two new graphs, (3) repeat.

An Euler-split divides the edges into two groups that can be colored separately. \( d - 1 \) Euler-splits can edge-color a graph with power-of-two degree \( d \) by partitioning it into \( d \) perfect matchings, which can each be assigned a different color. Graphs with non-power-of-two degree can be edge-colored using a similar method that incorporates one search for a perfect matching, and has only slightly worse asymptotic complexity [42].

The Fastpass path selection implementation maintains the bipartite graph in a size-optimized bitmap-based data structure that can fit entirely in a 32 KB L1 cache for graphs with up to 6,000 nodes. Limiting all accesses to L1 cache makes the graph walks needed for Euler-split fast, and enables a small number of cores to handle clusters with a few thousand nodes, even with microsecond-scale timeslots (see §2.7.3).
2.5 Handling Faults

Three classes of faults can render a Fastpass network ineffective: failures of in-network components (nodes, switches and links), failures of the Fastpass arbiter, and packet losses on the communication path between endpoints and the arbiter.

2.5.1 Arbiter failures

Fastpass runs multiple arbiters, one primary and a few secondaries. The arbiters all subscribe to a pre-determined IP multicast destination address to which all requests from endpoints are sent (responses are unicast to the endpoint). All the arbiters receive all requests (modulo packet loss), but only the designated primary responds to all requests; the secondaries drop the requests and generate no allocations.

The secondaries detect the failure of the primary as follows. The primary sends out tiny watchdog packets on the multicast group every $T_w$ microseconds. If a secondary receives no watchdog packets during an interval of time $T_d$, that secondary assumes that the primary has failed, and starts responding to client requests. If there is more than one secondary, a pre-defined rank order determines the order in which secondaries attempt to take over as the primary. Our current implementation uses only one secondary.

In practice, arbiters can be aggressive about detecting and reacting to failure, allowing recovery within 1–2 ms. An implementation can use $T_w = 100$ microseconds and $T_d = 1$ millisecond to achieve fast failover, consuming under 10 Mbits/s.

Upon an arbiter failover, the new arbiter must obtain an updated snapshot of all endpoint demands, so it can start allocating timeslots. The Fastpass Control Protocol (described below) assumes only soft-state at the arbiter, allowing endpoints to re-synchronize with the new arbiter in 1–2 round-trip times.

The transition period between old and new arbiters needs careful handling to prevent persistent queues from building up. For example, if the old arbiter tells $A$ to send to $C$ at some timeslot, and the new arbiter tells $B$ to send to $C$ at the same time, a queue will form at $C$'s ingress until encountering a free timeslot.
Conservative timing can ensure that by the time an arbiter failure is detected and the secondary takes over, no allocations made by the failed arbiter remain in the network. In our implementation, the arbiter allocates each timeslot 65 μs before the timeslot’s start time to leave enough time for notifications to reach the endpoints; this number is much smaller than the 1 millisecond before the secondary takes over.

As a result, the old and new arbiters do not have to share information to make the failover possible, simplifying the implementation: the new arbiter just resynchronizes with endpoints and starts allocating.

2.5.2 Network failures

Arbiters need to know the network topology under their control, so packets can successfully traverse the network in their allotted timeslots. If links or switches become unavailable undetected, the arbiter would continue to allocate traffic through them, causing packet loss.

Because Fastpass maintains low queue occupancies, it can use packet drops to detect network faults. Sporadic packet losses due to bit flips in the physical layer still occur, but persistent, correlated packet loss is almost surely due to component failure. The path each packet traverses is known to the endpoints, which helps localize the fault.

A network service performs fault isolation, separate from the arbiter. Endpoints report packet drops to this fault isolation service, which correlates reports from multiple endpoints to identify the malfunctioning component. The faulty component is then reported to the arbiter, which avoids scheduling traffic through the component until the fault is cleared. Since packet errors can be detected quickly, failure detection and mitigation times can be made extremely short, on the order of milliseconds.

2.5.3 Fastpass Control Protocol (FCP)

Communication between endpoints and the arbiter is not scheduled and can experience packet loss. FCP must protect against such loss. Otherwise, if an endpoint request or the arbiter’s response is dropped, a corresponding timeslot would never be allocated, and some packets would remain stuck in the endpoint’s queue.
TCP-style cumulative ACKs and retransmissions are not ideal for FCP. At the time of retransmission, the old packet is out of date: for a request, the queue in the endpoint might be fuller than it was; for an allocation, an allocated timeslot might have already passed.

FCP provides reliability by communicating aggregate counts; to inform the arbiter of timeslots that need scheduling, the endpoint sends the sum total of timeslots it has requested so far for that destination. The arbiter keeps a record of each endpoints’ latest demands; the difference from the kept record and the new aggregate demands specifies the amount of new timeslots to be allocated.

The counts are idempotent: receiving the same count multiple times does not cause timeslots to be allocated multiple times. Idempotency permits aggressive timeouts, leading to low allocation latency even in the face of occasional packet loss. Endpoints detect the loss of allocated timeslots using aggregate counts sent by the arbiter, triggering a request for more timeslots.

Handling arbiter failure. When a secondary arbiter replaces a failed primary, its aggregate counts are out of sync with the endpoints. The mismatch is detected using a random nonce established when the arbiter and endpoint synchronize. The nonce is incorporated into the packet checksum, so when an arbiter is out of sync with an endpoint, it observes a burst of failed checksums from an endpoint that triggers a re-synchronization with that endpoint.

Upon re-synchronization, the arbiter resets its aggregate counts to zeros, and the endpoint recomputes fresh aggregate counts based on its current queues. The process takes one round-trip time: as soon as the endpoint processes the RESET packet from the arbiter, it can successfully issue new requests to the arbiter.

2.6 Implementation

We implemented the arbiter using Intel DPDK [37], a framework that allows direct access to NIC queues from user space. On the endpoints, we implemented a Linux kernel module that queues packets while requests are sent to the arbiter. Our source code is at http://fastpass.mit.edu/
### 2.6.1 Client

FCP is implemented as a Linux transport protocol (over IP). A Fastpass qdisc (queueing discipline) queues outgoing packets before sending them to the NIC driver, and uses an FCP socket to send demands to and receive allocations from the arbiter.

The Fastpass qdisc intercepts each packet just before it is passed to the NIC, and extracts the destination address from its packet header.\(^4\) It does not process transport protocol headers (e.g., TCP or UDP).

The Fastpass arbiter schedules network resources, obviating the need for TCP congestion control. TCP’s congestion window (cwnd) could needlessly limit flow throughput, but packet loss is rare, so cwnd keeps increasing until it no longer limits throughput. At this point, TCP congestion control is effectively turned off.

