NetBump: User-extensible Active Queue Management with Bumps on the Wire

George Porter* Rishi Kapoor* Sambit Das* Mohammad Al-Fares*
Hakim Weatherspoon** Balaji Prabhakar†
Amin Vahdat*‡
*UC San Diego **Cornell University †Stanford University ‡Google Inc.

Abstract

Engineering large-scale data center applications built from thousands of commodity nodes requires both an underlying network that supports a wide variety of traffic demands, and low latency at microsecond timescales. Many ideas for adding innovative functionality to networks, especially active queue management strategies, require either modifying packets or performing alternative queuing to packets in-flight on the data plane. However, configuring packet queuing, marking, and dropping is challenging, since buffering in commercial switches and routers is not programmable.

In this work, we present *NetBump*, a platform for experimenting with, evaluating, and deploying a wide variety of active queue management strategies to network data planes with minimal intrusiveness and at low latency. NetBump leaves existing switches and endhosts unmodified by acting as a "bump on the wire," examining, marking, and forwarding packets at line rate in tens of microseconds to implement a variety of virtual active queuing disciplines and congestion control mechanisms. We describe the design of NetBump, and use it to implement several network functions and congestion control protocols including DCTCP and 802.1Qau quantized congestion notification.

1. INTRODUCTION

One of the ultimate goals in data center networking is predictable, congestion-responsive, low-latency communication. This is a challenging problem and one that requires tight cooperation between endhost protocol stacks, network interface cards, and the switching infrastructure. While there have been a range of interesting ideas in this space, their evaluation and deployment have been hamstrung by the need to develop new hardware to support functionality such as Active Queue Management (AQM) [13, 24], QoS [51], traffic shaping [20], and congestion control [1, 2, 21, 30]. While simulation can show the merits of an idea and support publication, convincing hardware manufacturers to actually support new features requires evidence that a particular technique will actually deliver promised benefits for a range of application and communication scenarios.

We consider a model where new AQM disciplines can be deployed and evaluated directly in production data center networks without modifying existing switches or endhosts. Instead of adding programmability to existing switches themselves, we instead deploy "bumps on the wire," called *NetBumps*, to augment the existing switching infrastructure.[1] Each NetBump exports a virtual queue primitive that emulates a range of AQM mechanisms [13, 14] at line rate that would normally have to be implemented in the switches themselves.

NetBump provides an efficient and easy way to deploy and manage active queue management separate from switches and endhosts. NetBumps enable AQM functions to be incrementally deployed and evaluated by their placement at key points in the network. This makes implementing new functions straightforward. In our experience, new queuing disciplines, congestion control strategies, protocol-specific packet headers (e.g. for XCP [21]), and new packets (for a new congestion control protocol we implement) can be easily built and deployed at line rate into existing networks. Developers can experiment with protocol specifics by simply modifying software within the bump.

The NetBump requirements are: rapid prototyping and evaluation, ease of deployment, support for line rate data processing, low latency (i.e. tens of μs), packet marking and transformation for a range of AQM and congestion control policies, and support for distributed deployment to support data center multipath topologies. We require low latency since we target LAN switches, rather than WAN router deployments. We greatly reduce the latency imposed by NetBump because our functionality is limited to modifications of packets in flight, with no actual queuing or buffering done within NetBump. We expect these bumps on the wire to be part of the production network that will form a proving ground to inform

[1] The "bump on the wire" term here is unrelated to previous work about IPsec deployment boxes [22].

eventual hardware development (see Fig. 1 for an example deployment scenario).

There are several implementation choices for building NetBumps. While it is possible to develop custom silicon for such functionality, the time to market is long (2-3 years) and the non-recurring engineering costs are often prohibitive for functionality that does not have a pre-existing market. Programmable network processors have been a hot topic for number of years [45]. However, their utility has been hampered by a difficult programming model, which is also the case for FPGA-based designs that are programmed in Verilog or VHDL. The complexity of these approaches prevent experimenting with novel network programming ideas.

On the other hand, software programmable routers like Click [23] and RouteBricks [8] provide a powerful and easy-to-program language for implementing in-network datapath extensions that are ideal for supporting NetBump. We implemented NetBump on RouteBricks, and found that its underlying NIC device driver batches packets to support higher throughput at the cost of higher latency. We also considered an alternative implementation based on a user-level, zero-copy, kernel-bypass network API. User-level networking is not a new idea, and commercial implementations are nearly 20 years old. However, we found that it performed well, and was able to support custom active queue management of minimum-sized 64-byte packets at a rate of 14.17Mpps (i.e. 10Gbps line rate) with one CPU core at 20-30μs. In part this performance is the result of NetBump's simpler packet handling model supporting pass-through functionality on the wire, as compared to general-purpose software routers.

The primary contributions of this paper are: 1) the design of a "bump on the wire" specifically focusing on evaluating and deploying new buffer management packet processing functions, 2) a simple virtual Active Queue Management (vAQM) implementation to indirectly manage the buffers of neighboring, unmodified switches, 3) the evaluation of several new programs implemented on top of NetBump, including an implementation of IEEE 802.1Qau-QCN L2 congestion control, and 4) an extensible and *distributed* traffic update and management platform for remote physical switch queues.

2. MOTIVATION

In this section we first present an example of NetBump functionality in action, and then motivate our requirements for a low-latency implementation.

2.1 NetBump Example

In Fig. 2, we show a simple network where two source hosts H_1 and H_2 each send data to a single destination

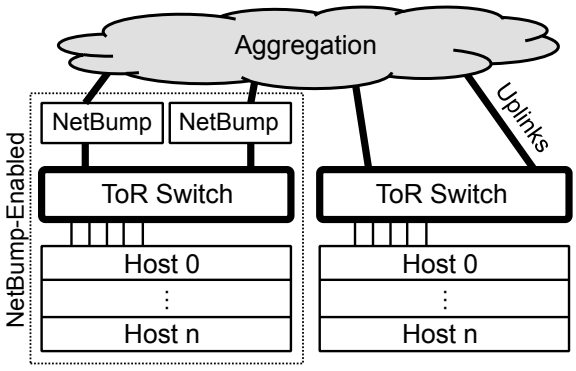


Figure 1: Deployment scenario in the data center. "NetBump-enabled racks" include NetBumps in-line with the Top-of-Rack (ToR) switch's uplinks, and monitor output queues at the host-facing ports.

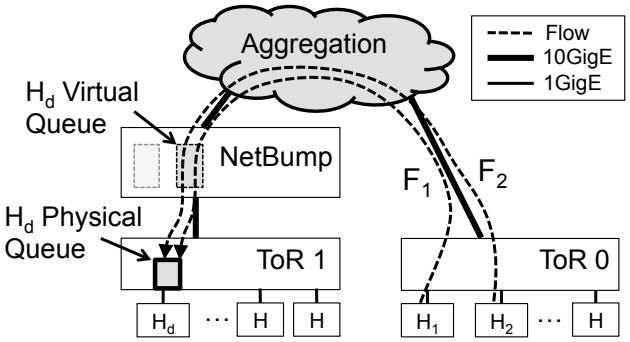


Figure 2: An example of NetBump at ToR switch, monitoring downstream physical queues.

host H_d (in flows F_1 and F_2, respectively). H_1 and H_2 are connected to Switch0 at 1Gbps. Switch 0 has a 10Gbps uplink to a NetBump (through the aggregation layer), and on the other side of the NetBump is a second 10Gbps link to Switch1. Destination host H_d is attached to Switch1 at 1Gbps. Flows F_1 and F_2 each have a maximum bandwidth of 1Gbps, and since host H_d has only a single 1Gbps link, congestion will occur on H_d's input or output port in Switch1 if $rate(F_1) + rate(F_2) >$ 1Gbps. Without NetBump, assuming Switch1 implements a drop-tail queuing discipline, packets from F_1 and F_2 will be interleaved in H_d's physical queue until the queue becomes full, at which point Switch1 will drop packets arriving to the full queue. This leads to known problems such as burstiness and lack of fairness.

Instead, as NetBump forwards packets from its input to its output port, it estimates the occupancy of a virtual queue associated with H_d's output port buffer. When a packet arrives, H_d's virtual queue occupancy is increased by the packet's size. Because NetBump has the topology information and knows the speed of

the link between Switch1 and H_d (§ 3.1), it computes the estimated *drain rate*, or the rate that data leaves H_d's queue. By integrating this drain rate over the time between subsequent packets, it calculates the amount of data that has left the queue since the last packet arrival.

Within NetBump, applications previously requiring new hardware development can instead act on the virtual queue. For example, to implement a Random Early Detection (RED) queuing discipline, the NetBump shown in Fig. 2 maintains a virtual output queue for each physical queue in Switch1. This virtual queue maintains two parameters, `MinThreshold` and `MaxThreshold`, as well as a weighted estimate of the current downstream queue length. According to the RED discipline, packets are sent unmodified when the moving average of the queue length is below the `MinThreshold`, the packets are marked (or dropped) probabilistically when the average is between the two thresholds, and packets are unconditionally marked (or dropped) when it is above `MaxThreshold`.

