UNIVERSITY OF CALIFORNIA, SAN DIEGO

Packet Pacer: An application over NetBump

A thesis submitted in partial satisfaction of the
requirements for the degree
Master of Science

in

Computer Engineering

by

Sambit Kumar Das

Committee in charge:

Professor Amin Vahdat, Chair
Professor George Papen, Co-Chair
Professor Yeshaiahu Fainman
Professor Bill Lin

2011

DEDICATION

Dedicated to my parents and grandparents ...

EPIGRAPH

Your time is limited, so don't waste it living someone else's life. Don't be trapped by dogma - which is living with the results of other people's thinking. Don't let the noise of other's opinions drown out your own inner voice. And most important, have the courage to follow your heart and intuition. They somehow already know what you truly want to become. Everything else is secondary.

-Steve Jobs, Stanford University commencement address, June 12, 2005.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

ACKNOWLEDGEMENTS

First and foremost, I would like to express my sincere gratitude to my advisor Prof. Amin Vahdat for his supervision, advice and guidance from the very early stage of my journey through the M.S. program, as well as giving me extraordinary experience throughout my stay. Above all and the most needed, he provided me unflinching encouragement and support in various ways. I am indebted to him more than he knows.

I would like to thank my teammates in this research - George Porter, Rishi Kapoor and Mohammad Al-Fares. Each one has made significant contribution to the "NetBump" project. Without the advice I received from them - in the form of awesome ideas from George to design discussions with Rishi to details of the QCN implementation from Mohammad - this thesis would not be in its current stage.

I would like to thank Nathan Farrington, who was my first mentor when I joined the team. Working with him was an amazing experience and gave me an ideal launching ground into the research world. My acknowledgements also goes to the rest of the members of DCSwitch group. It was a great learning experience to interact and work with each one of them.

Last but not the least, I would like to thank all the distinguished faculty at UC San Diego. It was truly a great research experience and I have gained a wealth of knowledge. The graduate program blended with high quality reseach with real world impact has always motivated me to innovate and I thank everyone for that.

VITA

2003-2007	Bachelor of Technology in Electronics and Communication Engineering, National Institute of Technology, Rourkela, India.
2006	Summer Research Intern in the Department of Electrical Engineering, Indian Institute of Technology, Madras, India.
2007-2009	Sr. Subject Matter Expert, Product Development Business Unit, Amdocs Development Ltd., Limassol, Cyprus.
2009-2011	Master of Science in Computer Engineering, Department of Electrical and Computer Engineering, University of California, San Diego.
2010	Research Intern, Packet Processing, Ericsson Research, San Jose, CA.
2009-2011	Graduate Student Researcher/Tutor, University of California, San Diego.

PUBLICATIONS

Porter, George; Kapoor, Rishi; Das, Sambit K.; Al-Fares, Mohammad; Weatherspoon, Hakim; Prabhakar, Balaji; Vahdat, Amin. "User-extensible Active Queue Management with Bumps on the Wire", Submitted for SOSP.2011.

G. Chandramowli, Buvaneshwari M., Alpesh Chaddha, Sambit Das and R.Manivasakan "Simulation and Performance of RPR (Resilient Packet Ring- IEEE 802.17) - MAC in Network Simulator."- Proceedings of the eighth international conference, Photonics 2006, December 13-16, 2006, Hyderabad, India.

ABSTRACT OF THE THESIS

Packet Pacer: An application over NetBump

by

Sambit Kumar Das

Master of Science in Computer Engineering

University of California, San Diego, 2011

Professor Amin Vahdat, Chair
Professor George Papen, Co-Chair

Many ideas for adding innovative functionalities to networks require either modifying packets or performing alternative queuing to packets in flight on the data-plane. Modifications to existing network is difficult and often faced with hindrance like ease of deployability and ability to test with production-like traffic. NetBump is a platform for experimenting, evaluating and deploying these ideas with minimal intrusiveness, while leaving the switches and endhosts unmodified.

In this thesis, we evaluate TCP packet pacing as an application over NetBump. We propose a modified token bucket implementation which accurately estimates the token requirement for sustained well-paced TCP flow, thus smoothing the bursty behavior. We were able to able to monitor various crucial features of the TCP flow and take informed decisions to pace the out going flow. Finally, we perform micro-benchmarks to see the effect of pacing. In general, our packet pacer implementations reduces the number of buffer overflows. Specifically, the modified token bucket implementation performs the best with zero buffer overflows.

Chapter 1

Introduction

1.1 Data Center Networks

Data Center Networks have been primary focus of research for the past decade. This trend has been triggered as more and more data are pushed to the cloud leading to an explosion in the size of the data center networks. What is also notable is the remarkable change in perception towards datacenter sizes.

In a parallel trend, virtualization of applications, servers, storage, and networks is becoming a common case. Consolidation of data center resources has offered opportunity for architectural transformation based on the use of 10GbE switches/routers. This has called for a complex network interconnect [MPF$^+$, GHJ$^+$] which provisions full bandwidth between two arbitrary nodes. Such large scale distributed systems and complex fabric are difficult to extend, thus hindering the deployability of innovative ideas.

1.2 Deployment in Data Center Networks

Deployment of new ideas in production networks has always been challenging. Standard deployment in data center networks requires either end host modification or modifications to network infrastructure. Changes to endhosts are not perceivable because of the sheer number. A server farm of 200,000 servers can only be termed a "moderately-size" datacenter today. Endhost modification calls for incremental deployment which has a longer execution time. Any hardware modification in the endhosts adds to the overall deployment of the idea.

The rate of innovation in network infrastructure has also been slow. Although there are a lot of possible approaches, each have constraints associated with them. The programming complexity of NetFPGAs and Network Processors impedes deployment. Also, NetFPGAs do not scale as well. Vendor switches provide most of the advantage on performance, scalability front. They can also handle production traffic and the complexity is abstracted away from the user. But these vendor switches are closed systems and the features exported can not be extended easily. Openflow [MAB$^+$] has shown some promises on handling most of the problem, but the OpenFlow functionalities are limited to the restricted supported features and header specific operations. Although these features are extendibles, the application has to adhere to OpenFlow specifications. Deep-packet inspection is not supported.

Software switches are traditionally known to be challenged on the scalability and the performance front. But advances in the field of multi-core technology has helped us narrow this gap. Now, the question we are trying to answer is: *What if we had a system which has high performance, scalable and provides interfaces to work at the granularity of packets without having to deal with the programming complexities of NetFPFAs and Network processors?*

NetBump is a step in this direction. These "bumps" can be placed in strategic places in the network, thus making the wire intelligent. This leaves the endhosts, unmodified; and network infrastructure, simple.

1.3 NetBump

NetBump is a simple programming model where new NIC and switch functionality can be deployed and evaluated as "bumps on the wire" in existing hardware/software network environments. This goal leads to at least the following requirements: rapid prototyping and evaluation, support for line rate processing, easy of deployment, low latency, and packet modification for a range of AQM policies.

The fundamental philosophy of the NetBump is to quickly evaluate new packet buffer and queuing mechanisms in deployed networks with minimal intrusiveness. Instead of adding programmability to the switches themselves, the NetBump augments existing switching infrastructure, allowing line rate processing with user-level C code.

NetBump exports a virtual queue primitive to implement a range of AQM mechanisms [GK99a, GK99b]. The contribution of the NetBump is to decouple virtual Active Queue Management (vAQM) from the switches themselves, enabling a variety of AQM functions to be deployed and evaluated by placing NetBumps at key points in the network.

Simple programming environment is a key advantage for NetBump. Network operators and application developers can develop and test their code in user-level C code, rather than Verilog, VHDL, or kernel code. This reduces the deployment time compared to fabricating custom ASICs for such functionalities. Programmable network processors have been a hot topic for number of years [Sha01]. However, their utility has been hampered by a difficult programming model. The complexities and the lead time of these approaches prevent experimenting with novel ideas.

In short, NetBump relies on a user-level, kernel bypass networking interface to move packets to and from the NIC and user-level threads with low latency (20 - 30 μs).

1.4 Packet Pacing

NetBump provides us with a programming paradigm to implement and evaluate a variety of network protocols involving packet modification at line rate. All our implementation till date has been focused on the use of virtual Active Queue Management (vAQM) to estimate the queue occupancy of the downstream port. RED, DCTCP, IEEE 802.1Qau-QCN are examples of implementation in this direction.

In this thesis, we shift our focus to a non-AQM based application. TCP packet pacing is one such application which is traditionally implemented with end host modifications. We claim that TCP pacing can be trivially implemented on the NetBump without the use the specialized hardware. We have implemented a modified token bucket approach which relies on packet reception as a primary event and non-blocking receive as secondary events to precisely track the token requirements to smoothen out the outgoing traffic and maintain appropriate Inter Packet Time (IPT).

1.5 Organization of the Thesis

This thesis is organized as follows: In Chapter 2, we discuss research related work to packet pacers. In Chapter 3, we briefly discuss the design requirements and the architectural details of the NetBump. Packet Pacer has been developed using the features exported by NetBump. Chapter 4 talks about the research motivation of packet pacing. Burstiness in TCP can be caused by multiple reasons, but, burstiness in general creates buffer overflow leading to packet drops. In Chapter 5, we discuss about different approaches taken for packet pacing. We describe the experimental setup and methodology in chapter 6 and evaluate the performance of packet pacers in chapter 7. We discuss the findings in Chapter 8 before concluding the thesis in Chapter 9.

Chapter 2

Related Work

2.1 NetBump

Software-based packet switches and routers have a long history as a platform for rapidly developing new functionality. The Click Modular Router [KMC$^+$00] is a pipeline-oriented, modular software router consisting of a large number of building blocks, each performing a single packet-handling task. Click's library of modules can be extended by writing code in C++ designed to work in the Linux kernel (or at userspace, albeit at generally reduced performance).

More recently, the RouteBricks [DEA$^+$] 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. A key distinction from NetBump is that RouteBricks and Click are both multi-port software switches focused on packet routing, while NetBump focuses on implementing virtual queuing within a pre-existing switching layer.

Typically the O/S kernel translates streams of raw packets to and from a higher level programmatic interface such as a socket. While a useful primitive, the involvement of the kernel can become a bottleneck. An alternative set of user-level networking abstractions and techniques have been developed [vEBBV, WBvE, ESW, BGC]. Here, user-space programs are responsible for packet handling, retransmission, and TCP sequence reassembly. To improve efficiency, user-level networking is typically coupled with zero-copy buffering, in which the memory that a packet is initially stored in is shared with each target application, avoiding extra per-packet copies. A second source of efficiency are kernel-bypass network drivers. These enable the application to directly access packets directly from the NIC memory, avoiding the need for kernel involvement on the datapath. Commercially-available NICs support these mechanisms [myr, che], and we use [myr] to implement NetBump.