The current implementation does not modify TCP. The evaluated system maintains long-lived flows, so flows are not cwnd-limited, and we preferred not to deal with the complexities of TCP. Nevertheless, modifying TCP’s congestion control would be worthwhile for improving short-lived TCP flow performance.

The client is not limited to sending exactly one packet per timeslot. Instead, it greedily sends packets while their total transmission time is less than the timeslot length. This aggregation reduces the amount of potentially wasted network bandwidth caused by many small packets destined to the same endpoint.

### 2.6.2 Multicore Arbiter

The arbiter is made up of three types of cores: comm-cores communicate with endpoints, alloc-cores perform timeslot allocation, and pathsel-cores assign paths.

The number of cores of each type can be increased to handle large workloads: each comm-core handles a subset of endpoints, so endpoints can be divided among more cores when protocol handling becomes a bottleneck; alloc-cores work concurrently using pipeline parallelism (§2.2); and pathsel-cores are “embarrassingly parallel”, since path assignments for different timeslots are independent.

\(^4\)On a switched network, MAC addresses could be used. However, in the presence of routing, IP addresses are required.
Figure 2-4: Multicore allocation: (1) allocation cores assign packets to timeslots, (2) path selection cores assign paths, and (3) communication cores send allocations to endpoints.

Fig. 2-4 shows communication between arbiter cores. Comm-cores receive endpoint demands and pass them to alloc-cores (not shown). Once a timeslot is completely allocated, it is promptly passed to a pathsel-core. The assigned paths are handed to comm-cores, which notify each endpoint of its allocations.

Performance measurements of each core type are presented in §2.7.

2.6.3 Timing

To keep queue occupancy low, end-node transmissions should occur at the times prescribed by the arbiter. Otherwise, packets from multiple endpoints might arrive at a switch’s egress port together, resulting in queueing.

The amount of queueing caused by time-jitter is determined by the discrepancy in clocks. For example, if all clocks are either accurate or one timeslot fast, at most two packets will arrive at any egress: one from an accurate node, the other from a fast node.

In general, say the clock discrepancy is bounded by $T$ timeslots, at any given time a switch queue can only contain packets from at most $T$ sources – those that were scheduled to send through the queue in those $T$ timeslots.

Clock synchronization. The deployment synchronizes end-node time using the IEEE1588 Precision Time Protocol (PTP), which achieves sub-microsecond clock synchronization by carefully mitigating causes of time-synchronization jitter. PTP-capable NICs timestamp synchronization
packets in hardware [55], thus avoiding jitter due to operating system scheduling. NICs with PTP support are widely available; the experiment used Intel 82599EB NICs.

Variable queueing delays inside the network also cause synchronization inaccuracies, and PTP-supported switches report their queueing delays so endpoints can compensate. However, Fastpass keeps queue-length variability low, enabling high quality time synchronization without PTP switch support.

**Client timing.** The client uses Linux high-resolution timers (hrtimers), previously demonstrated to achieve microsecond-scale precision when shaping flow throughput [68].

The client uses locks to synchronize access to the qdisc queues and to allocation state. Because waiting for these locks when transmitting packets causes variable delays, we allow the qdisc to send packets after their scheduled times, up to a configurable threshold. Endpoints re-request overly late allocations from the arbiter.

### 2.7 Evaluation

The goal of Fastpass is to simultaneously eliminate in-network queueing, achieve high throughput, and support various inter-flow or inter-application resource allocation objectives in a real-world datacenter network. In this section, we evaluate how well Fastpass meets these goals, compared to a baseline datacenter network running TCP.
Summary of Results

| §2.7.1   | (A) Under a bulk transfer workload involving multiple machines, Fastpass reduces median switch queue length to 18 KB from 4351 KB, with a 1.6% throughput penalty. |
| §2.7.1   | (B) Interactivity: under the same workload, Fastpass’s median ping time is 0.23 ms vs. the baseline’s 3.56 ms, 15.5× lower with Fastpass. |
| §2.7.2   | (C) Fairness: Fastpass reduces standard deviations of per-sender throughput over 1 s intervals by over 5200× for 5 connections. |
| §2.7.3   | (D) Each comm-core supports 130 Gbits/s of network traffic with 1 μs of NIC queueing. |
| §2.7.3   | (E) Arbiter traffic imposes a 0.3% throughput overhead. |
| §2.7.3   | (F) 8 alloc-cores support 2.2 Terabits/s of network traffic. |
| §2.7.3   | (G) 10 pathsel-cores support >5 Terabits/s of network traffic. |
| §2.7.4   | (H) In a real-world latency-sensitive service, Fastpass reduces TCP retransmissions by 2.5×. |

Table 2.1: Fastpass: Summary of Results

Experimental setup. We conducted experiments on a single rack of 32 servers, with each server connected to a top-of-rack (ToR) switch with a main 10 Gbits/s Ethernet (GbE) network interface card (NIC). Servers also have a 1 Gbit/s NIC meant for out-of-band communication. The 10 GbE top-of-rack switch has four 10 GbE uplinks to the four cluster switches [25]. Each server has 2 Intel CPUs with 8 cores each (16 hyper-threads per CPU, for a total of 32 hyper-threads) and 148 GB RAM. One server is set aside for running the arbiter. We turn off TCP segmentation offload (TSO) to achieve better control over the NIC send queues.

2.7.1 Throughput, queueing, and latency

Experiment A: throughput and queueing. Our first experiment compares the throughput and switch queue occupancy of Fastpass to the baseline network. Four rack servers run iperf to
generate traffic (20 TCP flows per sender) to a single receiver. The experiment lasts 20 minutes and is run twice—once with the baseline and once with Fastpass.

Fastpass achieves throughput close to the baseline's: 9.43 Gbits/s in the baseline run versus 9.28 Gbits/s with Fastpass. At the same time, Fastpass reduces the median switch queue occupancy from 4.35 Megabytes in the baseline to just 18 kilobytes with Fastpass, a reduction of a factor of 242× (Fig. 2-5).

It isn't just the median but also the tails of the queue-occupancy distribution that are a lot lower, as shown here:

<table>
<thead>
<tr>
<th></th>
<th>Median</th>
<th>90th %ile</th>
<th>99th</th>
<th>99.9th</th>
</tr>
</thead>
<tbody>
<tr>
<td>Baseline (Kbytes)</td>
<td>4351</td>
<td>5097</td>
<td>5224</td>
<td>5239</td>
</tr>
<tr>
<td>Fastpass (Kbytes)</td>
<td>18</td>
<td>36</td>
<td>53</td>
<td>305</td>
</tr>
</tbody>
</table>

Table 2.2: Distribution of switch queue lengths in KB.