Note that in this example, just as in all the network mechanisms presented in this paper, packets are never delayed or queued in the NetBump itself. Instead, NetBump marks, modifies, or drops packets at line rate as if the downstream switch directly supported the functionality in question. Note also that NetBump is not limited to a single queuing discipline or application–it is possible to compose multiple applications (e.g. QCN congestion control with Explicit Congestion Notification (ECN) marking [11]). Furthermore, AQM functionality can act only on particular flows transiting a particular end-to-end path if desired.

2.2 Design Requirements

The primary goal of NetBump is enabling rapid and easy evaluation of new queue management and congestion control mechanisms in deployed networks with minimal intrusiveness. We next describe the requirements that NetBump must meet to successfully reach this goal.

Deployment with unmodified switches and endhosts: We seek to enable AQM development and experimentation to take place in the data center or enterprise itself, rather than separate from the network. This means that NetBump works despite leaving switches and endhosts unmodified. Thus a requirement of NetBump is that it implements a virtual Active Queue Management (vAQM) discipline that tracks the status of neighboring switch buffers. This will differ from previous work that applies this technique within switches [13, 24], as our implementation will be remote to the switch.

Distributed deployment: Modern networks increasingly rely on multipath topologies both for redundancy in the face of link and switch failure, and for improving throughput by utilizing several, parallel links. Left unaddressed, multipath poses a challenge for the NetBump model since a single bump may not be able to monitor all of the flows heading to a given destination. Therefore a requirement for NetBump is that it supports enough throughput to manage a sufficient number of links, and that it supports a distributed deployment model. In a distributed model, multiple bumps deployed throughout the network coordinate with each other to manage flows transiting them. In this way, a set of flows taking separate network paths can still be subjected to a logically centralized, though physically distributed, AQM policy.

Ease of development: Rather than serving as a final deployment strategy, we see NetBump as an experimental platform, albeit one that is deployed directly on the production network. Thus rapid prototyping and reconfiguration are a requirement of its design. Specifically, the platform should export a clear API with which users can quickly develop vAQM applications using C/C++.

Minimizing latency: Many data center and enterprise applications have strict latency deadlines, and any datapath processing elements must likewise have strict performance guarantees, especially given NetBump's target deployment environment of data center networks, whose one-way latency diameters are measured in microseconds. Since the throughput of TCP is in part a function of the network round-trip time [40], any additional latency imposed by NetBump can affect application flows. To show this effect, we measured the completion times of two flows–one in which a single byte is exchanged between a sender-receiver pair, and one where 1MB is exchanged between that same pair. Fig. 3 shows the normalized completion times of each flow as a function of one-way middlebox latency. Perhaps not surprising, adding even tens of microseconds of one-way latency has a significant impact on flow completion times when the baseline network RTT is very small.

Forwarding at line rate: Although data center hosts still primarily operate at 1Gbps, 10Gbps has become standard at rack-level aggregation. Deploying a NetBump inline with top-of-rack uplinks and between 10Gbps switches will require an implementation that can support 10Gbps line rates. The challenge then becomes keeping up with packet arrival rates: 10Gbps corresponds to 14.88M 64-byte minimum sized packets per second, including Ethernet overheads.

3. DESIGN

In this section we describe the design of the primary NetBump vAQM pipeline, including how this design can scale to support faster links and a distributed deployment for multi-path data center designs. We discuss our implementation choices in § 5.

3.1 The NetBump Pipeline

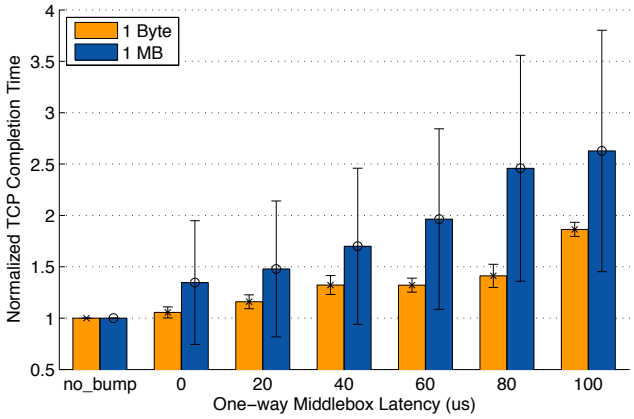


Figure 3: Effect of middlebox latency on completion time of short (1 Byte) and medium-sized (1MB) TCP flows. Baseline (direct-connect) transfer time was 213μs (1B), 9.0ms (1MB), others are through a NetBump with configurable added delay.

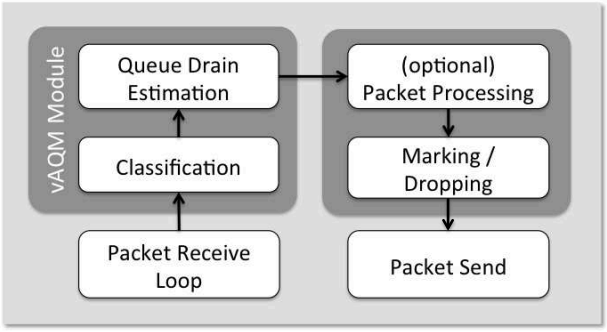


Figure 4: The NetBump pipeline.

The core NetBump pipeline consists of four algorithms: 1) packet classification, 2) virtual queue (VQ) drain estimation, 3) packet marking/dropping, and optionally 4) extensible packet processing.

Virtual Queue Table Data Structure: Each NetBump maintains a set of virtual queues, which differ from physical queues in that they do not store or buffer packets. Instead, as packets pass through a virtual queue, it maintains state on what its occupancy *would* be if it were actually storing packets. Thus each virtual queue must keep track of 1) the number and sizes of packets transiting it, 2) the packet arrival times, and 3) the virtual rate at which they drain from the queue. Note that packets actually drain at line rate (i.e. 10Gbps), however a virtual queue could be configured with a virtual drain parameter of 1Gbps or 100Mbps.

The virtual queue table is a simple data structure kept by the NetBump that stores each of these three parameters for each virtual queue supported by that bump. For the AQM functionality we consider, we only need to know the virtual queue occupancy and drain rate, and so each virtual queue keeps 1) the size in bytes of the queue, 2) the time the last packet arrived to the queue, and 3) the virtual queue drain rate. These values are updated when a packet arrives to the virtual queue.

1. Packet Classification: As packets arrive to the NetBump, they must first be classified to determine into which virtual queue they will be enqueued. This classification API is extensible in NetBump, and can be overridden by a user as needed. A reasonable scheme would be to map packets to virtual queues corresponding to the downstream physical switch output buffer that the packet will reside in when it leaves the bump. In this case the virtual queue is emulating the downstream switch port directly. Note that virtual queues do not have to be associated in this way, though they are for most of the applications we consider in this work.

To make this association, NetBump requires two pieces of information: the mapping of packet destinations to downstream output ports, and the speed of the link attached to that port. The mapping is needed to determine the destination virtual queue for a particular packet, and the link speed is necessary for estimating the virtual queue's drain rate. There are a variety of ways of determining these values. The bump could query neighboring switches (e.g. using SNMP) for their outgoing link speeds, or those values could be statically configured when the bump is placed in the network. For software-defined networks based on OpenFlow [15, 33], the central controller could be queried for host-to-port mappings and link speeds, as well as the network topology. In our evaluation, we statically configure the NetBump with the port-to-host mapping and link speeds.

2. Queue Drain Estimation: The purpose of the queue drain estimation algorithm is to calculate, at the time a packet is received into the bump, the occupancy of the virtual queue associated with the packet (Fig. 5). The virtual queue estimator is a leaky bucket that is filled as packets are assigned to it, and drained according to a fixed drain rate determined by the port speed [49].

Lines 1-6 implement the leaky bucket. First, the elapsed time since the last packet arrived to this virtual queue is calculated. This elapsed time is multiplied by a physical port's rate to calculate how many bytes would have left the downstream queue since receiving the last packet. The physical port's drain rate comes from the link speed of the downstream switch or endhost. This amount is then subtracted from the current estimate (or set to zero, if the result would be negative) of queue occupancy to get an updated occupancy. If this is the first packet to be sent to that port, then the default

Function	Description
`void init(vQueue *vq, int drainRate);`	Initializes a virtual queue and set the given drain rate.
`vQueue * classify(Packet *p) const;`	Classifies a packet to a virtual queue.
`void vAQM(Packet *p, vQueue *vq);`	Updates internal vAQM state during packet reception.
`int estimateQlen(vQueue *vq) const;`	Returns an estimate of a virtual queue's length.
`int process(Packet *p, vQueue *vq);`	Defines packet processing. Modify, duplicate, drop, etc.
`int forward(Packet *p) const;`	Gets the output port (Optional, for multi-NIC bumps).