2.2 Packet Pacing

Packet pacing as a technique to improve TCP performance by smoothing the traffic pattern has been around for a while. The main purpose of packet pacing is to reduce the number of buffer overflows in the switches. The buffer requirements in datacenters have been extensively studied in [AlCDR, AKM04]. In this section, we describe previous works on packet pacing with shallow buffers.

2.2.1 Packet Pacing in Endhost

Packet pacing in the endhost has been extensively analyzed. [Agg00] gives a very good understanding of how TCP congestion control mechanics can contribute to bursty behavior. TCP slow start, ACK compression and "holes" in TCP sequence number space because of packet losses are primary reasons behind burstiness. These effects are explained in section 4.1. Here, the authors evenly spread the transmission of a window of packets across the entire duration of the round trip time. Instead of transmitting the packets immediately, the sender delays the packets to spread them out at the rate refined by the congestion control algorithm - window size divided by the estimated round trip time. Similar technique is also applied for the ACK packets at the receiver.

Razdan et al. [RNN+02] also suggested packet pacing as a potential solution to increase TCP behavior with small buffers. It was noted that packet interdeparture time used instantaneous TCP sending rate. Instead, the authors used Bandwidth Share Estimate (BSE), maintained by TCP Westwood to set the pacing interval.

Precise Software Pacer (PSPacer) suggested by [PSP] is a software module which achieves precise network bandwidth control and smoothing of bursty traffic without any hardware support. Here, the key idea is to determine the transmission timings of packet by the number of bytes transmitted. If packets are transmitted back to back, the timing at which a packet is sent out can be determined by the number of bytes sent before the packet. The authors refer this as *byte clock*. As per the implementation, the gap between the "real packets" is filled with "gap packets". So, the timing for packet transmission can be precisely controlled by adjusting the size of the gap packets. PSPacer was capable of controlling the interpacket gap at 8 nanosecond resolution for Gigabit Ethernet. IEEE 802.3x PAUSE frames were used as gap packets, as they are discarded at the switch input port, thus maintaining the required interval. PSPacer was ethernet dependent and did not work with pseudo network devices. To avoid these limitation, PSPacer/HT [TTKO10] was suggested which used high resolution timers instead of gap packets. PSPacer/HT was implemented as a Queueing Discipline (QDisc) module.

2.2.2 Packet Pacing in NIC

[traa] suggests a packet pacing approach which uses specialized NIC hardware. The authors incorporate a per packet transmission timer field in operating systems packet buffer data structure (`skbuf` for Linux), which represents when the packet should be sent out. This information is used by the NIC which has a per-packet timer function implemented over a network processor.

2.2.3 Packet Pacing in Switches

Most of the initiative towards packet pacing in switches has been concentrated on Optical Packet Switched (OPS) Network buffering optical packets. These high-bandwidth networks have a significantly large delay-bandwidth product. Burstiness of internet traffic causes high packet drop rate and low utilization in small buffered OPS network.

[AAM10] talks about one such implementation on OPS networks. Here the authors have implemented a combination of XCP pacing and shared buffered switch architecture to increase goodput and reduce packet drop rate. The authors also compare various switch architecture and find that advanced switch architectures like combined input output buffered switches have high scheduling complexity, which may become a bottle neck at ultra-high speed of optical networks.

Similarly, [SEMO06] is another effort in this direction. Here, the authors claim that short-time-scale burstiness is the major contributor to performance degradation. It is proposed to mitigate the problem by pacing the traffic at the optical edge prior to injection into the OPS core.

Deploybility of all the above approaches is limited due to endhost modifications, specialized hardware, or switch level changes. In this thesis, we present packet pacers implemented over NetBump which does not have any such limitations. The implementation transparently paces out the packets, without negatively affecting the flows. Packet pacing can be implementation along with other NetBump applications/protocols thus, reducing the overall deployment cost.

Chapter 3

NetBump Architecture

3.1 Design Requirements

The primary goal of NetBump is to enable rapid development and easy evaluation of new ideas and protocols related to packet processing. To reach these goals, the NetBump must meet the following requirements:

1. **Development with unmodified switches and endhosts:** NetBump enables development to take place in the datacenter or enterprise network itself. It does this by leaving the switches and endhosts unmodified. vAQM within NetBump simulates the status of downstream switch buffers. This provides several benefits, including significantly reducing the occupancy of switch buffers network wide.

2. **Ease of development:** Datapath-modifying applications deployed within NetBump are written in userspace with C. NetBump applications can utilize any third-party libraries and development tools, making it easier to design, deploy, measure, and redesign new network mechanisms. Additionally, the overhead introduced by the NetBump is only a small constant above the latency of the program written by the user.

3. **Minimize latency** Many datacenter and enterprise applications have strict latency requirements, and any datapath processing elements must likewise

have strict performance guarantees. Since the network layer is the lowest layer in the stack, and since a single application-layer request might result in several round-trips, its switching and forwarding latency must be very low. A significant source of latency in commodity servers is the O/S kernel. While some progress has been made, kernel code does not yet scale linearly with the number of cores [BWCM+] . The approach taken in NetBump is to rely on kernel-bypass functionality now available in commercial NICs [myr, che] to completely avoid the kernel altogether, resulting in an average latency of approximately $20 - 30\mu s$.

Average latency introduced per packet is function of application running atop. Applications like packet pacing inherently queue the packets which adds to the overall end-to-end latency.

4. **Forwarding at line rate** 10Gbps is becoming standard in datacenters. Deploying a NetBump in link 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: 10 Gb/s corresponds to 14.88M 64-byte minimum sized packets per second, including Ethernet overheads. Our NetBump implementation is able to implement basic vAQM at very close to line rate with minimum-sized packets at 10 Gbps on a single core. It does this by allocating user-level application threads to their own cores, and avoiding packet copies by referencing the packets directly in the NIC buffer. For more complex protocols, we rely on multicore processors coupled with multi-processor motherboards. To provide sufficient parallelism, our hardware NICs support in-built hash functions to distribute flows across CPU cores.

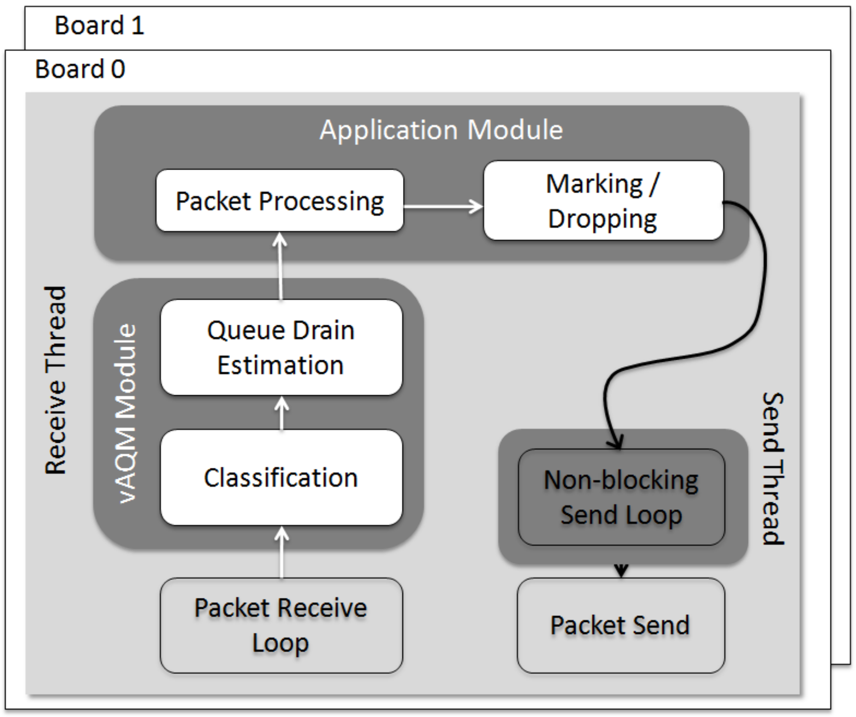

Figure 3.1: The design of the NetBump pipeline.

3.2 NetBump Design

3.2.1 Packet Reception

NetBump is based on NICs which supports kernel-bypass zero-copy APIs. The controller has NIC-local memory containing a circular ring buffer that points to driver allocated main memory used to store packets coming from the wire. On startup, the NetBump user-space application opens the NIC in kernel-bypass mode, specifying the number of rings. The NIC driver then allocates a region of main memory to store incoming packets. After receiving, the packets are placed in rings determined by hash function.

3.2.2 Virtual Active Queue Management

The Virtual Active Queue Management module estimate the downstream the queue occupancy. The module consists of three algorithms: packet classification, virtual queue drain estimation, and packet marking/dropping. When a packet is received from the NIC, it must be classified to determine which virtual queue will be enqueued into. The classification API is extensible in NetBump, and can be overridden by the user.

The purpose of the queue drain estimation algorithm is to simulate, at the time packet is received into the NetBump, what the queue occupancy is in the virtual queue associated with the packet. The virtual queue estimator is a leaky bucket that fills up as packets are assigned to it, and drained according to a fixed drain rate, based on port speed. The size of the virtual queue inside the bump is coupled with the actual size of the physical buffer in the downstream switch. Once the the packet reception crosses the vAQM threshold, the packet can be marked. Here packet marking takes place in the form of setting the ECN bit in the header.

3.2.3 Application

NetBump exports an ideal platform to implement a large number of network protocols which are based on queue length estimation. Along with that, NetBump can also be used to implement various non-vAQM types applications. Packet pacer is one such application. Internal buffers have the capability to queue the packets before sending it out. Additionally, deep packet inspection is an unique advantage exported by NetBump which makes it application-versatile, even extending to the security domain.

3.2.4 Packet Transmission

Packet transmission is done by issuing non-blocking, zero-copy, kernel-bypass send call on the packet. This copy will copy the packet content into the NIC-local memory. In rare cases it is possible that the send call might fail, so in that case the send thread loops on sending the packet until transmission is successful.

3.2.5 Multi-core parallelism

NetBump supports a multi-threaded implementation which relies in part on hardware support for multiple receive rings in the NIC, one for each NetBump thread. Each thread independently binds to its own receive ring, and the NIC includes a hardware based hash function that hashes packets to receive rings based on a configurable set of fields. Here each thread implements the entire pipeline and runs in its own core. The degree of parallelism supported depends on the number of compute cores; higher the parallelism, the more times that can be spent processing each incoming packet.

I would like to thank the entire NetBump team, especially, George Porter, Rishi Kapoor and Mohammad Al-Fares. Their sincere effort and contribution has made this architecture possible and the project successful.

Chapter 4

Research Motivation for TCP Packet Pacing

Originally, TCP was conceived and designed to run over a variety of communication links. The amount of data sent out per round trip time (RTT) is roughly equal to the delay-bandwidth product of the link. Typically, $75\mu s$ is a conservative estimate of the network budget for responding to a request in a data center. This implies that the switches should be capable of handling approximately 100 Kilobytes of in-flight data per port. On a switch level (typically 48 ports), we run into Megabytes of shared memory. The mismatch between the high capacity of these networks, and available buffer at the queues of individual network routers poses problem for TCP. Bursty behavior can easily fill-up the switch buffers, even through the average bandwidth of the flow is not significantly high. This scenario triggers a instantaneous buffer overflow in the switches, leading to packet drops. TCP reacts adversely to packet drops and gets into the state of fast recovery. In a worst case when more than 3 consecutive packets are dropped, TCP enters slow start leading to poor flow performance.