Most of the 1.6% difference in throughput can be attributed to Fastpass reserving about 1% of the achieved throughput for FCP traffic. The remainder is due to the client re-requesting timeslots when packet transmissions were delayed more than the allowed threshold (§2.6.3).

Switch queues are mostly full in the baseline because TCP continues to increase the sending rate until a packet is dropped (usually due to a full queue). In contrast, Fastpass's timeslot allocation keeps queues relatively empty: the 99.9th percentile occupancy was 305 KB over the entire experiment. Although the design intends queues to be strictly 0, the implementation does not yet achieve it because of jitter in endpoint transmission times. We believe that queueing can be further reduced towards the zero ideal by using finer-grained locking in the kernel module.

**Experiment B: latency.** Next, we measure the round-trip latency of interactive requests under high load. This experiment uses the same setup as Experiment A, augmented with a fifth machine that sends a small request to the receiving server every 10 ms, using ping.

Fastpass reduces the end-to-end round-trip time (RTT) for interactive traffic when the network is heavily loaded by a factor of 15.5×, from a median of 3.56 ms to 230 µs (Fig. 2-6). The tail of the distribution observes significant reductions as well:
Figure 2-5: Switch queue lengths sampled at 100ms intervals on the top-of-rack switch. The diagram shows measurements from two different 20 minute experiments: baseline (red) and Fastpass (blue). Baseline TCP tends to fill switch queues, whereas Fastpass keeps queue occupancy low.

<table>
<thead>
<tr>
<th></th>
<th>Median</th>
<th>90th %ile</th>
<th>99th</th>
<th>99.9th</th>
</tr>
</thead>
<tbody>
<tr>
<td>Baseline (ms)</td>
<td>3.56</td>
<td>3.89</td>
<td>3.92</td>
<td>3.95</td>
</tr>
<tr>
<td>Fastpass (ms)</td>
<td>0.23</td>
<td>0.27</td>
<td>0.32</td>
<td>0.38</td>
</tr>
</tbody>
</table>

Table 2.3: Latency comparison, Baseline vs. Fastpass.

Note that with Fastpass ping packets are scheduled in both directions by the arbiter, but even with the added round-trips to the arbiter, end-to-end latency is substantially lower because queues are much smaller. In addition, Fastpass achieves low latency for interactive traffic without requiring the traffic to be designated explicitly as “interactive” or “bulk,” and without using any mechanisms for traffic isolation in the switches: it results from the fairness properties of the timeslot allocator.

### 2.7.2 Fairness and convergence

**Experiment C: fairness and convergence.** Here we examine how fairly Fastpass allocates throughput to multiple senders, and how quickly it converges to a fair allocation when a sender arrives or departs. Five rack servers each send a bulk TCP flow to a sixth receiving server. The experiment begins with one bulk flow; every 30 seconds, a new bulk flow arrives until all five are active for 30 seconds, and then one flow terminates every 30 seconds. The entire experiment therefore lasts 270 seconds.
We calculate each connection’s throughput over 1-second intervals. The resulting time series for the baseline TCP and for Fastpass are shown in Figure 2-7.

The baseline TCPs exhibit much larger variability than Fastpass. For instance, when the second connection starts, its throughput is about 20-25% higher than the first connection throughout the 30-second interval; similarly, when there are two senders remaining between time 210 to 240 seconds, the throughputs “cross over” and are almost never equal. With more connections, the variation in throughput for TCP is more pronounced than with Fastpass.

To quantify this observation, we calculate the standard deviation of the per-connection throughputs achieved in each 1-second interval, in Mbits/s, when there are 3, 4, and 5 concurrent connections each for the baseline TCP and Fastpass. We then compute the median over all standard deviations for a given number of connections (a median over 60 values when there are 3 or 4 connections and over 30 values when there are 5 connections). The results are:

<table>
<thead>
<tr>
<th>#connections</th>
<th>Baseline</th>
<th>Fastpass</th>
<th>Improvement</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>543.86</td>
<td>15.89</td>
<td>34.2\times</td>
</tr>
<tr>
<td>4</td>
<td>628.49</td>
<td>0.146</td>
<td>4304.7\times</td>
</tr>
<tr>
<td>5</td>
<td>459.75</td>
<td>0.087</td>
<td>5284.5\times</td>
</tr>
</tbody>
</table>

Table 2.4: Throughput variability, Baseline vs. Fastpass.
Figure 2-7: Each connection’s throughput, with a varying number of senders. Even with 1s averaging intervals, baseline TCP flows achieve widely varying rates. In contrast, for Fastpass (bottom), with 3, 4, or 5 connections, the throughput curves are on top of one another. The Fastpass max-min fair timeslot allocator maintains fairness at fine granularity. The lower one- and two-sender Fastpass throughput is due to Fastpass qdisc overheads (§2.7.2).

These results show that Fastpass exhibits significantly lower variability across the board: its standard deviation of throughput is over 5200 times lower than the baseline when there are five concurrent connections.

Fastpass’s pipelined timeslot allocation algorithm prioritizes flows based on their last transmission time, so when a new flow starts, it is immediately allocated a timeslot. From that point on, all flows contending for the bottleneck will be allocated timeslots in sequence, yielding immediate convergence and perfect fairness over intervals as small as 1 MTU per flow (for 5 flows on 10 Gbits/s links, this yields fairness at the granularity of 6 μs).

The benchmark shows low total throughput for one and two senders because of packet processing overheads, which are usually reduced by TSO. (In contrast, Experiments A and B use many more connections, so they achieve high utilization). Fastpass senders also require additional processing in the Fastpass qdisc, which is limited to using one core; NIC support (§2.8.3) or a multicore implementation will alleviate this bottleneck.
Figure 2-8: As more requests are handled, the NIC polling rate decreases. The resulting queueing delay can be bounded by distributing request-handling across multiple comm-cores.

### 2.7.3 Arbiter performance

**Experiment D: request queueing.** To estimate how long requests queue at the arbiter before they are processed, we measure the NIC polling rate of the comm-core under increasing amounts of network traffic. Every 10 seconds, a rack server starts a TCP transfer to an unloaded server.

As the load increases, the arbiter spends more time processing requests, the NIC polling rate decreases (Fig. 2-8), and requests are delayed in the arbiter’s NIC queues. A deployment can control this queueing delay by limiting the amount of traffic each comm-core handles: 130 Gbits/s for 1 μs queueing, 170 Gbits/s for 10 μs, etc.