Table 1: The NetBump API.

```
Procedure vAQM(Packet *pkt, vQueue *VQ):
1   if (VQ→lastUpdate > 0) {
2     elapsedTime = pkt→timestamp – VQ→lastUpdate
3     drainAmt = elapsedTime * VQ→rate
4     VQ→tokens –= drainAmt
5     VQ→tokens = max(0, VQ→tokens)
6   }
7   VQ→tokens += pkt→len
8   VQ→lastUpdate = pkt→timestamp

Procedure RED(Packet *pkt, vQueue *VQ):
9   VQ→avg = calculateAvg(v→tokens)
10  if (VQ→avg > VQ→MaxThresh) {
11    VQ→tokens –= pkt→len
12    drop(pkt)
13  } else if (VQ→avg > VQ→MinThresh) {
14    calculate probability ρ
15    with probability ρ:
16      mark(pkt)
17  }
```

Figure 5: The queue drain estimation algorithm and the implementation of RED.

queue occupancy estimate of 0 is used instead. Lastly, the "last packet arrival" field of the virtual queue is updated accordingly.

A key design decision in NetBump is whether to couple the size of the virtual queue inside the bump with the actual size of the physical buffer in the downstream switch. If we knew the size of the downstream queue, then we could set the maximum allowed occupancy of the virtual queue accordingly. This would be challenging in general, since switches do not typically export the maximum queue size programmatically. Furthermore, for shared buffer switches, this quantity might change based on the instantaneous traffic in the network. In fact, by assuming a small buffer size in the virtual queue within NetBump, we can constrain the flow of packets to reduce actual buffer occupancy throughout the network. Thus, assuming small buffers in our virtual queues has beneficial effects on the network, and simplifies NetBump's design.

. Packet Marking/Dropping: At line 9 in Fig. 5, etBump has an estimate for the virtual queue occuancy. Here a variety of actions can be performed, ased on the application implemented in the bump. he example code shows a general random-early drop RED) application [16]. In this example, there is a nin" limit that results in packet marking, and a "max" mit that results in packet dropping. Packet marking kes the form of setting the ECN bit in the header, and opping is performed simply in software.

. Extensible Processing Stage: In addition to the AQM estimation and packet marking/dropping functionality built into the basic NetBump pipeline, developers can optionally include arbitrary additional packet processing. NetBump developers can include extensions to process packet streams. This API is quite mple, in that the extension is called once per packet, hich is represented by a pointer to the packet data and ngth field. Developers can read, modify, and adjust e packet arbitrarily before re-injecting the packet back into the NetBump pipeline (or dropping it entirely).

Packets destined to particular virtual queues can be forwarded to different extensions, each of which runs in its own thread, coordinating packet reception from the NetBump pipeline through a shared producer-consumer queue. By relying on multi-core processors, each extension can be isolated to run on its own core. This has the advantage that any latency induced by an extension only affects the traffic subject to that extension. Furthermore, correctness or performance bugs in an extension only affects the subset of network traffic enqueued in the virtual queues serving that extension. This enables an incremental "opt-in" experimental platform for introducing new NetBump functionality into the production network.

An advantage of the NetBump architecture is that packets travel a single path from the input port to the output port. Thus, unlike multi-port software routers, here packets can remain entirely on a single core, and stay within a single cache hierarchy. The only point of synchronization is the shared vAQM data structure, and we study the overhead of this synchronization and the resulting lock contention in § 6.2.5.

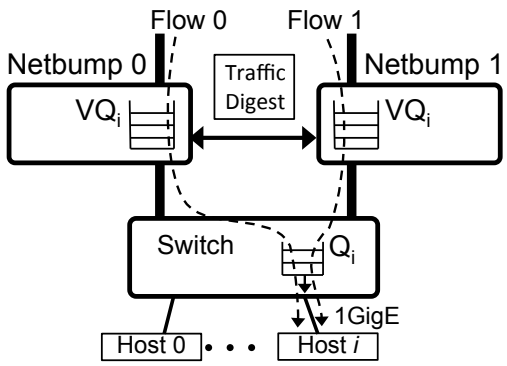


Figure 6: Flow0 and Flow1 both destined to $Host_i$, with two NetBumps monitoring the same Q_i buffer.

3.2 Scaling NetBump

Managing packet flows in multipath environments requires that NetBump scale with the number of links carrying a particular set of flows. This scaling operates within two distinct regions. First, supporting additional links by adding NICs and CPU cores to a single server, and second, through a distributed deployment model.

3.2.1 Multi-link NetBump

For multipath environments in which packets headed to a single destination might travel over multiple paths, it is possible to scale NetBump by simply adding new NICs and CPU cores. For example, a top-of-rack switch with two 10Gbps uplinks would meet these requirements. Here, a single server is only limited in the number of links that it can support by the amount of PCI bandwidth and the number of CPU cores. Each pair of network interfaces supports a single link (10Gbps in, and 10Gbps out), and PCIe gen 2 supports up to three such bi-directional links. In this case, "Multi-link" NetBump is still conceptually simpler than a software-based router, since packets still follow a single-input, single-output path. Each supported link is handled independently inside the bump, and we can assign to it a dedicated CPU core. The only commonality between these links is the vAQM table, which is shared across the links.

3.2.2 Distributed NetBump

For multi-path environments, where NetBumps must be physically separated, or for those with more links than are supported by a single server, we consider a distributed NetBump implementation. Naturally, if multiple NetBumps contribute packets to a shared downstream buffer, they must exchange updates to maintain accurate vAQM estimates. Note that the vAQM table maintains queue estimates for each of neighboring switch's ports (or a monitored subset).

In this case, where we assume the topology (adjacency matrix and link speeds) to be known in advance, NetBumps update their immediate neighbor bumps about the traffic they have processed (Fig. 6). Hence, updates are not the queue estimate itself, but tuples of individual packet lengths and physical downstream switch and port IDs, so that forwarding tables need not be distributed. Each *source* NetBump sends an update to its monitoring neighbors at a given tunable frequency (e.g. per packet, or batched), and each *destination* NetBump calculates a new queue estimate by merging its previous estimate with the traffic update from its neighbor, according to the algorithm in Fig. 5. In this design, updates are tiny; 4B per monitored flow packet (i.e. 2B for packet size and 2B for the port identifier). This translates to about 3MB/s of control traffic per 10Gbps monitored flow. Note also that updates can be transmitted on a dedicated link, or in-band with the monitored traffic. We chose the latter for our Distributed NetBump implementation.

The above technique introduces two possible sources of queue estimation error: 1) batching updates causes estimates to be slightly stale, and since packet sizes are not uniform, the individual packet components of a virtual queue and their respective order would not necessarily be the same, and 2) the propagation delay of the update. Despite this incremental calculation, the estimation naturally synchronizes whenever the buffer occupancy is near its empty/full boundaries.

4. DEPLOYED APPLICATIONS

In this section, we describe the design and implementation of several vAQM and congestion control applications we developed with NetBump.

4.1 Random Early Detection

The first vAQM scheme we implemented is Random Early Detection (RED) [12]. The goal of RED is to keep the average queue length low while achieving high throughput. RED buffer management consists of two parts: estimation of the average queue size using an exponentially-weighed moving average, and a decision on dropping or marking a packet. The packet mark or drop rate increases linearly from zero, when the average queue length is at `MinThresh`, to a maximum probability when the average queue length reaches `MaxThresh`. Our implementation is based on the algorithm proposed in the original RED paper. This vAQM stage maintains `MinThresh` and `MaxThresh` watermarks (with values of 20 and 60 packets, respectively), a w_q setting of 0.1, and varies max_p. We based these values on previous work by Floyd et al. [42].

4.2 Data Center TCP

We next implemented Data Center TCP (DCTCP) [2] on NetBump. The purpose of DCTCP is to improve the behavior of TCP in data center environments, specifically by reducing queue buildup, buffer pressure, and incast. It requires changes to the endhosts as well as network switches. A DCTCP-enabled switch marks the ECN bits of packets when the size of the output buffer in the switch is greater than the marking threshold K. Unlike RED, this marking is based on instantaneous queue size, rather than a smoothed average. The receiver is responsible for signaling back to the sender the particular sequence of marked packets (see [2] for a complete description), and the sender maintains an estimate α of the fraction of marked packets. Unlike a standard sender that cuts the congestion window in half when it receives an ECN-marked acknowledgment, a DCTCP sender reduces its rate according to: $cwnd \leftarrow cwnd*(1-\alpha/2)$. We support DCTCP in the endhosts by using a modified Linux TCP stack supplied by Kabbani and Alizadeh [18].

Implementing DCTCP in NetBump was straightforward, and relied on much of the same code as RED. Here, instead of computing a smoothed queue average of the downstream physical queue occupancy, we mark based on the instantaneous queue size. Next, we set both `LowThresh` and `HighThresh` to the supplied K (chosen to be 20 packets, based on the authors' guidelines [2]). We experimented with other values of K, and found that changing this value had little noticeable effect on aggregate throughput or rate convergence.