4.1 Burstiness in TCP

In this section, we describe some of the common causes for bursty nature of TCP.

4.1.1 TCP mechanism

There are various prior works describing the dynamics of TCP congestion control algorithm to be the major cause of bursty behavior in TCP. [SZC90,Agg00] extensively describe few of the common reasons.

TCP is a sliding window protocol, where the effective window used by a TCP sender is the minimum of the congestion window and the receiver's buffers size. This window is the number of bytes the sender is allowed to send without an acknowledgment. The sender uses the incoming acknowledgments to determine when to send new data, a mechanism referred to as ack-clocking [JK88].

The congestion window adjustment algorithm has two phases. In the *slow start phase* [JK88], the sender increases the congestion window rapidly in order to quickly identify the bottleneck rate. The sender typically starts with a window of 1 packet and doubling the window every round trip time (RTT), until a packet drop is detected. At this point the window is cut in half and sender enter the *congestion avoidance phase*. In this phase, the sender increases the congestion window by 1 packet per RTT.

Slow Start

During the slow start phase, the sender transmits two packets for every new acknowledgment. Since the acknowledgments are generated at the bottleneck rate, this implies that the sender is bursting data twice the bottleneck rate, leading to the formation of queue at the bottleneck link.

Since the buffer size at the bottle-necked link is limited, this bursty behavior will cause the switch to drop packets when the window size exceeds the switch buffer size by some small constant factor. In the worst case, if the switch buffer size is much less than the delay-bandwidth product, the sender will encounter drop too early in the slow start phase and the sender might take many round-trip to fill the bottle-necked bandwidth.

Losses

Successful retransmission of a lost packet may also trigger a traffic burst. At the receiver, the retransmitted packet will typically fill a “hole” in the sequence number space, enabling the receiver to acknowledge not just the lost packet but other packet that has been successfully received in the window. This triggers the sender to send a burst of packets.

ACK Compression

In the presence of two-way TCP traffic, ack-clocking can be disrupted due to ack-compression [SZC90]. Acknowledgments form the receiver can be queued behind data packets on the reverse path. Assuming that the switch services packets in FIFO order, this can cause acknowledgments to lose their spacing and reach the sender in a burst. This in turn causes bursty transmission at the sender.

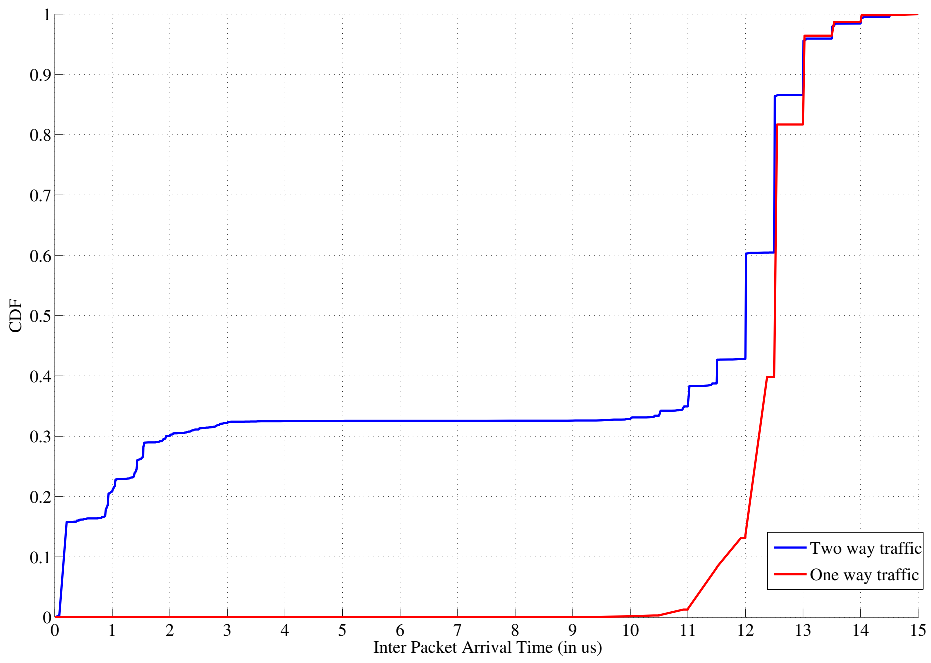

Figure 4.1: Effect of bi-directional flow on inter packet time

Figure 4.1 shows the inter packet time in case of single directional and bi-directional flows over a 1Gbps link. As expected, in case of a one-way traffic, the inter packet time is maintained at 12.5μs. But, in case of two-way traffic, TCP looses ack-clocking leading to burstiness. In this particular run, approximately 32% of the packets were bursty.

4.1.2 Large Segment Offload

Large Segment Offload (LSO) or TCP Offload Engine (TOE) is a technology used in the NIC to offload the processing of entire TCP/IP stack to the network controller. It is primarily used with high-speed network interface, such as Gigabit Ethernet and 10 Gigabit Ethernet, where the processing overhead of the network stack becomes significant [LSO, Wik].

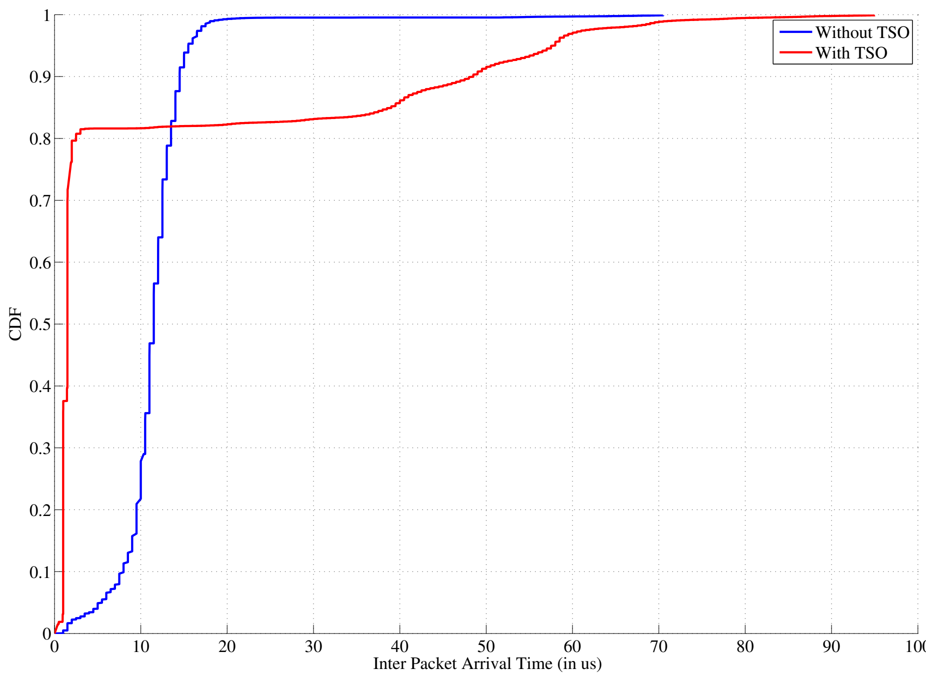

Figure 4.2: Effect of TCP Segment Offload on Inter Packet Arrival Time

In case of LSO, the TCP layer builds TCP message up to 64KB long and sends it down the networking stack through IP and ethernet device driver. The adapter then re-segments the message into multiple TCP frames to transmit on the wire. The TCP packets sent on the wire are either 1500 byte frames for a MTU of 1500 or up to 9000 byte frames for a MTU of 9000 (jumbo frames).

Figure 4.2 shows the effect of LSO on inter packet time for 1Gbps flow on a 10Gbps link. The NIC tries to send the TCP segments as fast as possible. This creates a burst on the wire. 82% of the packets were sent out in bursts. These bursts were separated by an idle phase of approximately $30\mu s - 70\mu s$.

4.1.3 Ethernet Flow Control

PAUSE is a flow control mechanism on full duplex Ethernet link segments defined by IEEE 802.3x and uses MAC Control frames to carry the PAUSE commands. Flow control in Ethernet is a data link layer protocol to cater a situation where a sender may be transmitting data faster than some other part of the network (including the receiving station) can accept it. The overwhelmed network element sends a PAUSE frame, which halts the transmission of the sender for a specified period of time.

Flow control using PAUSE frames is being widely deployed for data center bridging [flo]. In these scenarios, a PAUSE frame can cause buffer buildup in the upstream network element. This buffer is drained immediately after the pause duration, thus leading to a bursty behavior.

4.2 Effects of Packet Pacing

The primary effect to TCP packet pacing is to reduce the number of buffer overflows in downstream switches. A well-paced flow reduces the number of spikes in buffer utilization, thus leading to fewer packet drops. The drop-tail queue implementation in switches cause consecutive packet drops in case of buffer overflow. Consequently, packet pacing also reduces the number of slow starts and improve the performance of TCP flow.

Receive Livelock [MWMR97] is a state of the system where no useful progress is made, because some necessary resource is entirely consumed with processing receiver interrupts. Bursty nature of TCP traffic is one of the major cause of momentary receive livelock. The latency to deliver the first packet in a burst is increased almost by the time it takes to receive the entire burst. All these phenomenon adds jitter to the packets. TCP packet pacing reduces this jitter by reducing the number of receive livelocks and eliminating bursts.

In a data center network with strict SLAs, there are requirements to have reliable latency measurements for all the packets. But, most of the time we observe that there are some outlines. From Figure 4.2 its observed that approximately 17% of the packets have inter packet gap of more than $30\mu s$. These types of behaviors affect the end-to-end SLA of the request. Ideally, packet pacing reduced the number of outlines, thus leading to reliable response time.

Chapter 5

Packet Pacing: An application on NetBump

5.1 Packet Pacing

Packet Pacing has been developed as an application on the top of NetBump. Unlike previous approaches described Chapter 4, where pacing is either done at the endhosts or at the switches, we opt to implement packet packing at there “bumps on the wire” strategically placed throughout the network. The implementation is completely transparent as neither the endhosts nor the switches are aware of the presence of the bump other than in a positive way.

We believe that the packet packing should meet the following design requirements:

R1 Buffer Overflow: The primary goal for packet pacing is to smoothen out the flow and reduce buffer flow.

R2 Multiple Flows: The NetBump should be able to handle and pace multiple flows at the same time. The pacing on a given flow should not affect the throughput. Nor should it affect the throughput or add undue latency to parallel flows.

R3 Flow Integrity: Packets from a flow should not be reordered. Packets from different flows should be scheduled appropriately to maintain the inter packet gap.