**Experiment E: communication overhead.** To determine the network capacity requirements at the arbiter, we measure the total amount of control traffic the arbiter receives and transmits in experiment D. The network overhead of communication with the arbiter is 1-to-500 for request traffic and 1-to-300 for allocations for the tested workload (Fig. 2-9): to schedule as much as 150 Gbits/s, the comm core receives less than 0.3 Gbits/s of requests and sends out 0.5 Gbits/s of allocations. When the NIC polling rate decreases sufficiently, incoming request packets start getting dropped (seen around 160 Gbits/s). The arbiter is still able to allocate all traffic because FCP retransmissions summarize aggregate demands; hence, not every demand packet needs to be received.
Experiment F: timeslot allocation cores. To determine the number of arbiter cores necessary for timeslot allocation, we measure the throughput of the max-min fair timeslot allocation implementation. Requests are generated by a synthetic stress-test-core, rather than received from a comm-core. The workload has Poisson arrivals, senders and receivers chosen uniformly at random from 256 nodes, and requests are for 10 MTUs. We vary the mean inter-arrival time to produce different network loads.

<table>
<thead>
<tr>
<th></th>
<th>2 cores</th>
<th>4 cores</th>
<th>6 cores</th>
<th>8 cores</th>
</tr>
</thead>
<tbody>
<tr>
<td>Throughput (Gbits/s)</td>
<td>825.6</td>
<td>1545.1</td>
<td>1966.4</td>
<td>2211.8</td>
</tr>
</tbody>
</table>

Table 2.5: Throughput of different alloc-core pipelines lengths.

Eight alloc-cores support over 2.21 Terabits/s of network traffic, or 221 endpoints transmitting at a full 10 Gbits/s. This corresponds to the 256 nodes achieving over 86% network utilization.

Experiment G: path selection cores. To determine the number of arbiter cores needed for path selection, we measure the average processing time per timeslot as load increases. We use the synthetic workload described above with exponentially distributed request sizes (with a mean of 10 MTUs). The chosen topology has 32 machines per rack and four paths between racks, with no over-subscription (motivated in part by the “4-post” cluster topology [25]). Note that non-over-subscribed topologies could be considered worst-case topologies for path selection: over-subscription reduces the amount of traffic leaving each rack and thus simplifies path-selection.
Figure 2-10: Path selection routes traffic from 16 racks of 32 endpoints in <12 μs. Consequently, 10 pathsel-cores would output a routing every <1.2 μs, fast enough to support 10 Gbits/s endpoint links. Error bars show one standard deviation above and below the mean.

Fig. 2-10 shows that the processing time increases with network utilization until many of the nodes reach full degree (32 in the tested topology), at which point the cost of pre-processing\textsuperscript{5} the graph decreases, and path selection runs slightly faster.

Because path-selection can be parallelized by having a different core select paths for each timeslot, these measurements indicate how many pathsel-cores are required for different topologies. For example, path selection of 16 racks can be done in under 12 microseconds; hence, for 1.2 microsecond timeslots, 10 pathsel-cores suffice.

2.7.4 Production experiments at Facebook

Workload. We deployed Fastpass on a latency-sensitive service that is in the response path for user web requests. This service has a partition-aggregate workload similar to common search workloads [12]. Each server runs both an aggregator and a leaf; when an aggregator receives a query from a web server, it requests relevant data from several leaves, processes them, and returns a reduced result. Each rack of 30–34 aggregator-leaf servers works independently.

To maintain acceptable latency as traffic load changes during the day, the service adjusts the number of internal queries generated by each request; in aggregate, a rack handles between 500K and 200M internal queries per second. When load is low, the aggregator considers more items and

\textsuperscript{5}Path selection adds dummy edges to the graph until all nodes have the same degree (i.e., number of packets).
produces higher quality results. In times of heavy load, when the 99th percentile latency increases, the number of items considered per web request is reduced aggressively.

Cluster traffic is bursty, but most of the time utilizes a fraction of network capacity (Fig. 2-11). We measure throughput over 100 μs time intervals on one production server. 25% of these intervals have no ingress traffic, 25% have no egress traffic, and only 10% of these intervals have aggregate traffic exceeding 2 Gbits/s.

The production workload changes gently over time scales of tens of minutes, enabling comparative testing when schemes are applied in sequence. The live experiment ran for 135 minutes: the rack started in baseline mode, switched to Fastpass at 30 minutes, and back to baseline at 110 minutes.

**Experiment H: production results.** Fig. 2-12 shows that the 99th percentile web request service time using Fastpass is very similar to the baseline’s. The three clusters pertain to groups of machines that were assigned different load by the load-balancer. Fig. 2-13 shows the cluster’s load as the experiment progressed, showing gentle oscillations in load. Fastpass was able to handle the load without triggering the aggressive load-reduction.

Fastpass reduced TCP retransmissions by 2–2.5 × (Fig. 2-14). We believe the remaining packet loss is due to traffic exiting the rack, where Fastpass is not used to keep switch queues low. Extending the Fastpass scheduling boundary should further reduce this packet loss.
Figure 2-12: 99th percentile web request service time vs. server load in production traffic. Fastpass shows a similar latency profile as baseline.

2.8 Discussion

2.8.1 Large deployments

A single arbiter should be able to handle hundreds to thousands of endpoints. At larger scales, however, a single arbiter's computational and network throughput become bottlenecks, and several arbiters would need to cooperate.

A hierarchical approach might be useful: an arbiter within each cluster would send its demands for intra-cluster traffic to a core-arbiter, which would decide which timeslots each cluster may use, and what paths packets at these timeslots must follow. The cluster arbiters would then pass on these allocations to endpoints.

A core arbiter would have to handle a large volume of traffic, so allocating at MTU-size granularity would not be computationally feasible. Instead, it could allocate timeslots in bulk, and trust cluster arbiters to assign individual timeslots fairly among endpoints.

An alternative for large deployments could be the use of specialized hardware. An FPGA or ASIC implementation of timeslot allocation and path selection algorithms would likely allow a single arbiter to support much larger clusters.
2.8.2 Routing packets along selected paths

Packets must be made to follow the paths allocated to them by the arbiter. Routers typically support IP source routing only in software, if at all, rendering it too slow for practical use. Static routing [76] and policy based routing using VLANs are feasible, but could interfere with existing BGP configurations, making them less suitable for existing clusters. Tunneling (e.g. IP-in-IP or GRE) entails a small throughput overhead, but is supported in hardware by many switch chipsets making it a viable option [29].