4.3 Quantized Congestion Notification

We also implemented the IEEE 802.1Qau-QCN L2 Quantized Congestion Control (QCN) algorithm [1]. QCN-enabled switches monitor their output queue occupancies and upon sensing congestion (using a combination of queue buildup rate and queue occupancy), they send feedback packets to upstream *Reaction Points*. The Reaction Points are then responsible for adjusting the sending rate according to a prescribed formula. For every QCN-enabled link, there are two basic algorithms:

Congestion Point (CP): For every output queue, the switch calculates a *feedback measure* (F_b) whenever a new frame is queued. This measure captures both the *rate* at which the queue is building up (Q_δ), as well as the *difference* (Q_{off}) between the current queue occupancy and a desired equilibrium threshold (Q_{eq}, assumed to be 20% of the physical buffer). If Q denotes the current queue occupancy, Q_{old} is the previous iteration, and w is the weight controlling rate build-up, then:

$$Q_{off} = Q - Q_{eq} \qquad Q_\delta = Q - Q_{old}$$
$$F_b = -(Q_{off} + wQ_\delta)$$

Based on F_b, the switch probabilistically generates a congestion notification frame proportional to the severity of the congestion (the probability profile is similar to RED [12], i.e. it starts from 1% and plateaus at 10% when $|F_b| \geq F_{bmax}$). This QCN frame is destined to the upstream reaction point from which the just-added frame was received. If $F_b \geq 0$, then there is no congestion and no notification is generated.

Reaction Point (RP): Since the network generates signals for rate decreases, QCN senders must probe for available bandwidth gradually until another notification is received. The reaction point algorithm has two phases: Fast-Recovery (FR) and Additive-Increase (AI), similar, but independent from, BIC-TCP's dynamic probing.

The RP algorithm keeps track of the sending Target Rate (TR) and Current Rate (CR). When a congestion control frame is received, the RP algorithm immediately enters the Fast Recovery phase; it sets the target rate to the current rate, and reduces the current rate by an amount proportional to the congestion feedback (by at most 1/2). Barring further congestion notifications, it tries to recover the lost bandwidth by setting the current rate to the average of the current and target rates, once every *cycle* (where a cycle is defined in the base byte-counter model as 100 frames). The RP exits the Fast Recovery phase after five cycles, and enters the Additive Increase phase, where the RP continually probes for more bandwidth by adding a constant increase to its target rate (1.5Mbps in our implementation), and again setting the current sending rate to the average of the CR and TR.

5. IMPLEMENTATION

NetBump can be implemented using a wide variety of underlying technologies, either in hardware or in software. We evaluated three such choices: 1) the stock Linux-based forwarding path, 2) the RouteBricks software router, and 3) a user-level application relying on kernel-bypass network APIs to read and write packets directly to the network. We call this last implementation *UNetBump*. We show in Fig. 7 the latency distributions of these systems when forwarding 1500B packets at 10Gbps (except Linux with 9000B). The baseline for comparison being a simple loopback.

All of our implementations are deployed on HP DL380G6 servers with two Intel E5520 four-core CPUs, each operating at 2.26GHz with 8MB of cache. These servers have 24 GB of DRAM separated into two 12GB banks, operating at a speed of 1066MHz. For the Linux and UNetBump implementations, we use an 8-lane Myricom 10G-PCIE2-8B2-2S+E dual-port 10Gbps NIC which has two SFP+ interfaces, plugged into a PCI-Express Gen 2 bus. For RouteBricks, we used an

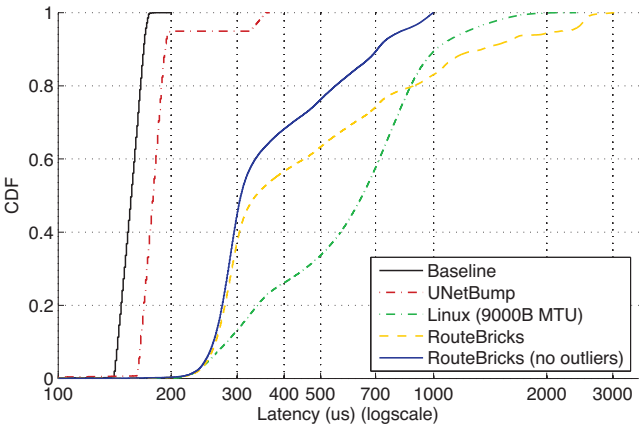


Figure 7: Forwarding latency of baseline, UNetBump, Linux, RouteBricks (batching factor of 16, and a Click burst factor of 16), both with and without an outlier queue.

Intel E10G42AFDA dual-port 10Gbps NIC (using an 82598EB controller) with two SFP+ interfaces.

5.1 Linux

The Linux kernel natively supports a complete IP forwarding path, including a configurable set of queuing disciplines that are managed through the "traffic control (tc)" extensions [26]. Linux **`tc`** supports flow and packet shaping, scheduling, policing, and dropping. While **`tc`** supports a variety of queuing disciplines, it does not support managing the queues of remote switches. This support would have to be added to the kernel. In our evaluation we used Linux kernel version 2.6.32. We found that the latency overheads of the Linux forwarding path were very high, with a mean latency above $500\mu s$, and a 99th percentile above $1500\mu s$. Furthermore, our evaluation found that Linux was not able to forward non-Jumbo frames at speeds approaching 10Gbps (and certainly not with minimum-sized packets). Based on these microbenchmarks, we decided not to further consider Linux as an implementation alternative.

5.2 RouteBricks

RouteBricks [8] is a high-throughput software router implementation built using Click's core, extensive element library, and specification language. It increases the scalability of Click in two ways–by improving the forwarding rate within a single server, and by federating a set of servers to support throughputs beyond the capabilities of a single server. To improve the scalability within a single server, RouteBricks relies on a re-architected NIC driver that supports multiple queues per physical interface. This enables multiple cores to read and write packets from the NIC without imposing lock contention, which greatly improves performance [7, 9, 31, 53]. Currently, RouteBricks works only with the **`ixgbe`** device driver, which delivers packets out of the driver in fixed-size batches of 16 packets each. We built a single-node RouteBricks server using the same HP server architecture described above, but with the Intel E10G42AFDA NIC (the only available NIC that RouteBricks driver patch still supported). This server used the Intel **`ixgbe`** driver (version 1.3.56.5), with a batching factor of 16.

The use of this batching driver improves throughput by amortizing the overhead of transferring those packets over an entire batch, rather than on a packet-by-packet basis. This enables RouteBricks to support very high line rates, however the use of batching increases latency on an individual packet basis.[2] We also tried a batching factor of 1 and found that the throughput dropped below 10Gbps. Indeed RouteBricks was designed for high throughput, not low-latency. There is nothing in the Click or RouteBricks model that precludes low-latency forwarding, however for this work we chose not to use RouteBricks.

5.3 UNetBump

In user-level networking, instead of having the kernel deliver and demultiplex packets, the NIC instead delivers those packets directly to the application. This is typically coupled with kernel-bypass support, which enables the NIC to directly copy packets into a memory region mapped to the application. User-level networking has been further developed to better support virtualization by enabling individual flows to bypass the hypervisor and terminate directly in a guest VM [27].

User-level networking is a well-studied approach that has been implemented in a number of commercially-available products. Myricom offers Ethernet NICs with user-level networking APIs that we use in our evaluation. Intel supports user-level, kernel-bypass networking via the PF_RING/DNA driver [41]. SolarFlare offers a set of "Solarstorm" NICs with this functionality [48], as does SMC [47]. There have been at least two efforts to create an open and cross-vendor API to user-level, kernel-bypass network APIs [39, 43]. In this paper, we re-evaluate the use of user-level networking to support low-latency applications, especially those requiring low latency variation. Note that it is possible to layer the RouteBricks/Click runtime on top of the user-level, kernel-bypass APIs we use in UNetBump.

6. EVALUATION

Our evaluation seeks to answer the following questions: 1) How expressive is NetBump? 2) How easy is it to deploy applications? 3) How effective is vAQM es-

[2]Despite extensive debugging with the help of the RouteBricks authors, we could not lower this latency further.