R4 Transparency: Packet pacing should not pose any overhead for the endhosts or switches. Packet pacing should be done transparently without affecting the flow.

5.2 Packet Pacing Design

To cater to requirement R2 and R3, we are drawn to an implementation where flows are not blocked because of pacing. The high performance single threaded implementation of NetBump does not suit in this situation.

5.2.1 Multi-Threaded Implementation

Figure 3.1 shows an overview of the multi-threaded implementation of packet pacer. Here the receive side is decoupled from the send side to avoid packet blocking.

Taking advantage of NetBump's kernel-bypass, zero-copy programming model, the packet can be received off the wire with minimal overhead. Instead of being notified of packet reception, the NetBump polls for packets using `receive()` API call. The non-blocking `receive()` functionality from sniffer10g is used so that, the packet is immediately delivered to the user-space application without adding any additional delay. The `receive()` call will return the pointer to the packet and packet length, and will also set it timestamp. The receive thread copies the packet content stored in the rings into the preallocated receive queues.

Similarly, the send thread is responsible for sending out the packets. This is done by issuing a non-blocking, zero-copy, kernel-bypass send call on the packet. This call will copy the packet content into the NIC-local memory if space is available, and arrange the packets to be transmitted on the wire. NetBump is responsible for reliably transmitting the packet on the wire.

During the course of the experiments, we observed limitation with the injector10g API from myricom where the API buffers packets before sending out on the wire. This behavior is limited to low bandwidth flows and is not observed with 10G flows. The API buffers the packets till it receives multiple packets to be sent out. This works against the software pacing and introduces bursts in the outgoing traffic. As an alternative implementation, we also used kernel based `sendto()` implementation to transmit packets. We have discussed the results for such an approach in the following sections.

To take advantage of the multi-core capabilities, the threads are pinned to different cores thus reducing CPU contentions. The receive and send threads are co-located on the same CPU die. This approach takes advantage of shared L3 cache and reduces the movement of packets among cores over the QPI bus. The packets can also be tied to their respective memory banks in the NUMA architecture.

5.3 Pacing Mechanism

The key idea behind packet pacing is to precisely time when the packets are sent out. Different previous implementations (Pause frames, NIC based hardware timer paper) rely on endhost kernel modifications, or custom NICs to implement packet pacing. Our implementation obviates the need for any custom hardware/software changes in the endhost and does packet pacing when packets are in-flight. This satisfies requirement R3 stated above.

5.3.1 Which flow to pace?

Packet pacing is an expensive operation and care is taken to choose which flows need to be paced. Buffer overflow is a primary side-effect of bursty traffic and we use that as a metric to choose the flows to be paced. This idea is inspired by the HULL implementation [AKE+]. We mark the flow to be paced only when it leads to a buffer overflow. We assume an ECN aware downstream switch is capable of marking the ECN bit in the packet once the queue size exceeds the queue threshold. For this reason, we monitor the ACK packets. Once we receive an ACK with ECN bit marked, we move the flow to the paced set. In case of multiple competing flows, a timeout can be placed on the flow, to prevent the starvation of new flows to be paced in the presence of old flows.

This design decision is consistent with the real-life datacenter traffic pattern which consists of few elephant flows and multiple small flows. The elephant flows have high degree of burstiness because of higher delay-bandwidth product. We target these flows for better performance.

5.3.2 Flow classification

As the packets are received from the NIC, it must be classified into flows. These classified flows are input to the packet pacer module. In this section, we explore two different techniques for packet classification, namely, RSS hashing and NetBump classifier. One or more of these classifications can be marked to be 'un-paced' where the corresponding flows will not be paced. These classifications are reserved for control packets which should not be delayed by the NetBump. These un-paced traffic is handed over to the send thread immediately, without any delay or buffering.

RSS Hash

Receive Side Scaling (RSS) is a feature available in modern NICs where incoming traffic can be routed to specific queues efficiently balancing network load across CPU cores and increasing performance in a multi-processor systems. By default, the myricom library uses deterministic hashing to make sure that packets that are contained in a particular flow are always delivered to the same ring/thread. The myricom library can operate in either SNF_RSS_FLAGS or SNF_RSS_FUNCTION. The SNF_RSS_FLAGS mode allows the user to functionally specify which parts of a packet are significant in the RSS hashing process. The packets can be classified on the basis of IP address, source port or destination port. In the SNF_RSS_FUNCTION mode, the API provides the flexibility to segregate the traffic on the basis of a custom hash function. The limitation of this mode is that the hashing is serialized in the software.

Table 5.1: The NetBump packet data structure.

Field name	Description
data	Pointer to packet contents
len	Length of packet
mark	Used internally for packet marking
timestamp	NIC-supplied timestamp, in ns
delay	Delay to be added for that packet, in ns

NetBump Classifier

NetBump exports API for fine grain custom packet classification. The classification API is overwritten to classify the traffic on the basis of destination port. The control traffic is placed on 'un-paced' classifier.

5.3.3 Delay Based approach

NetBump monitors the flows and estimates throughput and corresponding inter-packet time. The packets are timestamped by the NIC on arrival. The values are stored in per-packet data structure as shown in Figure 5.1. The per-packet timestamp by the NIC is used to accurately determine the flow bandwidth without the use of expensive timing calls (like `clock_gettime()`). `clock_gettime()` has an overhead of 600ns per call.

The bandwidth of the flow is measured at a granularity of 1000 packets. Once the packet bandwidth is determined, the inter-packet time is calculated. The packet timestamp is used to evaluate the current inter packet delay. Additional delay to maintain appropriate spacing is calculated and stored in the 'delay' field of the packet. The packet is then transferred to the send thread.

We take advantage of multi-core architecture and have a send thread for every receive thread. There is a one-to-one mapping between the threads and flows. For higher performance, we pin the receive and the send thread to the same CPU die. Typically, the send threads are allocated to the hyperthreads (core 8-15). The send threads wait for predetermined duration before sending out the packet. System call to `sleep()` is too expensive for introducing delay at the granularity of

```
Procedure delay_iters(int64_t n): {
static volatile double x, y;
static volatile double z = 1.000001;
int64_t i;
y = z;
x = 1.0;
for (i = 0; i < n; i++)
   x *= y;
return x;
}
```

Figure 5.1: Sample Code for delay_iter

nanoseconds. Instead, we use tight loop to introduce delay between the packets.

Tight Loop Delaying

We employ tight loop for a fixed number of iterations to introduce delay. The volatile keyword is intended to prevent the (pseudo)compiler from applying any optimizations on the code that assume values of variables cannot change "on their own." Figure 5.2 shows the variation of observed delay. The variation is more or less linear with spikes observed in between.

The use of volatile variable restricts its use in multi-threaded environment as concurrent call to the procedure does not guarantee reliable delay.

NOP based Delay

To avoid the limitation of delay using volatile variable, we have taken a naive approach to execute NOP assembly instruction in tight loop. This approach generates delay in a reliable manner. Figure 5.2 compares the delay observed with both approaches. We attribute the spikes in the Volatile based method to context switching by the operating system.

Only one flow can be paced per thread. This is major limitation of the

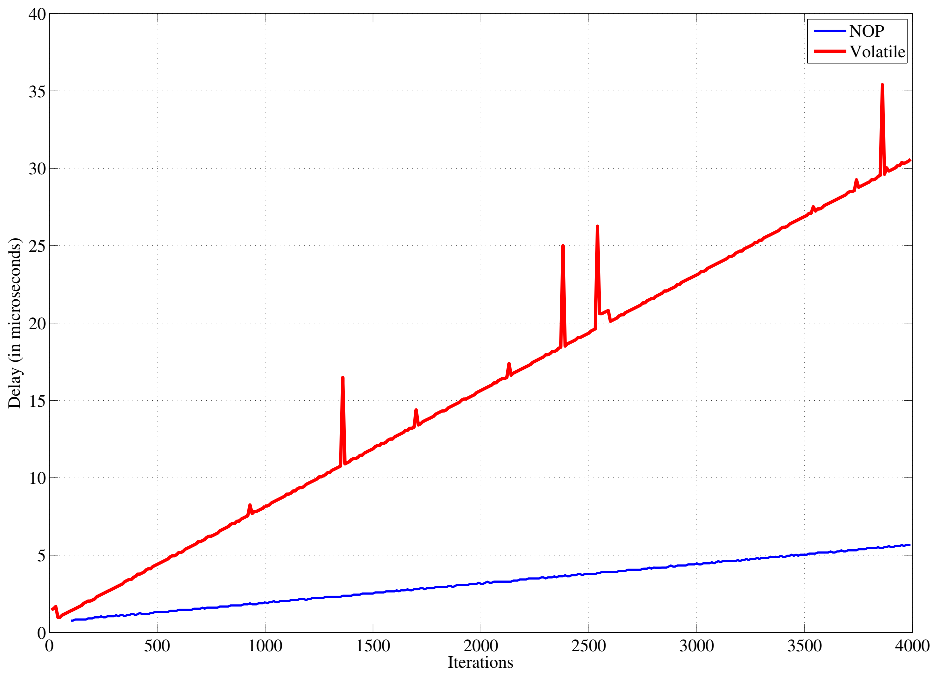

Figure 5.2: Variation of delay generated using the NOP and volatile approach

approach as the number of flows that can be paced is limited by the number of available cores.

5.3.4 Modified Token Bucket approach

Token bucket is a common algorithm used to control the amount of traffic that can be injected into a network. Packet reception is treated as events for the token bucket implementation. Typically, the amount of tokens added to the token bucket is proportional to the time difference between the events and the bandwidth of the flow.

TCP behavior has been bursty (Section 4) and can be best described in Figure 5.4 Typically, the packets arrive in short bursts. For a 1Gbps flow (over 10Gbps link), the inter-packet arrival time between the packets is typically $1.5\mu s$. The bursts are separated by approximately $40\mu s$. So, as per the traditional token bucket implementation, although we add tokens in a restricted manner in the

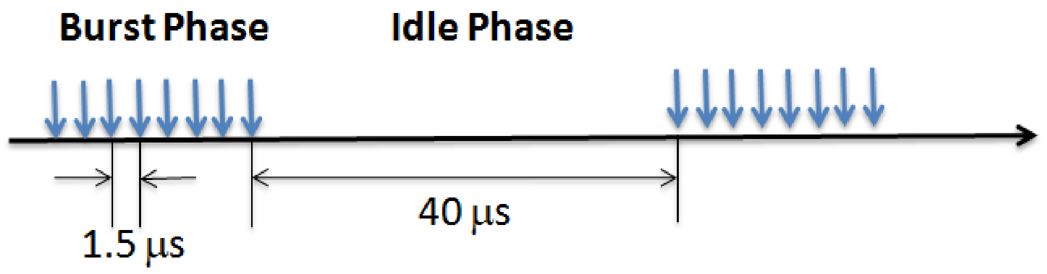

Figure 5.3: Schematic of the bursty behavior of TCP

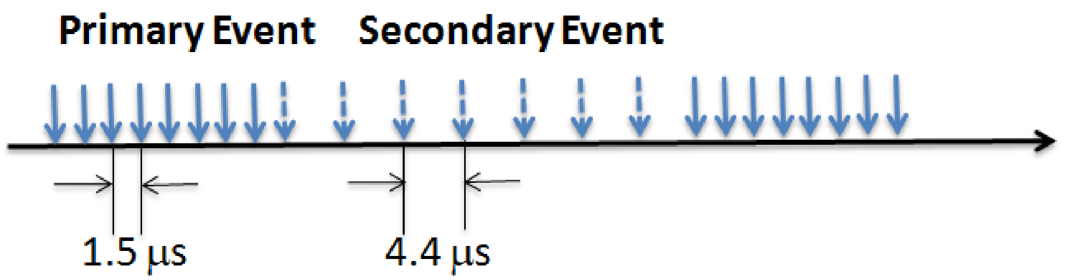

Figure 5.4: Schematic of the events for modified token bucket approach

bursty phase, we end up adding a burst of tokens after the end of the idle phase. This excess token is responsible to bursty output traffic.