Finally, routing along a specific path can be achieved by what we term ECMP spoofing. ECMP spoofing modifies fields in the packet header (e.g., source port) to specific values that will cause each switch to route the packet to the desired egress, given the other fields in the packet header and the known ECMP hash function. The receiver can then convert the modified fields to their original values, stored elsewhere in the packet.

2.8.3 Scheduling support in NICs

Fastpass enqueues packets into NIC queues at precise times, using high resolution timers. This frequent timer processing increases CPU overhead, and introduces operating-system jitter (e.g., due to interrupts). These timers would not be necessary if NICs implement support for precise packet
Figure 2-14: Median server TCP retransmission rate during the live experiment. Fastpass (middle) maintains a $2.5\times$ lower rate of retransmissions than baseline (left and right).

A current limitation of Fastpass is that it allocates at the granularity of timeslots, which can result in internal fragmentation: if an endpoint has less than a full timeslot worth of data to send, network bandwidth is left unused. This waste is reduced or eliminated when an endpoint sends many small packets to the same destination, which are batched together (§2.6.1).

Under the worst-case workload, where servers spray tiny packets to many destinations, Fastpass would not be able to reach link capacity. However, such a worst-case workload cannot utilize the hardware acceleration features in common NICs (namely, TCP Segmentation Offload, TSO), so generating a large volume of tiny packets is likely to cause a CPU bottleneck anyway. Measurements in large datacenters show workloads that are well-behaved for Fastpass [29, 57].

For workloads that nonetheless need to send tiny amounts of data to a large number of destinations, a possible mitigation is for the arbiter to divide some timeslots into smaller fragments and allocate these fragments to the smaller packets.
2.9 Summary

We showed how to design and implement a datacenter network in which a centralized arbiter determines the times at which each packet should be transmitted and the path it should take. Our results show that compared to the baseline network, the throughput penalty is small but queueing delays reduce dramatically, flows share resources more fairly and converge quickly, and the software arbiter implementation scales to multiple cores and handles an aggregate data rate of 2.21 Terabits/s.

Even with such a seemingly heavy-handed approach that incurs a little extra latency to and from the arbiter, tail packet latencies and the number of retransmitted packets reduce compared to the status quo, thanks to the global control exerted by the arbiter. Our results show that Fastpass opens up new possibilities for high-performance, tightly-integrated, predictable network applications in datacenters, even with commodity routers and switches.

We hope we have persuaded the reader that centralized control at the granularity of individual packets, achieved by viewing the entire network as a large switch, is both practical and beneficial.
Chapter 3

Flowtune: Flowlet-based Allocation

3.1 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.

3.1.1 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 3-1 shows how the allocator chooses a flow’s rate given link prices. It takes the flow’s utility function (3-1a). Then, it determines the flow’s price per unit bandwidth, which is the sum of prices on links it traverses (3-1b). The utility minus price determines the profit (3-1c); the allocator chooses the rate that maximizes the flow’s profit.

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.
(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 3-1: Illustration of how NUM chooses a flow rate.

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.

### 3.1.2 The NUM framework

The following table summarizes the notation used in this paper:

<table>
<thead>
<tr>
<th>L</th>
<th>Set of all links</th>
<th>L(s)</th>
<th>Links traversed by flow s</th>
</tr>
</thead>
<tbody>
<tr>
<td>S</td>
<td>Set of all flows</td>
<td>S(ℓ)</td>
<td>Flows that traverse link ℓ</td>
</tr>
<tr>
<td>p_ℓ</td>
<td>Price of link ℓ</td>
<td>c_ℓ</td>
<td>Capacity of link ℓ</td>
</tr>
<tr>
<td>x_s</td>
<td>Rate of flow s</td>
<td>U_s(x)</td>
<td>Utility of flow s</td>
</tr>
<tr>
<td>G_ℓ</td>
<td>By how much link ℓ is over-allocated</td>
<td></td>
<td></td>
</tr>
<tr>
<td>H_ℓ</td>
<td>How much flow rates on ℓ react to a change in p_ℓ</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 3.1: NED Notation

The goal is to allocate rates to all flows subject to network resource constraints: for each link ℓ ∈ L,

$$\sum_{s ∈ S(ℓ)} x_s ≤ c_ℓ.$$ (3.1)

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, $\sum_{s ∈ S} U_s(x_s)$. Thus, the rate
allocation should be the solution to the following optimization problem:

\[
\max \sum_s U_s(x_s)
\]

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

subject to (3.1).

**Solving NUM using prices.** The capacity constraints in (3.1) make it hard to solve the optimization problem directly. Kelly’s approach to solving NUM [44] 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.¹

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 3.A outlines why price duality works. Related NUM algorithms are discussed in §4 and Appendix 3.1.3.

### 3.1.3 Related NUM algorithms

**Gradient.** Arguably the simplest algorithm for adjusting prices is Gradient projection [46], 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., \(\gamma\) must be small). This is because depending on flow utility functions,

¹We discuss the requirements for convergence further below.
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 $\ell$ based not only on how flows on $\ell$ will react, but also based on how price changes to all other links impact flows on $\ell$:

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

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 [11]. 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.

### 3.1.4 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 $\ell$ will react to a change in that link’s price, which we denote $H_{\ell \ell}$.\(^2\)

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:

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

\(^2\)H is in fact the Hessian; NED computes the Hessian’s diagonal, $H_{\ell \ell}$. The Hessian’s diagonal is where NED gets its name.
Algorithm 1 Single iteration of Newton-Exact-Diagonal

NED updates rates $x = (x_s)$ given prices $p = (p_\ell)$ (“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 $p = (p_\ell)$, for each flow $s \in S$, update the rate:

$$x_s = x_s(p) = (U'_s)^{-1}(\sum_{\ell \in L(s)} p_\ell).$$  \hspace{1cm} (3.3)

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

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

$$p_\ell \leftarrow \max \left(0, p_\ell - \gamma H_{\ell \ell}^{-1} G_\ell\right),$$  \hspace{1cm} (3.4)

where $\gamma > 0$ is a fixed algorithm parameter (e.g. $\gamma = 1$), $G_\ell = (\sum_{s \in S(\ell)} x_s) - c_\ell$, $H_{\ell \ell} = \sum_{s \in S(\ell)} \frac{\partial x_s(p)}{\partial p_\ell}$.

From (3.3), $\frac{\partial x_s(p)}{\partial p_\ell} = ((U'_s)^{-1})'(\sum_{m \in L(s)} p_m)$.

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 $U_\ell$ 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 [44].