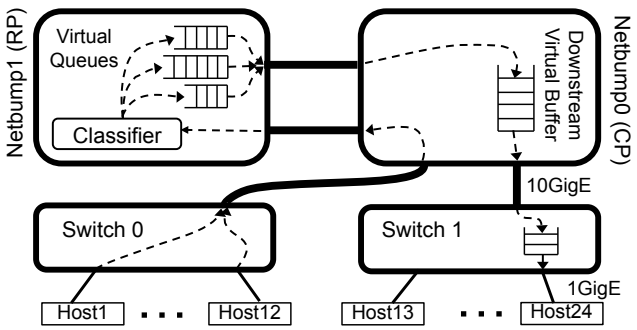


Figure 8: Two-rack 802.1Qau-QCN and DCTCP Testbed

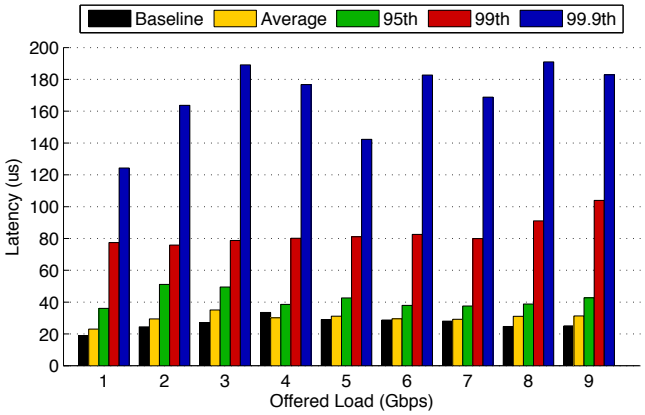


Figure 9: Latency percentiles imposed by UNetBump vs. offered load. Baseline is the loopback measurement overhead.

timation in practice? 4) What are the latency overheads and throughput limitations of NetBump?

To answer these, we built and deployed a set of NetBump prototypes in our experimental testbed. We started by evaluating the baseline latency and latency variation of the range of implementation choices. Based on these measurements, we proceeded with construction of UNetBump, a fully-functional prototype based on user-level networking APIs. We then evaluate a range of AQM functionalities with UNetBump.

6.1 Testbed Environment

Our experimental testbed consists of a set dual-processor Nehalem server described above, using either Myricom NICs, or in the case of RouteBricks, the Intel NIC. The Myricom NICs use the Sniffer10G driver version 1.1.0b3. We use copper direct-attach SFP+ connectors to interconnect the 10Gbps endhosts to our NetBumps. Experiments with 1Gbps endhosts rely on a pair of SMC 8748L2 switches that each have 1.5MB of shared buffering across all ports. Each SMC switch has a 10Gbps uplink that we connect to the appropriate NetBump.

We evaluate NetBump in three different contexts. The first is in microbenchmark, to examine its throughput and latency characteristics. Here we deploy NetBump as a loopback (simply connecting the two ports to the same host) to eliminate the effects of clock skew and synchronization. The second simply puts a NetBump inline between two machines, and tests NetBump's operation at full 10Gbps. Separating the source and destination to different machines enables throughput measurement with real traffic.

The third testbed, Fig. 8, evaluates NetBump in a realistic data center environment in which it might be deployed right above the top-of-rack switch. Here, we have two twelve-node racks of endhosts, each connected to a 1Gbps switch. A 10Gbps uplink connects the two 1Gbps switches and the NetBump is deployed inline with those uplinks. In this case, the NetBump actually has four 10Gbps interfaces–two to the uplinks of each of the two SMC 1Gbps switches, and two that connect to a second NetBump. We use this testbed to evaluate 802.1Qau-QCN, with one NetBump acting as the Congestion Point (CP) and the other as the Reaction Point (RP).

6.2 Microbenchmarks

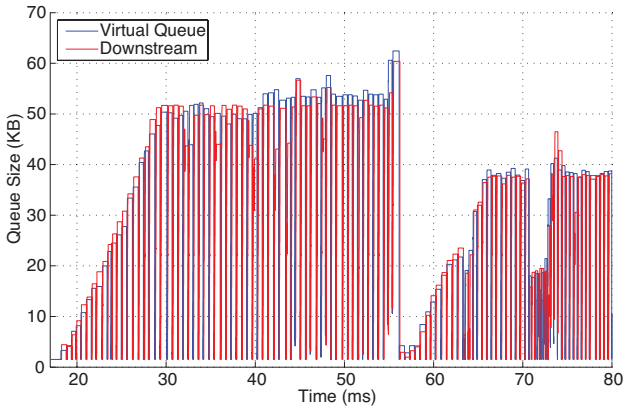
6.2.1 *NetBump Latency*

A key metric for evaluation is the latency overhead. To measure this, we use a loopback testbed and had a packet generator on the client host send packets onto the wire, through the NetBump, and back to itself. To calibrate, we also replace the NetBump with a simple loopback wire, which gives us the baseline latency overhead of the measurement host itself. We subtract this latency from the observed latency with the NetBump in place, giving us the latency of just the NetBump. We generated a constant stream of 1500-byte packets sent at configurable rates (Fig. 9).

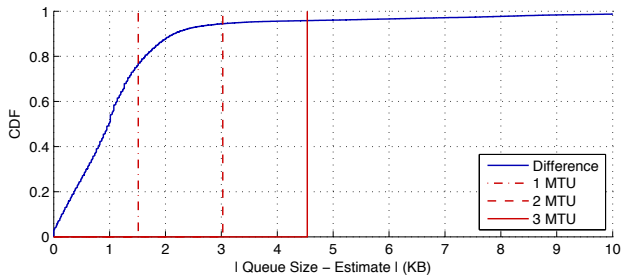
For UNetBump, the latency is quite low for the majority of forwarded packets. There is a jump in latency at the tail due to NIC packet batching when they arrive above a certain rate. There is no way to disable this batching in software, even though we were only using a single CPU core which could have serviced a higher packet rate without requiring batching. The forwarding performance of UNetBump was sufficient to keep up with line rate using minimum-sized packets and a single CPU core.

6.2.2 *vAQM Estimation Accuracy*

To evaluate the accuracy of the vAQM estimation, we ran **`iperf`** sessions between two hosts, connected in series by a NetBump and another pass-through machine (which records the timestamps of incoming frames). Since we cannot export physical buffer occupancy of



(a) Virtual queue size vs. actual downstream queue. Running an **iperf** TCP session between two 10G hosts, rate-limited to 1Gbps with a 40KB buffer downstream to induce congestion.



(b) CDF of the queue size difference. The estimate is within two 1500B MTUs 95% of the time.

Figure 10: Downstream vAQM estimation accuracy.

commercial switches, we use the frame timestamps and lengths from the downstream pass-through machine to recreate the output buffer size over time, knowing the drain-rate. Fig. 10 shows the NetBump virtual queue size vs. the actual downstream queue. The estimate was within two MTUs 95% of the time.

6.2.3 *Distributed NetBump*

We also measured the accuracy of queue estimation when multiple NetBumps exchange updates to estimate a common downstream queue. In the first experiment, measure the effect of update latency on queue estimate accuracy. We varied the timestamp interleaving of two TCP **iperf** flows that share a downstream queue in order to simulate receiving delayed updates from a neighboring NetBump. Fig. 11 shows the CDF of the difference between the delayed inter-bump estimation and the in-sync version; Even when update latency was 25μs, the difference was always under 2MTU.

Next, we show the accuracy of NetBump's queue estimating of a downstream queue, based solely on updates from its neighbor. In our implementation, the updates are transmitted in-band with the monitored

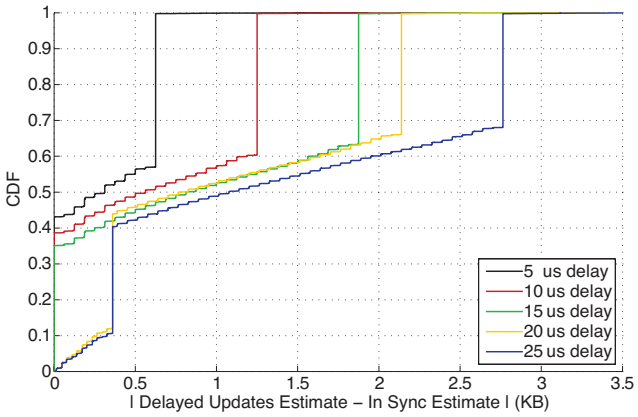


Figure 11: CDF of the absolute difference between the queue estimate with delayed updates and the in-sync version. The combined throughput is rate-limited to 5Gbps, and the downstream buffer is 40KB.

traffic. Fig. 12 gives the CDF of the difference between the actual queue size and the distributed NetBump estimate. We observe that the estimate is within 3MTUs 90% of the time. Note, however, the effect of update batching: estimates quickly drift when updates are delayed. Fig. 13 shows a typical relative difference CDF when background elephant flows are present (i.e. some flows are observed directly, and others indirectly through updates).

6.2.4 *Effect of assigning CPU affinity*

One of the challenges of designing NetBump was not only maintaining a low average latency, but also reducing variance. Modern CPU architectures provide separate cores on the same die and physically separate memory across multiple Non-Uniform Memory Access (NUMA) banks. This means that access time to memory banks changes based on which core issues a given request. To reduce latency outliers, we allocated memory to each UNetBump thread from the same NUMA domain as the CPU core it was scheduled to.