To avoid this problem, we have developed a modified token bucket implementation with two event mechanisms. The primary event mechanism is based on the packet reception. As expected, the amount of token added for the primary event is proportional to the time gap between the events.

The secondary event mechanism is a virtual 'tick' which is generated in the idle phase. As mentioned earlier, we employ a non-blocking `receive()` poll based implementation to receive the packets. `receive()` returns EAGAIN when there is no packet in the NIC rings to be received. We observed that in the idle phase, `receive()` returns EAGAIN reliably. 10 EAGAIN returns from `receive()` call approximates to 4819ns of elapsed time. This can be used as the secondary event where tokens are added. The estimated timestamp is incremented accordingly. The estimated timestamp is reset to the timestamp for the packet received. In our experience, the estimated timestamp closely followed the actual timestamp.

Packets are transferred to the send thread at both primary and secondary event. We employ C++ STL `splice()` operation to move packets from the receive threads to the send thread. The list containing the 'un-paced' packets (control packets) is spliced entirely. For the rest of the queues, packets are spliced in a controlled manner depending on the amount of available tokens. Before the `splice` operation, the number of packets, *numPkt*, that can be trasferred is computed. *numPkt* packets are then spliced from the head of the queue.

This technique can be implemented over a multi-receive single-send setup. Here every different receive thread has its own queue and mutex. The send thread polls the receive threads for packets. Distinct mutex per thread reduces lock contentions. The send thread takes the lock and splices the packets to its buffer which can then be sent out asynchronously.

5.4 Token Thread Approach

In most of the other NetBump applications like rate limiting, QCN, we use packet reception as an event to add tokens. Even for a well paced 1Gbps flow, tokens were added at ideal IPT, i.e $12\mu s$, which is very course grained.

Instead of having the token bucket mechanism completely ingrained in the packet bucket implementation, we have an alternate mechanism where the token handling is done by a separate token threads. These threads are pinned to unused cores. In this mechanism, we can have different token bucket threads adding tokens at different rate. The receive thread can derive its tokens from any one of the available token threads. These token threads generate tokens are different rates and the inter packet time can be determined accordingly. This model is very similar to a actual hardware implementation where there is dedicated module for token generation.

We use this model to increase the efficiency of the rate limiter module in

```
Procedure modified_token_bucket(): {
int i, numPkt;
std::list<struct packet *>::iterator end;
std::list<struct packet *>::iterator localListEnd;
idleCount++;
if(idleCount == 10 ) {
  idleCount = 0;
  for(i =0 ; i < 10; i++) {
    if(token[i] < 4500)                 //Limit Queue Length to 3Pkts
    token[i] += 550; }                  //Tokens for 1Gbps
  if(idleLastTS != 0)
  idleLastTS += 4500;
  for( i = 1; i < 10; i++) {
    numPkt = token[i] / 1500;           // Determine the number of packets
    int llSize = llCount[i];
    if(numPkt > 0) {
      localListEnd = localList.end();
      end = localListArray[i].begin();
      if(numPkt < llSize) {             // In case we have more packets
        advance(end, numPkt);
        token[i] -= numPkt*1500;
        llCount[i] -= numPkt;
      } else {                          // splice the entire list
        end = localListArray[i].end();
        token[i] -= llSize*1500;
        llCount[i] = 0; }
      if(token[i] < 0) token[i] = 0;    // Limit queue length
      localList.splice(localList.end(), localListArray[i], localListArray[i].begin(),
    end);}                              // Splice the required number of packets.
  }
}
```

Figure 5.5: Pseudo-code for modified token bucket implementation

NetBump. Instead of adding tokens every $12\mu s$, using this approach we reduce the granularity, which in turn provides a much tighter control over the output rate. In our implementation, the token threads are also capable of keeping track of the capacity of the virtual port, thus enabling them to react to capacity changes. We maintain low queue occupancy in the NetBump which enables us to reduce the feedback loop and reaction time.

The tokens are updated by the token thread, which runs on a different core than the rate limiter. At such fine granularity, maintaining consistency of data structures between the cores becomes a big problem. The updates between cores are not visible without appropriate synchronization because of compiler optimizations. But, locking overhead (Table 7.2) is also significantly high which causes the system to halt. To avoid this situation, we declare the data structures as `volatile`. This avoids any kind of compiler optimizations, thus enabling the updates to be eventually visible between cores. This is a very relaxed consistency model where the system operates on the available value of share data structures, assuming that its not too stale. [pin] states that the cache-to-cache ping pong latency for 2 cores on different dies and different socket is 225 cycles. This is worst case situation, and in our case, both the cores are on the same socket. This means that the updates are visible and the values in the shared data structure are still valid. We use this approach to improve the performance of the rate limiter module in NetBump.

Chapter 6

Experiments Methodology

6.1 Implementation

Equipment: Our NetBump implementation of a single HP DL380G6 server with two Intel E5520 four-core CPUs, each operating at 2.26GHz with 8MB of cache. This server has 24 GB of DRAM separated into two 12 GB banks, operating at a speed of 1066 MHz. Plugged into the PCI-Express Gen 2 bus is a single 8-lane Myricom 10G-PCIE2-8B2-2S+E dual-port 10Gbps NIC which has two SFP+ interfaces, as well as a 10G-PCIE2-8B2-2C+E dual-port NIC with two CX-4 interfaces. Both cards load Myricom's Sniffer10G driver version 1.1.0b3. The host runs Debian Linux 2.6.28. We use copper direct-attach SFP+ connectors to interconnect the 10 Gbps endhosts and our NetBump. Experiments involving 1 Gbps endhosts rely on a pair of SMC 8748L2 switches that each has 1.5 MB of shared buffering across all ports. Each SMC switch has a 10 Gbps CX-4 uplink that we connect to the CX-4 ports of one of our NICs. The 10Gbps uplinks were introduced through CX-4 modules in the expansion slot. Auto-negotiation could not be turned ON on these slots as a result of which, burstiness due to PAUSE frame (Section 4.1.3) could not be simulated. NetBump is built with GNU gcc 4.4.5, using the -O3 optimization setting.

NetBump Configuration: Extensive care has been taken in the NetBump implementation to not just maintain low average latency , but also to reduce variance in latency. A major source of latency outlines comes from multi-core processors. Modern CPU architecture provide both separate memory across NUMA banks. This means that the access time to different memory banks changes based on which codes issues a given request. To reduce latency outlines, NetBump was booted with a Linux instance in which all but one core were removed from the default Linux scheduler. The cores (7 in our case) could only be used when NetBump explicitly set a given thread's affinity to that particular core.

6.2 Methodology

Figure 6.1 shows the testbed setup to evaluate the performance of the packet pacer. The testbed involves 2 NetBumps: Pacer and Monitor. The endhosts and NetBumps are connected over 10Gbps interface using SFP+ connectors. The 'Pacer' NetBump implements the various packet pacing techniques. The 'Monitor' NetBump is used for measurements purpose and keeps track of the inter packet arrival time of the paced flow.

The 'monitor' testbed can also be used as a trigger point to start packet pacing. Once the monitor detects buffer overflows using the vAQM mechanism, it marks the ECN bit of the packet. The 'pacer' NetBump detects the corresponding ACK packet from the server and moves the flow to the pacing mode. Alternatively, to reduce the control loop, the 'monitor' can also send a message to the 'pacer' informing about the buffer flow.

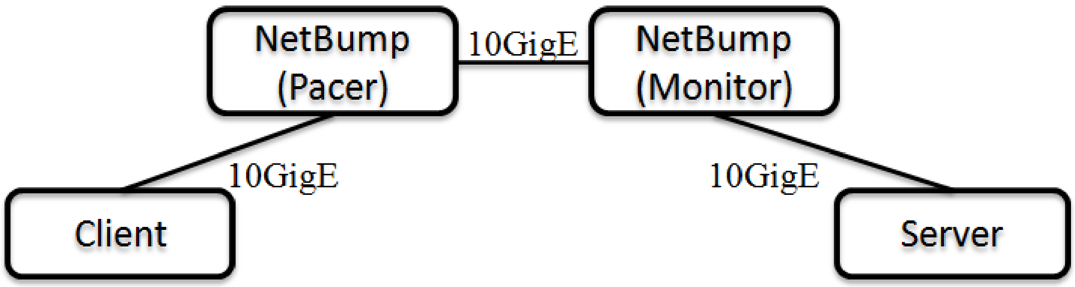

Figure 6.1: Testbed configuration to evaluate Packet Pacing

We use `iperf` as the packet generator on the client side. We restrict the outgoing traffic from the client to 1G using standard Linux traffic control technique, `tc` [Trab].

The setting for `tc`:

```
tc qdisc add dev eth5 root handle 1:  htb default 99
tc class add dev eth5 parent 1:  classid 1:99 htb rate 1000Mbit
```

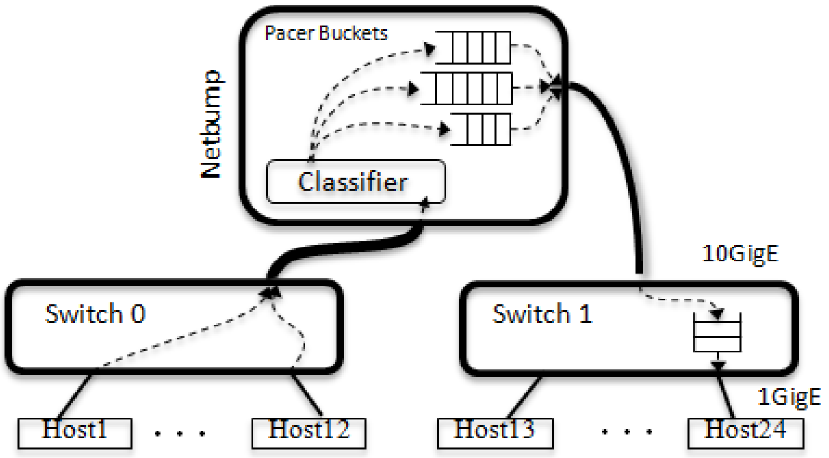

Figure 6.2: Testbed configuration to evaluate 1G flows

The second testbed, shown in Figure 6.2, evaluates the NetBump in a datacenter 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 link connects the 2 1Gbps switches and the NetBump is deployed inline with those uplinks. The NetBump is connected to the SMC switch uplink over CX-4 connectors. We use this setup to evaluate the inter packet time for 1Gbps flows with and without cross traffic. This also enables us to observe the IPT when the packet traverses from a 1G interface to 10G uplink.