### 3.2 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 §3.2).

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 $r_\ell = \frac{\sum \alpha \in \mathcal{S}(\ell) x_\alpha}{c_\ell}$. The most over-congested link has the ratio $r^* = \max_{\ell \in L} r_\ell$; all flows are scaled using this ratio:

$$\bar{x}_\alpha = \frac{x_\alpha}{r^*}.$$  \hfill (3.5)

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 $w \log x$, this preserves the fairness of allocation. However, as shown in §3.2, 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 $\ell$ by at least a factor of $r_\ell$, which guarantees the rates through the link are at most the link capacity. Formally, F-NORM sets

$$\bar{x}_\alpha = \frac{x_\alpha}{\max_{\ell \in L(s)} r_\ell}.$$  \hfill (3.6)

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 §3.3.
(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 3-2: Partitioning of network state.

(a) Step 1: 2x2 processors. (b) Step 2: 4x4 processors. (c) Step 3: 8x8 processors.

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

3.3 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 [10] or a Google Jupiter aggregation block [63], 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.

Figure 3-2 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. 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 3-3 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 [29, 4], 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 3-3 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.

3.4 Evaluation

We evaluate Flowtune using a cluster deployment, micro-benchmarks, and ns-2 and numeric simulations. ns-2 simulations allows comparison with state-of-the-art schemes whose implementations are only readily available in the ns-2 simulator: pFabric [6], sfqCoDel [52], and XCP [43].

<table>
<thead>
<tr>
<th>EC2 Experiments (§3.4.1)</th>
</tr>
</thead>
<tbody>
<tr>
<td>(A) On Amazon EC2, Flowtune’s sharing of available throughput is more fair than the baseline Linux implementation running Cubic.</td>
</tr>
<tr>
<td>(B) Flowtune makes transfers on EC2 more predictable: Many-to-One Coflow Completion Time was sped up by $1.61 \times$ in p95 and $1.24 \times$ in p90.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Multicore micro-benchmarks (§3.4.2)</th>
</tr>
</thead>
<tbody>
<tr>
<td>(C) A multi-core implementation optimizes traffic from 384 servers on 4 cores in $8.29\mu$s. 64 cores schedule 4608 servers’ traffic in $30.71\mu$s – around 2 network RTTs.</td>
</tr>
</tbody>
</table>
ns-2 Simulations (§3.4.3)

(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: \(8.6 \times 10^{-9}\)x and \(1.7 \times 2.4\times\) lower than DCTCP and pFabric on 1-packet flowlets, and \(3.5 \times 3.8\times\) than sfqCoDel on 10-100 packets.

(I) Flowtune keeps p99 network queuing delay under 8.9 μs, \(12\times\) 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 (§3.4.4)

(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.

Table 3.2: Flowtune: Summary of Results.

3.4.1 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 3-4 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 3-5 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.

<table>
<thead>
<tr>
<th>Metric</th>
<th>Baseline</th>
<th>Flowtune</th>
<th>Speedup</th>
</tr>
</thead>
<tbody>
<tr>
<td>median</td>
<td>1.859249</td>
<td>1.787622</td>
<td>1.04×</td>
</tr>
<tr>
<td>p90</td>
<td>2.341086</td>
<td>1.881433</td>
<td>1.24×</td>
</tr>
<tr>
<td>p95</td>
<td>3.050718</td>
<td>1.894544</td>
<td>1.61×</td>
</tr>
</tbody>
</table>

Table 3.3: Comparison of coflow completion time, Baseline vs. Flowtune.

3.4.2 Multicore micro-benchmarks

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.

(C) Iteration time. This micro-benchmark measures the average NED iteration time, i.e., the “Run
1 iteration” in Figure 1-8c. The following table shows the number of cycles taken for different
choices of network sizes and loads:
<table>
<thead>
<tr>
<th>Cores</th>
<th>Nodes</th>
<th>Flows</th>
<th>Cycles</th>
<th>Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>384</td>
<td>3072</td>
<td>19896.6</td>
<td>8.29 μs</td>
</tr>
<tr>
<td>16</td>
<td>768</td>
<td>6144</td>
<td>21267.8</td>
<td>8.86 μs</td>
</tr>
<tr>
<td>64</td>
<td>1536</td>
<td>12288</td>
<td>30317.6</td>
<td>12.63 μs</td>
</tr>
<tr>
<td>64</td>
<td>1536</td>
<td>24576</td>
<td>33576.2</td>
<td>13.99 μs</td>
</tr>
<tr>
<td>64</td>
<td>3072</td>
<td>49152</td>
<td>57035.9</td>
<td>23.76 μs</td>
</tr>
<tr>
<td>64</td>
<td>4608</td>
<td>49152</td>
<td>73703.2</td>
<td>30.71 μs</td>
</tr>
</tbody>
</table>

Table 3.4: NED iteration time for different numbers of cores, nodes, and flows.

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 §3.4.3, and discussed in §3.5.

**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.

### 3.4.3 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 [6]. 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 [57]. Appendix 3.B 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 $\gamma$ 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.
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 3-6 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.

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 3-7 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 §3.5.

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 3-8 shows the measured reduction in update traffic for different thresholds compared to the 0.01 threshold in Figure 3-7. Notifying servers of changes
Figure 3-8: Notifying servers when rates change by more than a threshold substantially cuts control traffic volume.

Figure 3-9: The fraction of rate-update traffic remains constant as the network grows from 128 to 2048 servers.

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 3-9 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.

Comparison to prior schemes

We compare Flowtune to DCTCP [4], pFabric [6], XCP [43], and Cubic+sfqCoDel [52].
Figure 3-10: Improvement in 99th percentile flow completion time with Flowtune. Note the different scales of the y axis.

(II) 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 [20].

Figure 3-10 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 performs 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 (§3.4.3), which causes flows to finish slowly.
Figure 3-11: Flowtune keeps 2-hop and 4-hop 99th-percentile queuing delays below 8.9 μs. At 0.8 load, XCP has $3.5 \times$ longer queues, DCTCP $12 \times$. pFabric and sfqCodel do not maintain FIFO ordering so their p99 queuing delay could not be inferred from sampled queue lengths.

(I) Queuing delay. The following experiments collected queue lengths, drops, and throughput from each queue every 1 ms. Figure 3-11 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 $12 \times$ longer, contributing to the significant speedup shown in Figure 3-10. 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 3-12 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, but timeouts due to these drops could result in high p99 FCT, which affects medium-sized flows (figure 3-10). 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.

(K) Fairness. Figure 3-13 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 $2 \times$ higher rate, losing a point when a flow gets $2 \times$ lower rate. Flowtune has better fairness than the compared schemes: a flow’s fairness score has
Figure 3-12: pFabric and sfqCoDel have a significant drop rate (1-in-13 for sfqCoDel). Flowtune, DCTCP, and XCP drop negligible amounts.