Given the significant additional latency that may be introduced by the unmodified Linux kernel scheduler, we compare latency of NetBump with and without CPU-affinity and scheduler modifications. Our control experiment uses default scheduling. To improve on this, we exclude all but one of the CPU cores from the default scheduler, and ensure that the UNetBump user-space programs execute on the reserved cores. We then examined the average, 95th, 99th, 99.9th, and maximum latencies through NetBump compared to the baseline (Table 2). CPU-affinity had a minor effect on latency on average, but was most pronounced on outlier packets. The maximum observed latency was 17 times smaller with CPU-affinity at the 99.9th

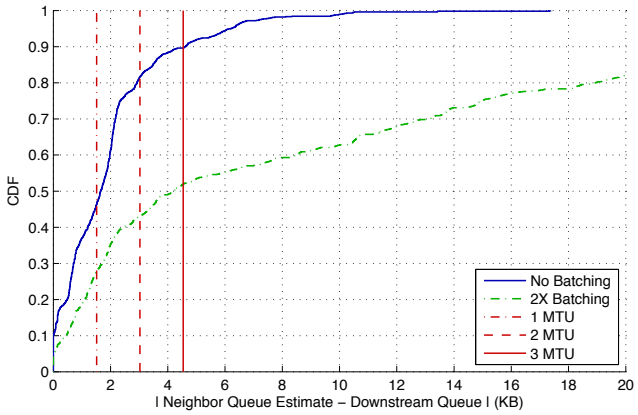


Figure 12: CDF of the difference between actual queue size and the Distributed NetBump estimate using a 1Gbps rate-limited TCP flow and a 40KB buffer. The estimating NetBump does not observe the monitored traffic directly.

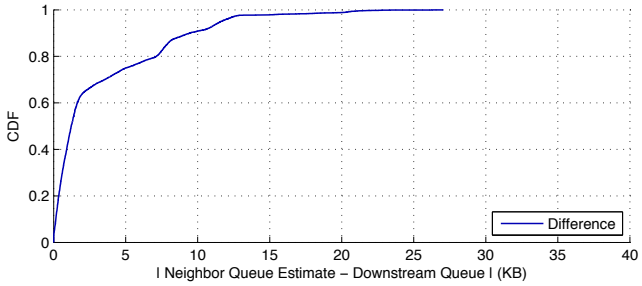


Figure 13: Typical distributed NetBump relative error with background elephant flows.

percentile, showing the importance of explicit resource isolation in low-latency deployments.

6.2.5 *Multicore performance*

In UNetBump, basic vAQM estimation can be done at 10Gbps using only a single CPU core. However, to support higher link rates, additional cores might be necessary. The NIC itself will partition flows across CPU cores using a hardware hash function. In this scenario, a user-space thread would be responsible for handling each ring pair, and the only time these threads must synchronize would be when updating the vAQM state table. To evaluate the effect of this synchronization on the latency of NetBump in a multi-threaded implementation, we examined the effect of vAQM table lock overhead. As a baseline, a single-threaded forwarding pipeline (FP) has a latency of 29.16μs. Running NetBump with two FPs (two ring pairs in the NIC and each FP running on its own core) increased that latency by 17.9% to 35.5μs. Further running NetBump with four FPs on four cores increased the latency by an additional 1.95% to 36.8μs. Thus we find that the synchronization

Latency (μs)	Avg	95th	99th	99.9th	Max
No Affinity	32	39	76	1,322	3,630
With Affinity	30	42	83	169	208

Table 2: UNetBump latency percentiles vs. CPU core affinity.

Application	# lines of code
NetBump core	940
RED	29
DCTCP	29
QCN	464

Table 3: Coding effort for NetBump and its applications.

overhead is minimal to gain back a four-fold increase in computation per packet, or alternatively, a four-fold increase in supported line rate. A key observation is that NetBump avoids some of the required synchronization overheads found in software routers [7, 9, 32] with multiple ports, since in NetBump each input port only forwards to a single output port, preventing packets from spanning cores or causing contention on shared output ports.

6.3 Deployed Applications

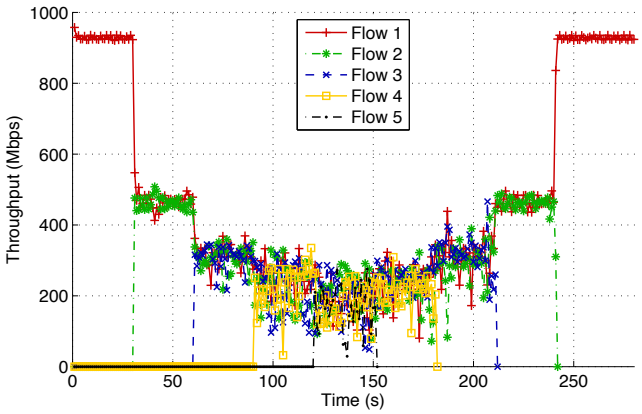
One metric highlighting the ease of writing new applications with NetBump is shown in Table 3. Most of our applications took only 10s of lines of code, and QCN, which is much more complex, was written in less than 500 lines of code. The time commitment ranged from hours to a couple of days in the case of QCN. We now examine each application in detail.

6.3.1 *Random Early Detection*

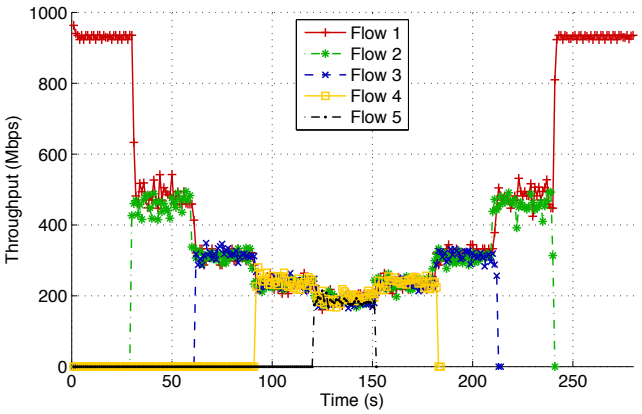
Fig. 14 shows how deploying RED lowers the downstream buffer occupancy for a set of flows. Here three alternatives are compared: RED with two max_p parameter values, as well as the baseline drop-tail queuing discipline. One observation was that experimenting with different RED parameters was very easy with NetBump, and thus we could rapidly explore its parameter space by simply providing different arguments to our NetBump application's command line.

6.3.2 *Data Center TCP*

The next experiment represents a recreation of the DCTCP convergence test presented by Alizadeh et al. [2] performed in our two-rack testbed (see Fig. 8). Five source nodes each open a TCP connection to one of five destination nodes in 25 second intervals. In the baseline TCP case (Fig. 15(a)), due to buffer pressure and a drop-tail queuing discipline, the bandwidth



(a) Baseline TCP (CUBIC)



(b) DCTCP

Figure 15: The effect on fairness and convergence of DCTCP on five flows sharing a bottleneck link.

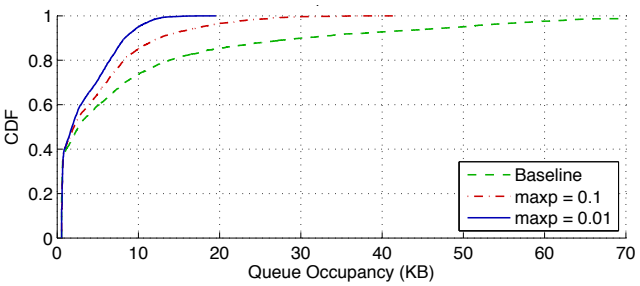


Figure 14: Evaluation of RED with two max_p parameter settings, showing the effect of reduced buffering given a higher marking probability.

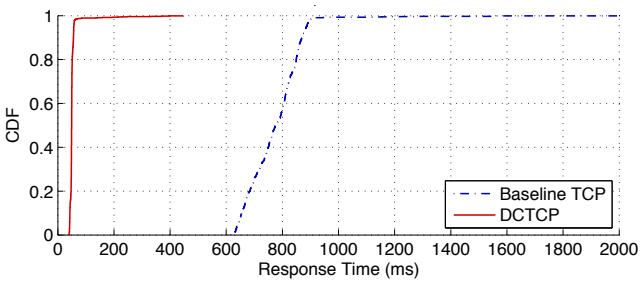


Figure 16: Baseline TCP (CUBIC) and DCTCP response times for short RPC-type flows in the presence of background elephant flows.

is shared unfairly, resulting in a wide oscillation of throughput and unfair sending rate among the flows. Fig. 15(b) shows the throughput of DCTCP-enabled endpoints and a DCTCP vAQM strategy in the NetBump. Like in the original DCTCP work, here the fair sharing of network bandwidth results from the lower queue utilization afforded by senders appropriately backing off in response to NetBump-set ECN signals.