We employ a slightly different testbed configuration (shown in Figure 6.3) to evaluate the effect of token threads approach on rate limiting and 802.1Qau-QCN layer-two congestion control specification. In this case, the NetBump actually have four 10Gbps interfaces - two CX-4 connectors to each of the two SMC 1 Gbps

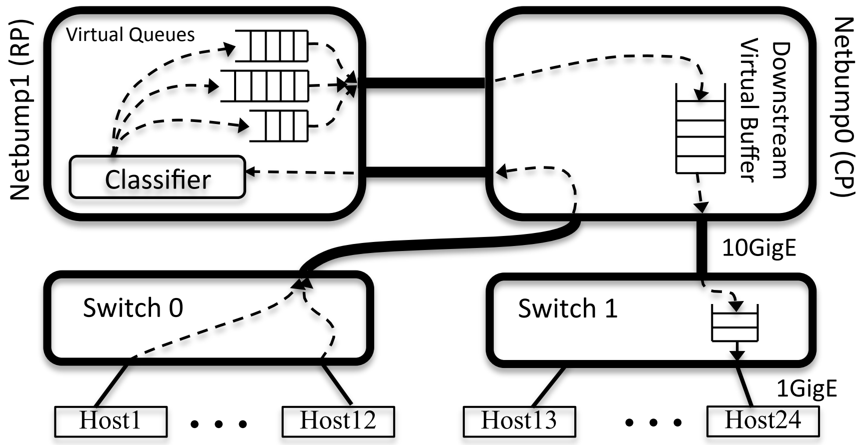

Figure 6.3: 2-Rack testbed to evaluate QCN

switches, and two SFP+ connectors that connect to a second NetBump. Here one NetBump acts as the Congestion Point (CP) and the other acts as the Reaction Point (RP).

I would like to thank George Porter for the testbed setup which enables us to run a variety of experiments. I would also like to thank Mohammand Al-Fares for original the QCN implementation.

Chapter 7

Evaluation

7.1 Inter-Packet Arrival Time

We consider inter-packet arrival time (IPT) to be our primary metric of evaluation. Bursty TCP traffic with low inter-packet arrival time fill up the switch buffer quickly, thus leading to packet drops.

Observations:

- Inter-packet time for a line rate flow is function of packet size. The time taken to transmit a packet is dependent on the underlying NIC architecture. For example, the time taken to transmit a 1514 byte packet (along with preamble and FCS) is approximately $12\mu s$ - $15\mu s$ over a 1Gbps link, where as the same is observed to be around $1.5\mu s$ over a 10Gbps link.

- TCP Segmentation Offload (TSO) has no effect on IPT for 1Gbps flow. The effect is pronounced over the 10Gpbs link. At high delay-bandwidth link, the congestion window can be significantly large, which leads to huge buffers being transferred from the kernel to TCP offload engine. This causes large number of TCP segments, which leads to burstiness.

- When packets from a 1Gbps interface is transferred to a 10Gbps up-link, the transmission time over 10Gbps interface is reduced, but the IPT is still maintained. So, it is very difficult to simulate burstiness over 1G interface.

Table 7.1: Inter Packet Time for different offered load

Required Load(in Gbps)	**Inter Packet Time (μs)**
1	10.550507
2	5.540362
3	3.662109
4	2.659462
5	2.116895
6	1.781650
7	1.389060
8	1.198500
9	1.060432
9.5	0.940450

A synthetic packet generator was used to generate traffic with different offered load using the myricom APIs. Table 7.1 shows the inter-packet delay that needs to be introduced to achieve the required load. The required delay is consistent with the calculated inter packet arrival time. As seen from the table, the inter packet delay is less than $2\mu s$ for a 6Gbps flow. This implies that the scope of pacing a flow decreases at higher bandwidth. In this thesis, we focus on pacing the common case flow of 1Gbps. This handles the general case where the servers in the racks are connected to the Top of Rack (ToR) switch. The input traffic can consist of multiple 1Gbps flow which is first classified and paced separately.

7.2 Microbenchmark

The performance of the packet pacer is heavily dependent on the accuracy of the myrisnf `send()` API. To evaluate the performance of the `send()` API, we send out packets every $5.4\mu s$ from a synthetic packet generator in the client. The client was configured to use myrisnf driver. The 'Pacer' NetBump measured the inter-packet arrival time and used raw socket based `sendto()` to send out the packets. The output of the 'Pacer' was sent out to the 'Monitor' where the interpacket arrival time was measured.

From Figure 7.1 we observe that the inter-packet arrival of the packets from the Client is not exactly $5.4\mu s$. 50% of the packets had an IPT of $1.5\mu s$ while rest

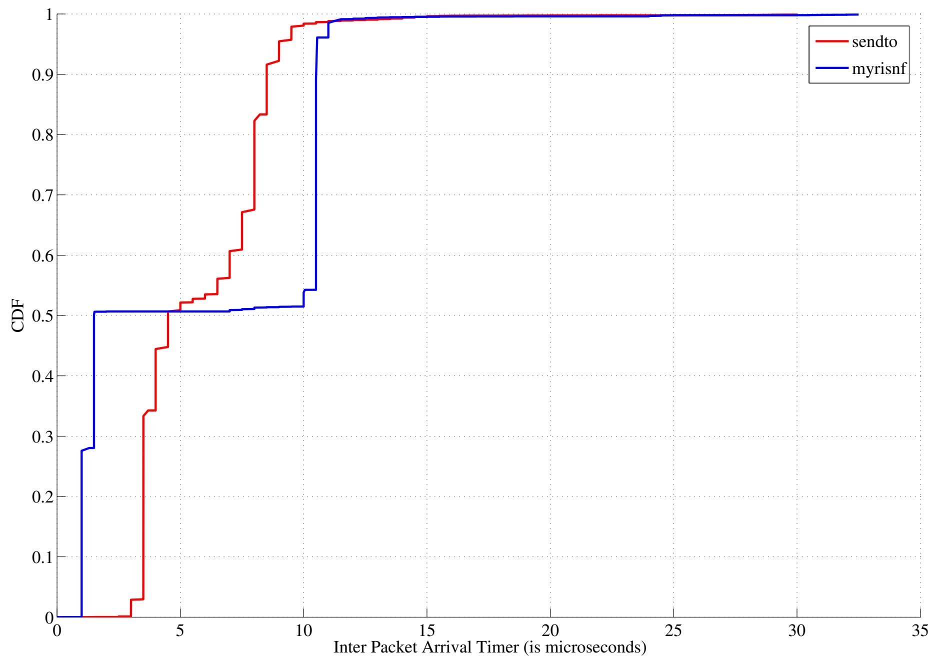

Figure 7.1: Comparison between `sendto()` and myrisnf `inject()` API

of the packets had $10.5\mu s$. This implies that the myricom `send()` functionality internally buffers the packets for a while before sending it out. This behavior is only observed at low throughput and not visible at 10Gbps. When the same traffic is sent out using raw sockets, it was observed that the `sendto()` has an operating granularity of $4\mu s$. Additionally, the `sendto()` introduces considerable kernel overhead.

On the basis of these results, we assume that the incoming flows are less than 1Gbps. This assumption also relates to the real life scenario where the input from the endhosts is limited to interface speed.

Effect of Lock

Synchronization overhead is significantly visible while operating at higher bandwidth. So, in this section, we evaluate the overhead of locking on inter-bump latency. The packets were timestamped when received. Latency was calculated on

Table 7.2: Lock overhead in multi-recv single-send setup

Receive Threads (per port)	Avg. Latency (ns)	Std. Deviation (ns)
1	3298.20	1265.55
2	3669.28	1631.20
3	3945.31	1766.57
4	4036.08	3907.43
5	4085.19	6007.71
6	4160.16	2471.18

the basis of timestamp just before the packet was handed over to the send API. We use `clock_gettime()` for getting the timestamps. The timestamps were measured every 1000 packets.

Table 7.2 shows the effect of lock contentions. Reliable latency with low standard deviation was observed till 3 receive threads were used. In this setup, we had a total of 6 receive threads (3 threads per port) each pinned to their respective cores. The send threads were pinned to hyperthreads 14 and 15 respectively. When the number of receive threads was increased, we observed high standard deviation. We attribute this variation to the use of hyperthreads causing cache contentions. To avoid variability, we restrict the setup to use only one hyperthread per core.

7.3 Delay Based Approach

In this section we evaluate the delay based approach for packet pacing. Here, we had 3 flows through the NetBump. The traffic was classified using RSS Hash on the basis of destination port. Packet pacing was done using the NOP - delay approach and volatile variables.

Figure 7.2 shows the effect of packet pacing by NOP and volatile based approach. As expected from Figure 5.2, the NOP based approach has less variation with respect to the volatile approach. This result is also reflected in the inter-packet arrival time. The NOP approach does an accurate packet pacing. The inter-packet time for 98% of the packets was between $10\mu s$ to $13\mu s$.

Variation in the delay 5.2 observed by the volatile method is the probable

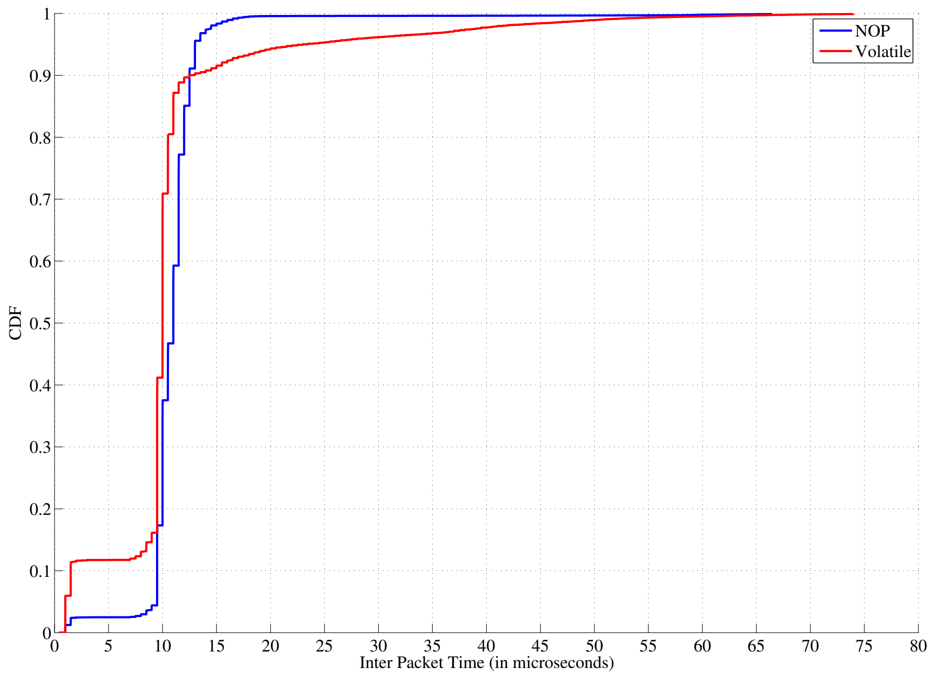

Figure 7.2: Packet Pacing using the NOP and Volatile delay approach

cause of variation in packet pacing, where 6% of the packets had IPT greater than $20\mu s$. In this approach, only 80% of the packets were paced. 11% of the packets remained unpaced. We attribute this to the `send()` API limitation where variation in delay causes the packet to the buffered by the API before being sent out.