Figure 3-13: Comparison of proportional fairness of different schemes, i.e., $\sum \log_2(\text{rate})$. Flowtune allocates flows closer to their proportional-fair share.

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.

3.4.4 Numerical simulations

Experiments in this section compared different NUM optimizers using numerical simulations. Simulations ran the web flow size distribution described in §3.4.3.

(L) Over-allocation in NUM. Figure 3-14 shows the total amount of over-capacity allocations when there is no normalization. FGM is the Fast Weighted Gradient Method [13]. 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
Figure 3-14: Normalization is necessary; without it, optimization algorithms allocate more than link capacities.

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.

(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 3-15 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.

3.5 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.
Figure 3-15: Normalizing with F-NORM achieves close to optimal throughput, while avoiding over-capacity allocations. U-NORM’s throughput is low in comparison.

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 [24] networks, given the MPLS configuration stream. In VL2 [29]-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 manage 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 accommodate 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 [30], 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 separate 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 §3.4.3 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 [57]. Lower load translates directly to reduced update traffic (Figure 3-7).

In small deployments of a few hundred endpoints, it might be feasible to install a few NICs in the allocator. Figure 3-8 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.

### 3.6 Summary

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 $8.6 \times 10.9 \times$ and $2.1 \times 2.9 \times$ lower p99 FCT for 1-packet and 1-10 packet flows compared to DCTCP.

Compared to Fastpass, Flowtune scales to $8 \times$ more cores and achieves $10.4 \times$ higher throughput per core, does not require allocator replication for fault-tolerance, and achieves weighted proportional-fair rate allocations quickly in between 8.29 $\mu$s and 30.71 $\mu$s ($\leq 2$ RTTs) for networks that have between 384 and 4608 nodes.
Appendix

3.A Appendix: 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 (3.2) is strictly concave. The constraints in (3.2) 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 (3.2) is non-empty, bounded and convex. The Lagrangian of (3.2) is

$$\mathcal{L}(x, p) = \sum_{s \in S} U_s(x_s) - \sum_{\ell \in L} p_\ell \left( \sum_{s \in S(\ell)} x_s - c_\ell \right). \quad (3.7)$$

with dual variables $p_\ell$, and the dual function is defined as

$$D(p) = \max \mathcal{L}(x, p) \text{ over } x_s \geq 0, \text{ for all } s \in S. \quad (3.8)$$

The dual optimization problem is given by

$$\min D(p) \text{ over } p_\ell \geq 0, \text{ for all } \ell \in L. \quad (3.9)$$

From Slater’s condition in classical optimization theory, the utility of the solution of (3.2) is equal to its Lagrangian dual’s (3.9), and given the optimal solution $p^*$ of (3.9) it is possible to find the optimal solution for (3.2) from (3.8), i.e., using the rate update step. More details on solving NUM using Lagrange multipliers appear in [44, 11].
3.B Appendix: Simulation CDFs

This section reproduces the flow size distribution graphs from [57], 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 3.B.1 shows the flow size CDF. Table 3.B.1 summarizes statistics of the different workloads.

<table>
<thead>
<tr>
<th>Metric</th>
<th>cache</th>
<th>hadoop</th>
<th>web</th>
</tr>
</thead>
<tbody>
<tr>
<td>mean</td>
<td>567658.4</td>
<td>1296082.4</td>
<td>33105.3</td>
</tr>
<tr>
<td>median</td>
<td>22924</td>
<td>651</td>
<td>1419</td>
</tr>
<tr>
<td>p90</td>
<td>2432831</td>
<td>117471</td>
<td>55179</td>
</tr>
<tr>
<td>p95</td>
<td>2716140</td>
<td>266706</td>
<td>208966</td>
</tr>
<tr>
<td>p99</td>
<td>3131038</td>
<td>6405830</td>
<td>417147</td>
</tr>
<tr>
<td>p999</td>
<td>5663439</td>
<td>251359175</td>
<td>2560769</td>
</tr>
</tbody>
</table>

Table 3.B.1: Statistics of the different flow size distributions (bytes).
Chapter 4

Related work

**Distributed approaches.** Distributed approaches usually set out to solve a restricted datacenter problem: minimizing Flow Completion Time (FCT), meeting deadlines, balancing load, reducing queueing, or sharing the network. To our knowledge, no previous scheme provides a general platform to support all these features, and some schemes perform sub-optimally because they lack complete knowledge of network conditions.

DCTCP [4] and HULL [5] reduce switch queueing, but increase convergence time to a fair-share of the capacity, and do not eliminate queueing delay.

In MATE [24] and DARD [76], ingress nodes reroute traffic selfishly between paths until converging to a configuration that provides good load balance across paths.

In multi-tenant datacenters, Seawall [61] provides weighted fair sharing of network links, and EyeQ [41] enforces traffic constraints.

Schemes such as pFabric [6] and PDQ [34] modify the switches to reduce flow completion time, while D³ [73] and PDQ minimize flow completion times to meet deadlines. These switch modifications raise the barrier to adoption because they need to be done across the network. PDQ and pFabric use a distributed approximation of the *shortest remaining flow first* policy, which Fastpass can implement in the arbiter.

Differentiated Services (DiffServ, RFC2474) provides a distributed mechanism for different classes of traffic to travel via distinct queues in routers. The number of DiffServ Code Points available is limited, and in practice operational concerns restrict the number of classes even further.
Most commonly, there are classes for bulk traffic and latency-sensitive traffic, but not a whole lot more.

NUMFabric [50] 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.

**Centralized approaches.** Several systems use centralized controllers to get better load balance and network sharing, but they work at "control-plane" granularity, unable to quickly and consistently enforce policy as traffic changes in the datacenter.

Hedera [3] gathers switch statistics to find elephant flows and reroutes those to avoid network hotspots. Compared to Fastpass, it is orders of magnitude slower in adapting to changing traffic. Hedera is complementary to Flowtune: Hedera will re-route flows based on congestion, and Flowtune will allocate rates given their Hedera-assigned paths. Further, integrating the two systems can give Hedera its required input information with very low latency.

Mordia [26] and Datacenter TDMA [71] 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, Fastpass and Flowtune update allocations within tens of microseconds.

Orchestra [16] coordinates transfers in the shuffle stage of MapReduce/Dryad so all transfers finish roughly together, by assigning weights to each transfer. Orchestra is an application-level mechanism; the actual transfers use TCP with its multiple RTTs to converge and large oscillations around the fair allocation.