Another contribution of reduced queue buildup is better support for mixtures of latency-sensitive and long-lived flows. Fig. 16 shows the CDF of response time for 10,000 RPC-type requests in the presence of two large elephant flows, comparing stock TCP endpoints without NetBump DCTCP support. This figure recreates a key DCTCP result: signaling the long flows to reduce their rates results in smaller queues, lower RTT, and in the end, shorter response times.

6.3.3 *Quantized Congestion Notification*

Another example of how the NetBump programming model enabled easy and rapid prototyping and evaluation of new protocols was deploying 802.1Qau-QCN. Our implementation of QCN is 464 lines of code, and it took around 2-3 days to write and debug. Developing QCN within NetBump enabled us to easily tune parameters and evaluate their effect. This was especially important given QCN's novelty, and the lack of other tools or simulations we could have used to study it. Using the testbed topology of Fig. 8, we use NetBump0 as the CP, and NetBump1 as the RP. In our RP, we chose a virtual queue size of 100KB (and Q_{eq} at 20KB).

Through our evaluation, we found that the feedback control loop tends to be more stable when the frequency of feedback messages is higher and their effect smaller. For this reason, we use $F_{bmax} = 32$, and plateau the probability profile at 20%. We also found that due to the burstiness of the packet arrival rate, we had to decrease w to 1 to avoid unnecessary rate drops. Our implementation also needed to consider the relative flow weights in the entire queue when choosing which flow to rate limit, rather than just using the current packet. We use the byte counter-only model of RP in our implementation. For the Additive Increase phase, we use cycles of 100 packets, and an increase of 1.5Mbps (to exaggerate and show the convergence of the virtual port current rates), and 600 packet cycles for the Fast Recovery phase.

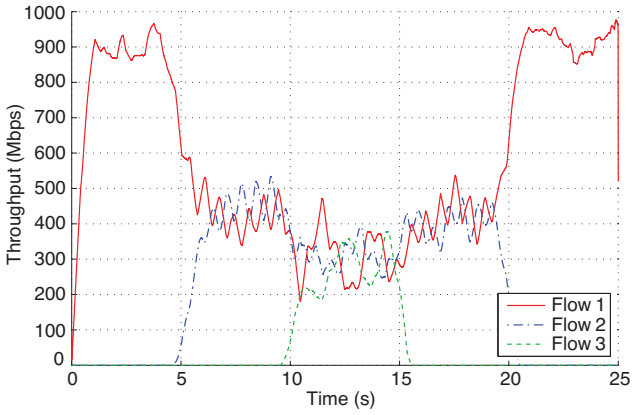


Figure 17: QCN with three 1Gbps UDP flows. With QCN enabled, the RP virtual queue occupancy never exceeded 40%, as opposed to persistent drops downstream without.

The throughput of three 1Gbps UDP flows sharing the same bottleneck link is shown in Fig. 17. Without QCN, the downstream buffer would be persistently overwhelmed by the three UDP flows from 5-20s, but with QCN enabled, congestion is pushed upstream and the virtual queue occupancy never exceeded 40%, thereby preventing drops for potential mice flows.

6.4 Evaluation Summary

In this section we have described several vAQM and congestion control applications built with NetBump. We found that even though some had been extensively studied in the literature (e.g. RED), finding the particular parameters to make them work well in our network required several attempts. NetBump simplified this design–deploy–evaluate loop. Furthermore, in the case of QCN, there was little experience available due to its novelty and lack of deployments. The ability to develop our implementation in software, while testing it with real traffic, proved extremely useful.

7. RELATED WORK

This section describes previous work that we build upon to design and implement NetBump.

Virtual Queuing and AQM: In virtual queuing (VQ), metadata about an incoming packet stream is maintained to simulate the behavior of those packets in a hypothetical physical queue. We differ from previous work in that we maintain VQs outside of the switch itself. VQ provides a basis for a variety of active queue management (AQM) techniques. AQM manipulates packets in buffers in the network to enact changes in the control loop of that traffic, typically to anticipate and reduce packet drops and queue overflows, or reduce buffer sizes. A large amount of work examined AQM in a variety of settings [17, 34]. One proposal, Active Virtual Queue [24], reduces queue sizes in traffic with small flows, which typically pose challenges for the TCP control loop. Random early detection (RED) [16] signals congestion by dropping packets with a particular probability as congestion builds, and unconditionally dropping packets after a certain threshold. Due to the inefficiency of dropping packets to signal congestion, the early congestion notification ECN [25] field was developed to decouple packet drops from congestion indicators. Several proposals for improving on RED have been made [4], including Data Center TCP (DCTCP) [2].

Quantized congestion notification [1,30] was proposed as an L2 congestion control mechanism. QCN tries to ensure that a switch buffer stays below a configurable maximum size. QCN provides congestion control for non-TCP traffic, and can respond faster than the round-trip time. Implementations of QCN have been developed on 1Gbps networks [29], as well as emulated within FPGAs at 10Gbps networks [37,38]. Our deployment is done at 10Gbps and distributed across multiple network hops. Approximate-Fairness QCN (AF-QCN) [19] is an extension that modifies notifications to input links weighted by the ratio of their queue occupancy.

Datapath Programming in Software: The Click Modular Router [23] is a pipeline-oriented, modular software router consisting of a large number of building blocks, each performing a simple packet-handling task. Click's library of modules can be extended by writing code in C++ designed to work in the Linux kernel or userspace. The RouteBricks [8] project has focused on scaling out a Click runtime to support forwarding rates in excess of tens of Gbps by relying on distribution of packet processing across cores, as well as across a small cluster of servers. ServerSwitch [29], is another recent software router design that allows programming commodity Ethernet switching chips (with matching/modification of standard header fields), but delegates general packet processing to the CPU (e.g. for XCP). Besides avoiding the associated latency of crossing the kernel/user-space boundary, NetBump leverages kernel-bypass to allow *arbitrary* packet modification to support new protocol headers at line rate. A key distinction from these projects is that Click, RouteBricks and ServerSwitch are all multi-port software switches focused on packet routing, while NetBump focuses on pass-through virtual queuing within a pre-existing switching layer.

SideCar [46], on the other hand, is a recent proposal to delegate a small fraction of traffic requiring special processing from the ToR switch to a companion server. While superficially similar, the redirection and traffic sampling are not applicable for NetBump's vAQM use-case, where low-latency is a key design requirement. For

these reasons, we consider these efforts to be orthogonal to this work.

Several efforts have looked at ways of mapping packet handling tasks necessary to support software routers to multi-core, multi-NIC queue commodity servers. Egi et al. [9], and Dobrescu et al. [7] investigate the effects of casting forwarding paths across multiple cores, and find that minimizing core transitions is necessary for high performance. NetBump takes a similar approach to the "split traffic" and "cloning" functionality described, in which an entire forwarding path resides on a single core and cache hierarchy. Manesh et al. [32] study the performance of multi-queue NICs as applied to packet forwarding workloads.

Typically the OS kernel translates streams of raw packets to and from a higher-level interface such as a socket. While a useful primitive, the involvement of the kernel can become a bottleneck and an alternative set of user-level networking techniques have been developed [5, 10, 50, 52]. Here, user-space programs are responsible for TCP sequence reassembly, retransmission, etc. User-level networking is typically coupled with zero-copy buffering, in which the memory that a packet is initially stored in is shared with target applications. Kernel-bypass drivers also enable applications to directly access packets from NIC memory, avoiding kernel involvement on the datapath. Commercially-available NICs already support these mechanisms [6,35,41,47,48].

Datapath Programming in Hardware: Perhaps the best-known and widely-used hardware forwarding platform is the NetFPGA [28, 36], a powerful development tool for FPGA devices. However, the complexity of FPGA programming remains a challenge. Two recent projects sought to address this: Switchblade [3] provides modular building blocks that can support a wide variety of datapaths, and Chimpp [44] converts datapaths specified in the Click language into Verilog code suitable for an FPGA.

In addition, network processors (NPs) [45] have been used to prototype and deploy new network functionality. They have the disadvantage of a difficult-to-use programming model and limited production runs. Their primary advantage is their multiple functional units, providing significant parallelism to support faster data rates. Commodity CPUs have since greatly increased their number of cores, and can also provide significant per-packet processing at high line rates.

8. CONCLUSIONS

In this work, we presented NetBump, a platform for developing, experimenting with, and deploying alternative packet buffering and queuing disciplines with minimal intrusiveness and at low latency. NetBump leaves existing switches and endhosts unmodified. It acts as a "bump on the wire", examining, optionally modifying, and forwarding packets at line rate in tens of microseconds to implement a variety of virtual active queuing disciplines and congestion control protocols implemented in user-space. We built and deployed several applications with NetBump, including DCTCP and 802.1Qau-QCN. These applications were quickly developed in hours or days, and required only tens or hundreds of lines of code in total.