7.4 Modified Token Bucket Approach

The delay based approach is limited to the number of cores in the system. As each thread paces its corresponding flows. This limitation is alleviated by the modified token bucket approach. Here a single core is capable of handling many flows. The implementation should have the capability to tune the inter packet time with respect to the offered bandwidth. In this setup, we vary the offered load and observe its effect on the inter packet arrival time. The pacer module is capable of adjusting to different offered load. The mean IPT varies as calculated.

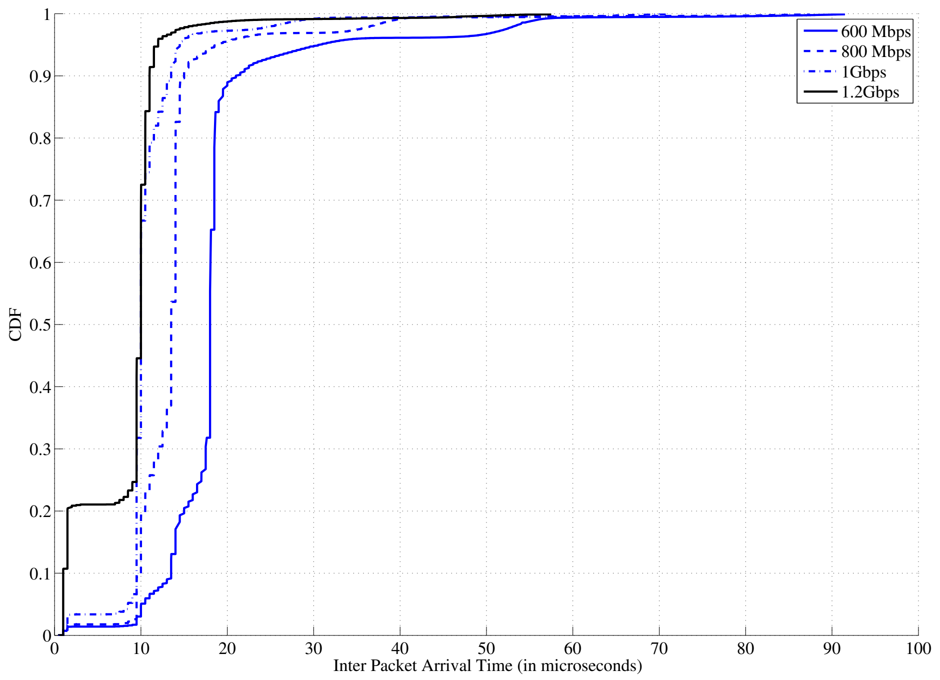

Figure 7.3: Effect of offered bandwidth on inter packet arrival time

It is also observed that the performance is better for flows less than 1Gbps. Once the system is subjected to a flow greater than 1Gbps, we observe that 20% of the packets are still bursty. This is because `send()` API limitations, where the API waits for some kind of timeout before sending the packet out.

Figure 7.4 shows the effect of modified token bucket implementation on an unpaced TCP flow. Approximately 86% of packets from the unpaced flow are bursty. The gap between the burst usually refers to the sender waiting for acknowledgment. The modified token bucket based packet pacer is able to pace out the packets with average IPT as $14\mu s$. The minimum gap between any 2 packets is approximately $10\mu s$, whereas the maximum gap is approximately $20\mu s$. The implementation is able to trace the approximate timestamp in the idle phase. The number of tokens added is proportional to the time different between events. The time difference for token addition is capped to $5\mu s$.

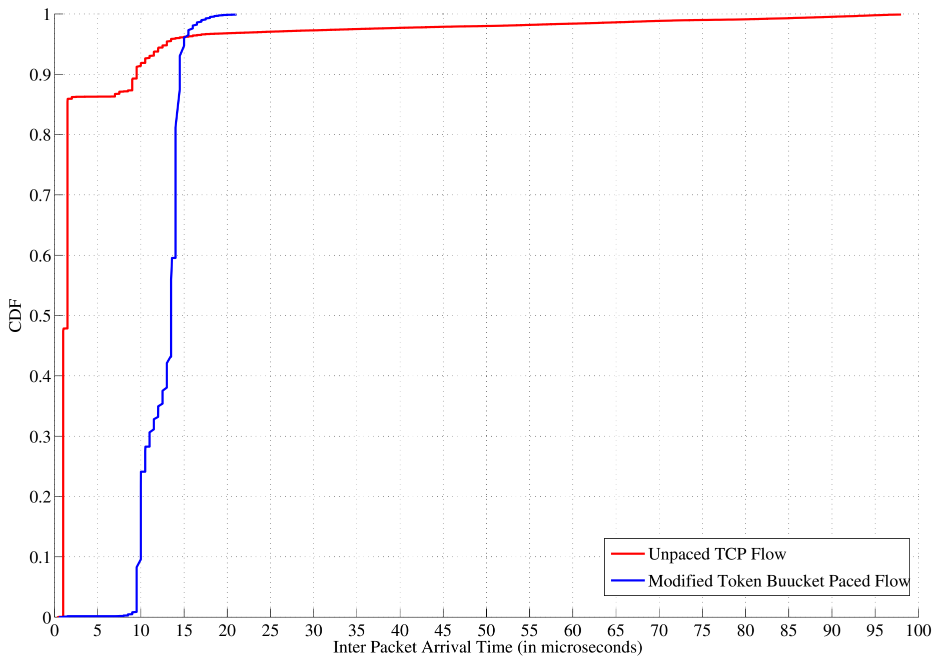

Figure 7.4: Packet Pacing using the Modified token bucket approach

7.5 Effect on Buffer Overflow

The primary goal of packet pacing is to reduce buffer overflow in the downstream switches. To validate this requirement, we ran experiments to evaluate the effect of packet pacing of 1Gbps flow on buffer overflow. Packet pacing is done by the 'Pacer'. We program the drain rate of the 'monitor' to be 1Gbps. The monitor maintains the number of times the vAQM module runs out of token. Ideally, the packet should be marked/dropped, but we leave the packet unchanged, so that it does not affect the TCP flow adversely.

In the absence of packet pacing, the effect of burstiness is distinctly observed with 20,044 instances of buffer overflow. The vAQM module is not able to keep up with the token requirement for the long bursts. The NOP based approach has better performance on packet pacing with only 1,565 buffer overflows. The modified token bucket approach performs the best with no buffer overflows.

This shows that Modified token bucket approach works best for packet

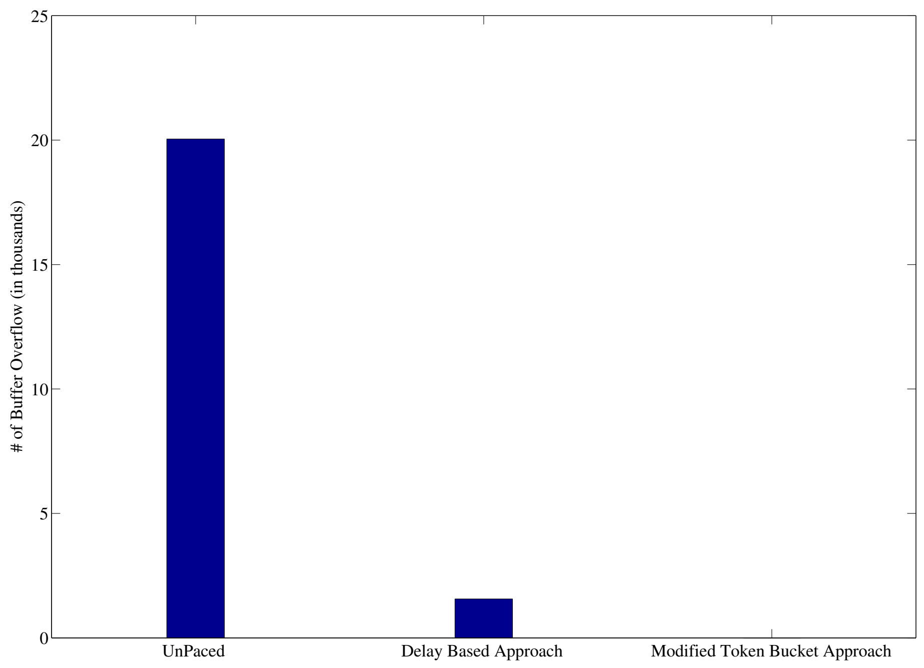

Figure 7.5: Buffer overflow for different approaches

pacing. The combination of primary and secondary events precisely tracks the token requirements of the pacer and help precisely determine the send time for the packet.

7.6 Token Thread approach

We use the Token Thread approach to see the effect of rate limiting and 802.1Qau-QCN layer-two congestion control specication [AAK$^+$]. QCN switches (Congestion Point) 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 (RP). The Reaction Points are then responsible for adjusting the sending rate accordingly. The original QCN implementation(for NetBump) was developed by Mohammad Al-Fares and was deployed on the traditional packet reception based rate limiter. The implementation suffered from wavy throughput behavior at the receiver.

In our current implementation, we have multiple token threads. These token threads generate tokens at the capacity of the virtual ports, which is continuously updated on the basis of the feedback from the Congestion Point (CP). Tokens are added a granularity of $2.71\mu s$ and we assume that updates from different cores are visible to the rate limiter with in approximately 250 cycles.