Several systems control datacenter routes and rates, but are geared for inter-datacenter traffic. BwE [45] 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 [40]), and SWAN [35] receives demands from non-interactive services, computes rates, and reconfigures OpenFlow switches every 5 minutes. measured Fastpass and Flowtune support a richer set of policies, with orders of magnitude smaller update times.

**NUM solvers.** The first-order methods [44, 46, 65] do not estimate $H_{t\ell}$ or use crude proxies. Gradient projection [46] adjusts prices with no weighting. Fast Weighted Gradient [13] uses a crude upper bound on the convexity of the utility function as a proxy for $H_{t\ell}$.
The Newton-like method [11], like NED, strives to use $H_{ee}$ 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 $H_{ee}$ explicitly from flow utilities, saving the time required to obtain estimates, and getting an error-free result. Appendix 3.1.3 discusses the Gradient, Newton and Newton-like methods in more detail.

Recent work [72] 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 [77] 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).

Note: see §3.1.3 for further discussion.

Parallel architectures. Conflict-free Replicated Data Types [60] (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 [15], 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 [32], 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.
Chapter 5

Conclusion

This thesis presented two new network architectures, Fastpass and Flowtune, which allow operators to specify explicit network resource allocation policy. The network then quickly and consistently enforces this policy. Both Fastpass and Flowtune can be deployed in software only on commodity CPUs and require no modification to switch mechanisms.

Fastpass and Flowtune assume in a centralized controller responsibilities traditionally vested in endpoints and switches: congestion control, and scheduling and queue management, and in Fastpass’s case also path selection (Figure 1-10). Removing responsibility from switches offers operators more control, while making switches simpler, and thus potentially cheaper and easier to scale.

Through the design of the Fastpass Control Protocol and efficient cache-aware algorithms, Fastpass and Flowtune address the three concerns in centralized control: latency of allocation, scaling to large networks, and fault tolerance. Fastpass can handle 2.21 Terabits/s (§2.7, F) and allocate within 10s of microseconds (§2.7, B). Flowtune can allocate 4608 servers in 30.71 µs (§3.4, C). Controllers maintain only soft state, and FCP re-establishes connections aggressively allowing hot backup replication for fault tolerance.

In both architectures, the controllers ensure aggregate allocations through each link do not exceed link capacity, practically eliminating in-network queueing and packet drops, and enabling networks to run at higher utilizations.

We believe that networks that allow operators to explicitly specify resource allocation policy, are an important tool for network engineers and application developers in agreeing on and enforcing
application performance goals. We hope the algorithms and techniques proposed in this work will help power networks whose performance is more easily managed and thus provide better service quality to users.

5.1 Future work

Several open questions can serve as the subject of future work.

Handling mice. We have measured Flowtune to schedule 4608 endpoints in 31 μs. But what happens to flows with a handful of packets which should transmit in much less than 31 μs ("mice")? One possible solution is for Flowtune to allocate to flows, say, only 95% of capacity and let mice transmit without going through the allocation process. Another option is to use Fastpass for mice and Flowtune for the others ("elephants"), and for the two systems to employ a mechanism that divides capacity between them.

External traffic. Fastpass and Flowtune clusters must support communication with other clusters and the Internet, where the outside hosts do not coordinate with the controller. Can the system measure and compensate for incoming traffic, and potentially apply policy to incoming traffic? In one potential solution, endpoints can measure the amount of external traffic and report to the controller, which instructs endpoints to mark or drop packets according to policy. However how to do this remains an open problem.

Deadlines, co-flows. Fastpass supports policies that can be translated into per-packet priorities, and Flowtune maximizes aggregate utility. However applications might require other types of guarantees: for the transfers to finish by a specified time ("deadlines"), or for a group of flows to be scheduled according to its worst-performing flow ("co-flows"). Is it possible to modify Fastpass and Flowtune to support these policies? One possible direction is to allow flowlet utility functions to change in time, thereby creating a market; flows will change how aggressively they bid for throughput as their deadline approaches. Co-flows will bid more strongly on their poorly-performing flows to get better performance.

Additional topology support. Fastpass and Flowtune both support Clos topologies. Can Fastpass support other topologies? (this would require efficient path selection). In Flowtune, NED should support any topology, but we have shown an efficient multicore algorithm only for 2-tier Clos
topologies. Can this be extended to 3-tier Clos? To the Wide Area Network? To other topologies such as Jellyfish [64]?

**Multi-resolution allocation in Fastpass.** An idea to scale the Fastpass arbiter would be to use multiple timeslot sizes. Then, the arbiter can do less work per packet on long flows, as those would use fewer, larger timeslots. Some questions that need solving: How to decide how many timeslots to produce of each size? At a particular time $t$, is there only one timeslot size, or a mixture of different sizes? Will the allocator decide on timeslot size a priori, or dynamically split timeslots of some hosts?

**NIC scheduling support.** Fastpass and Flowtune currently schedule and pace packets in software. This consumes precious CPU cycles, and in the case of VMs, precludes the direct use of NIC queues by VMs (SR-IOV). This is a classic setup for hardware acceleration: the operating enqueues packets to the NIC. An endpoint driver reads notifications of packets/flowlets and programs the transmission schedule or flow rates. The NIC can then take care of the actual transmission of packets. With PTP, the NIC can achieve even higher transmission precision than Fastpass’s software implementation.

**Further scaling.** We believe with further work, Fastpass and Flowtune would scale to larger networks and faster link speeds. With Fastpass, one might explore how to better use the cheap 64-bit bitwise logic operations, how to rearrange data structures to mostly use sequential reads when processing demands, hierarchical allocation, and multi-resolution timeslots (mentioned above). With Flowtune, one might explore RDMA for cheaper communication, using integers instead of floating-point arithmetic, and hierarchical allocation.

### 5.2 A personal note

It is an exhilarating experience, advancing from the whiteboard to a crude implementation, then refining it first a VM testbed, then on 1 Gbits/s links, then 10 Gbits/s links, starting with synthetic traffic and finally operating on production traffic. As the volume, frequency, and heterogeneity of packets increase, all poor design decisions, bugs, and race conditions surface, and we uncovered deeper insights into our system.

I vividly remember those few times I activated Fastpass on production traffic at Facebook, each time hoping the system will achieve its promised performance, and going back to fix the
different corner cases we hadn’t anticipated. And that time we activated Fastpass, and everything just worked. Few achievements in my experience were more powerful than successfully designing, implementing and deploying a system that robustly makes, communicates, and enforces decisions at microsecond-resolution and wire-speed.

Like many endeavors in computer science systems research, implementing a system from start to finish takes years. If you have the academic freedom and a great idea, do not be discouraged by the time constants. There is nothing quite like it.
Bibliography