A major barrier to developing and deploying new network functionality is the difficulty of programming the network datapath. In this work we evaluate NetBump as deployed on a variety of software-programmable systems, including Linux and Click/RouteBricks, and a user-level, kernel-bypass networking networking API. We found this latter implementation choice, despite being the oldest, provided the lowest-latency performance, supporting line-rate forwarding of minimum-sized packets at 9.5Gbps across each of these applications. The adoption of multi-core processors, along with kernel-bypass commodity NICs, provides a feasible platform to deploy data modifications written in user-space at line rate. Our experience has shown that NetBump is a useful and practical platform for prototyping and deploying new network functionality in real data center environments.

9. REFERENCES

[1] M. Alizadeh, B. Atikoglu, A. Kabbani, A. Lakshmikantha, R. Pan, B. Prabhakar, and M. Seaman. Data center transport mechanisms: Congestion control theory and ieee standardization. In *Allerton CCC, 2008*.

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

[3] M. B. Anwer, M. Motiwala, M. b. Tariq, and N. Feamster. Switchblade: A platform for rapid deployment of network protocols on programmable hardware. In *ACM SIGCOMM 2010*.

[4] J. Aweya, M. Ouellette, D. Y. Montuno, and K. Felske. Rate-based proportional-integral control scheme for active queue management. *IJNM*, 16, 2006.

[5] P. Buonadonna, A. Geweke, and D. Culler. An implementation and analysis of the virtual interface architecture. In *ACM/IEEE CDROM 1998*.

[6] Chelsio Network Interface. `http://www.chelsio.com`.

[7] M. Dobrescu, K. Argyraki, G. Iannaccone, M. Manesh, and S. Ratnasamy. Controlling parallelism in a multicore software router. In *ACM Presto 2010*.

[8] M. Dobrescu, N. Egi, K. Argyraki, B.-G. Chun, K. Fall, G. Iannaccone, A. Knies, M. Manesh, and S. Ratnasamy. Routebricks: exploiting parallelism to scale software routers. In *ACM SOSP 2009*.

[9] N. Egi, A. Greenhalgh, M. Handley, M. Hoerdt, F. Huici, L. Mathy, and P. Papadimitriou. Forwarding path architectures for multicore software routers. In *ACM Presto 2010*.

[10] D. Ely, S. Savage, and D. Wetherall. Alpine: a user-level infrastructure for network protocol development. In *USITS 2001*.

[11] S. Floyd. Tcp and explicit congestion notification. *SIGCOMM Comput. Commun. Rev.*, 24:8–23, October 1994.

[12] S. Floyd and V. Jacobson. Random early detection gateways for congestion avoidance. *IEEE/ACM TON 1993*.

[13] R. J. Gibbens and F. Kelly. Distributed connection acceptance control for a connectionless network. In *Teletraffic Engineering in a Competitive World*. Elsevier, 1999.

[14] R. J. Gibbens and F. Kelly. Resource pricing and the evolution of congestion control. In *Automatica 35*, 1999.

[15] N. Gude, T. Koponen, J. Pettit, B. Pfaff, M. Casado, N. McKeown, and S. Shenker. Nox: towards an operating system for networks. *SIGCOMM CCR 2008*.

[16] C. Hollot, V. Misra, D. Towsley, and W.-B. Gong. A control theoretic analysis of RED. In *INFOCOM 2001*.

[17] C. V. Hollot, V. Misra, D. Towsley, and W. Gong. On designing improved controllers for AQM routers supporting TCP flows. Technical report, Amherst, MA, USA, 2000.

[18] A. Kabbani and M. Alizadeh. Personal communication, 2011.

[19] A. Kabbani, M. Alizadeh, M. Yasuda, R. Pan, and B. Prabhakar. Af-qcn: Approximate fairness with quantized congestion notification for multi-tenanted data centers. *IEEE Hot Interconnects 2010*.

[20] S. Karandikar, S. Kalyanaraman, P. Bagal, and B. Packer. TCP rate control. In *ACM SIGCOMM 2000*.

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

[22] A. D. Keromytis and J. L. Wright. Transparent network security policy enforcement. In *USENIX ATC 2000*.

[23] E. Kohler, R. Morris, B. Chen, J. Jannotti, and M. F. Kaashoek. The click modular router. *ACM ToCS*, 2000.

[24] S. Kunniyur and R. Srikant. An adaptive virtual queue (AVQ) algorithm for active queue management. *IEEE/ACM TON 2004*.

[25] A. Kuzmanovic. The power of explicit congestion notification. In *ACM SIGCOMM 2005*.

[26] Linux Traffic Control howto. `http://tldp.org/HOWTO/Traffic-Control-HOWTO`.

[27] J. Liu, W. Huang, B. Abali, and D. Panda. High Performance VMM-Bypass I/O in Virtual Machines. In *USENIX ATC 2006*.

[28] J. Lockwood, N. McKeown, G. Watson, G. Gibb, P. Hartke, J. Naous, R. Raghuraman, and J. Luo. Netfpga–an open platform for gigabit-rate network switching and routing. In *IEEE MSE 2007*.

[29] G. Lu, C. Guo, Y. Li, Z. Zhou, T. Yuan, H. Wu, Y. Xiong, R. Gao, and Y. Zhang. Serverswitch: a programmable and high performance platform for data center networks. In *USENIX NSDI 2011*.

[30] Y. Lu, R. Pan, B. Prabhakar, D. Bergamasco, V. Alaria, and A. Baldini. Congestion control in networks with no congestion drops. *Allerton CCC 2006*.

[31] M. Manesh, K. Argyraki, M. Dobrescu, N. Egi, K. Fall, G. Iannaccone, E. Kohler, and S. Ratnasamy. Evaluating the suitability of server network cards for software routers. In *ACM Presto 2010*.

[32] M. Manesh, K. Argyraki, M. Dobrescu, N. Egi, K. Fall, G. Iannaccone, E. Kohler, and S. Ratnasamy. Evaluating the suitability of server network cards for software routers. In *ACM PRESTO 2010*.

[33] N. McKeown, T. Anderson, H. Balakrishnan, G. Parulkar, L. Peterson, J. Rexford, S. Shenker, and J. Turner. Openflow: enabling innovation in campus networks. *SIGCOMM CCR 2008*.

[34] V. Misra, W.-B. Gong, and D. Towsley. Fluid-based analysis of a network of aqm routers supporting TCP flows with an application to RED. In *ACM SIGCOMM 2000*.

[35] Myricom Sniffer10G. `http://www.myricom.com/support/downloads/sniffer.html`.

[36] J. Naous, G. Gibb, S. Bolouki, and N. McKeown. NetFPGA: reusable router architecture for experimental research. In *ACM PRESTO 2008*.

[37] NEC/Stanford: 10G QCN Implementation on Hardware. `http://www.ieee802.org/1/files/public/docs2009/au-yasuda-10G-QCN-Implementation-1109.pdf`.

[38] 10G QCN Implementation on Hardware. `http://fif.kr/AsiaNetFPGAws/slide/1-4_slide.pdf`.

[39] Openonload. `http://www.openonload.org`.

[40] J. Padhye, V. Firoiu, D. Towsley, and J. Kurose. Modeling tcp throughput: a simple model and its empirical validation. In *ACM SIGCOMM 1998*.

[41] PF_RING Direct NIC Access. `http://www.ntop.org/products/pf_ring/dna/`.

[42] Redparameters.txt. `http://icir.org/floyd/REDparameters.txt`.

[43] L. Rizzo and M. Landi. netmap: memory mapped access to network devices. In *ACM SIGCOMM 2011*.

[44] E. Rubow, R. McGeer, J. Mogul, and A. Vahdat. Chimpp: a click-based programming and simulation environment for reconfigurable networking hardware. In *ACM/IEEE ANCS 2010*.

[45] N. Shah. Understanding network processors. Master's thesis, University of California, Berkeley, Calif., 2001.

[46] A. Shieh, S. Kandula, and E. G. Sirer. Sidecar: building programmable datacenter networks without programmable switches. In *ACM Hotnets 2010*.

[47] Smc smc10gpcie-10bt network adapter. `http://www.smc.com/files/AY/DS_SMC10GPCIe-10BT.pdf`.

[48] SolarFlare Solarstorm Network Adapters. `http://www.solarflare.com/Enterprise-10GbE-Adapters`.

[49] J. Turner. New directions in communications (or which way to the information age?). *Communications Magazine, IEEE*, 2002.

[50] T. von Eicken, A. Basu, V. Buch, and W. Vogels. U-net: a user-level network interface for parallel and distributed computing. In *ACM SOSP 1995*.

[51] Z. Wang. *Internet QoS: Architectures and Mechanisms for Quality of Service*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1st edition, 2001.

[52] M. Welsh, A. Basu, and T. von Eicken. ATM and fast ethernet network interfaces for user-level communication. *IEEE HPCA 1997*.

[53] Q. Wu, D. J. Mampilly, and T. Wolf. Distributed runtime load-balancing for software routers on homogeneous many-core processors. In *ACM Presto 2010*.