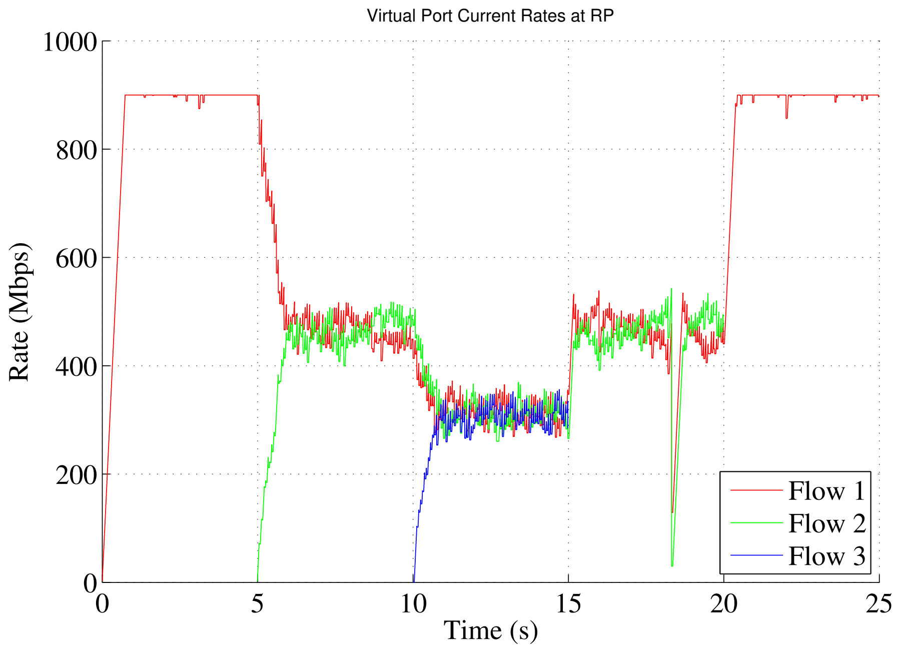

Figure 7.6: Effect of Token Thread based approach on QCN performance

The Token thread approach improves the convergence behavior of the the UDP flows. The fine grain timer based token thread provides a much tighter control over the output rate and the Reaction Point was able to converge to the target rate (from Congestion Point) within a buffer capacity of 12 packets. Along with this, we also reduced the probability of feedback messages from CP so as to avoid consecutive feedback messages from CP.

In our experience, we find couple of QCN parameters which affect the convergence property. We observe that the feedback control loop tends to be more stable when the frequency of feedback is lower and their effect smaller. One of the primary reason behind the wavy characteristics in the previous implementation was large number of consecutive feedback messages. The packet reception based rate limiter was not able to react to these quickly changes, as a result of which the target rate was reduced drastically. To avoid this scenario, we reduced the probability of sending feedback messages from 20% to 5%.

Along with the frequency of feedback generation, the effect of individual messages plays an important role. In the original implementation, the feedback message was quantized in a scale of 128. To reduce the feedback effect, we currently quantize in a scale of 1024. We believe that, this change will increase the sensitivity of the QCN implementation. All these changes contributed to better convergence to fair-share bandwidth (as shown in Figure 7.6).

Chapter 8

Discussion

On the basis of our experiments, we find that the modified token bucket method performs better in comparison to the delay based method. The primary events are used to add tokens at packet reception. The fine grain timing estimates using the non-blocking receive implementation was used to avoid large token additions at the end of the idle phase. This was the key concept which reduced TCP bursts.

The delay-based approach is simple naive approach which does packet pacing on the basis of the calculated bandwidth. The multi-threaded implementation of the NetBump provides an ideal platform for packet pacing as the receive side is decoupled from the send side. The multi-core parallelism can be exploited to share load between the cores and pace packets independently.

We observed that the accuracy of packet pacing reduces by 10% in the presence of multiple flows. This was observed both in case of delay-based and modified token bucket approach. The primary cause of this kind of behavior is `send()` API where the API coalesces packets with low inter packet time or waits for a timeout of approximately $10\mu s$. We tried going through the API source code to manipulate configurations to avoid this kind of timeout. Although we were able to tweak the settings, the observed variation was not predictable. The lack of proper documentation and comment in the myrisnf source code made it difficult to estimate the effect.

The sniffer10g API from myricom has been design to target high throughput even with minimum sized packets. So, some of the design decisions might not be turned for low throughput low latency setup. But, it would great if there was some support from myricom to get reliable packet transmission without buffering or timeouts.

Although we are aware of the DBL library from Myricom which guarantees packets to be sent out in a reliable fashion with least possible latency, we didn't go ahead with the implementation as it breaks the transparency requirements. In an attempt to implement NetBump using the DBL API, we found that raw sockets were not supported, ie, packets over `raw` socket were not accelerated. Moreover, packet reception using DBL API needs the application to be bound to a given port. This means that the NetBump has to be designed as a gateway and special attention is required for control packets. The NetBump could not be placed inline with the network elements.

Chapter 9

Conclusion

On the basis of our experiments, we conclude that NetBump provides an ideal platform for implementing packet pacing. The NetBump provided a very good platform to understand the flow characteristics at different offered load. This provided a very strong motivation behind this work. We have devised multiple approaches for implementing pacers and found that the modified token bucket approach works the best. The ultimate goal for packet pacing is to reduce the number of buffer overflows, and this approach achieves that by recording zero buffer overflows.

With our understanding of token bucket implementation on high throughput networks, we were able to implement similar approach to the rate limiter module of NetBump and significantly improve its performance. This was reflected in better convergence characteristics of 3-flow QCN experiment.

The accuracy of packet pacing in the presence of multiple flows can be increased further by doing driver/firmware level modifications which will enable packet transmission without buffering. For this, we need support from Myricom.

Bibliography

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

[AAM10] Onur Alparslan, Shin'ichi Arakawa, and Masayuki Murata. Comparison of packet switch architectures and pacing algorithms for very small optical ram. In *International Journal on Advances in Internet Technology*, 2010.

[Agg00] Amit Aggarwal. Understanding the performance of tcp pacing. pages 1157–1165, 2000.

[AKE$^+$] Mohammad Alizadeh, Abdul Kabbani, Tom Edsall, Balaji Prabhakar, Amin Vahdat, and Masato Yasuda. Hull: A high bandwidth, ultra-low latency data center fabric architecture. In *Under Submission.*

[AKM04] Guido Appenzeller, Isaac Keslassy, and Nick McKeown. Sizing router buffers. In *IN PROCEEDINGS OF ACM SIGCOMM*, pages 281–292, 2004.

[AlCDR] Hussam Abu-libdeh, Paolo Costa, Austin Donnelly, and Antony Rowstron. Symbiotic routing in future data centers.

[BGC] Philip Buonadonna, Andrew Geweke, and David Culler. An implementation and analysis of the virtual interface architecture. In *ACM/IEEE CDROM 1998.*

[BWCM$^+$] Silas Boyd-Wickizer, Austin T. Clements, Yandong Mao, Aleksey Pesterev, M. Frans Kaashoek, Robert Morris, and Nickolai Zeldovich. An analysis of linux scalability to many cores. In *USENIX OSDI 2010.*

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

[DEA$^+$] Mihai Dobrescu, Norbert Egi, Katerina Argyraki, Byung-Gon Chun, Kevin Fall, Gianluca Iannaccone, Allan Knies, Maziar Manesh, and

Sylvia Ratnasamy. Routebricks: exploiting parallelism to scale software routers. In *ACM SOSP 2009*.

[ESW] David Ely, Stefan Savage, and David Wetherall. Alpine: a user-level infrastructure for network protocol development. In *USITS 2001*.

[flo] Ethernet Flow Control. `http://www.cisco.com/en/US/prod/collateral/switches/ps9441/ps9670/white_paper_c11-542809_ns783_Networking_Solutions_White_Paper.html`.

[GHJ$^+$] Albert Greenberg, James R. Hamilton, Navendu Jain, Srikanth Kandula, Changhoon Kim, Parantap Lahiri, David A. Maltz, Parveen Patel, Sudipta Sengupta, and Generalterms Design Performance Reliability. Vl2:a scalable and flexible datacenter network.

[GK99a] R. J. Gibbens and F. Kelly. Distributed connection acceptance control for a connectionless network. In *Teletraffic Engineering in a Competitive World*, pages 941–952. Elsevier, 1999.

[GK99b] R. J. Gibbens and F. Kelly. Resource pricing and the evolution of congestion control. In *Automatica 35*, pages 1969–1985, 1999.

[JK88] Van Jacobson and Michael J. Karels. Congestion avoidance and control, 1988.

[KMC$^+$00] Eddie Kohler, Robert Morris, Benjie Chen, John Jannotti, and M. Frans Kaashoek. The click modular router. *ACM ToCS*, 2000.

[LSO] Large Segment Offload. `http://en.wikipedia.org/wiki/Large_segment_offload`.

[MAB$^+$] Nick McKeown, Tom Anderson, Hari Balakrishnan, Guru Parulkar, Larry Peterson, Jennifer Rexford, Scott Shenker, and Jonathan Turner. Openflow: enabling innovation in campus networks. *SIGCOMM CCR 2008*.

[MPF$^+$] Radhika Niranjan Mysore, Andreas Pamboris, Nathan Farrington, Nelson Huang, Pardis Miri, Sivasankar Radhakrishnan, Vikram Subramanya, and Amin Vahdat. Portland: A scalable fault-tolerant layer 2 data center network fabric.

[MWMR97] Jeffrey Mogul, Dec Western, Jeffrey C. Mogul, and K. K. Ramakrishnan. Eliminating receive livelock in an interrupt-driven kernel. *ACM Transactions on Computer Systems*, 15:217–252, 1997.

[myr] Myricom Sniffer10G. `http://www.myri.com/scs/download-SNF.html`.

[pin] Ping Pong Latency. `http://www.anandtech.com/show/2143/2`.

[PSP] PSPacer. `http://code.google.com/p/pspacer/`.

[RNN$^+$02] Ashu Razdan, Alok Nandan, Alok N, Ren Wang, Medy Sanadidi, and Mario Gerla. Enhancing tcp performance in networks with small buffers, 2002.

[SEMO06] Vijay Sivaraman, Hossam Elgindy, David Morel, and Diethelm Ostry. Packet pacing in short buffer optical packet switched networks. In *in Proceedings of IEEE Infocom*, 2006.

[Sha01] N. Shah. Understanding network processors. Master's thesis, University of California, Berkeley, Calif., 2001.

[SZC90] Scott Shenker, Lixia Zhang, and David D. Clark. Some observations on the dynamics of a congestion control algorithm, 1990.

[traa] Transmission timer approach for rate based pacing tcp with hardware support.

[Trab] Traffic Control. `http://tldp.org/HOWTO/Traffic-Control-HOWTO/index.html`.

[TTKO10] Ryousei Takano, TomohiroKudoh, Yuetsu Kodama, and Fumihiro Okazaki. High-resolution timer-based packet pacing mechanism on the linux operating system, 2010.

[vEBBV] 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*.

[WBvE] Matt Welsh, Anindya Basu, and Thorsten von Eicken. Atm and fast ethernet network interfaces for user-level communication. *IEEE HPCA 1997*.

[Wik] TCP Offload Engine. `http://en.wikipedia.org/wiki/TCP_Offload_Engine`.