University of Toronto SNL Technical Report TR08-UT-SNL-04-30-00

Time-Sensitive Network Experiments

Neda Beheshti*, Yashar Ganjali†, Monia Ghobadi†, Nick McKeown*, Jad Naous*, Geoff Salmon†

* Computer Systems Laboratory
Department of Electrical Engineering, Stanford University
{nbehesht, nickm, jnaous}@stanford.edu

† Computer Systems and Networks Group
Department of Computer Science, University of Toronto
{yganjali, monia, geoff}@cs.toronto.edu

***Abstract*— Time-sensitive network experiments are difficult. There are major challenges in generating realistic traffic with highly accurate packet arrivals, and monitoring various system performance metrics which require high precision in timing is not trivial. Parts of these problems arise from the fact that generic network software and hardware components do not provide high-precision timing guarantees.**

In this paper, we study the challenges associated with performing time-sensitive network experiments in a testbed including realistic network traffic generation, delay emulation, switching/multiplexing of flows, and collecting high-resolution packet level traffic information. We describe some seemingly minor details that can have significant influence on the results of experiments, and therefore require careful attention. We also show how some of these issues can be addressed by simple software tuning. For others, one needs support from hardware. We show how we have addressed some of these problems using a customized (programmable and configurable) network component called NetFPGA.

I. Introduction

It is commonly believed that the current Internet has significant deficiencies that need to be fixed, however, making changes to the current Internet infrastructure is not easy, if possible at all. Any new protocol or design implemented on a global scale requires extensive and accurate experimental testing in sufficiently realistic settings. While simulation tools provide a means for analyzing different aspects of computer networks, the model they use is abstract and restricted. There are many issues in a practical setup that are not considered or are over-simplified in simulation settings. A real router, for example, with many stages of buffering and a unique architecture can not be modeled accurately by simulation tools.

Networking experiments are intrinsically difficult for several reasons: *i*) Creating a network with multiple routers and an arbitrary topology which can be representative of a real backbone network requires significant resources, *ii*) Network components (like routers) have proprietary architectures, which means it is almost impossible to figure out all their internal details, *iii*) Making changes to network components is not always possible, *iv*) We cannot always use real network traces (for example, when there is a feedback loop in the system) and generating high volumes of artificial traffic which closely resembles operational traffic is not trivial, and *v*) We need a measurement infrastructure which collects traces and measures various metrics throughout the network.

These problems become more pronounced in the context of time-sensitive networking experiments. These are experiments that need very high-precision timing for packet injections into the network, or require packet level traffic measurements with accurate timing. Experimenting with new congestion control algorithms, buffer sizing in Internet routers, and denial of service attacks which use low-rate packet injections [7] are examples of time-sensitive experiments, where a subtle variation in packet injection times can change the results significantly.

In the first part of this paper, we study problems associated with time-sensitive networking experiments. We describe some drawbacks of using commodity network software and hardware components, and show how ignoring precise timing issues can significantly change the outcome of experiments. The second part of this paper, identifies some desirable properties of a network testbed used for time-sensitive experiments. To keep the discussion concrete, we use the example of buffer sizing experiments where the goal is to measure the performance of a network in presence of very small buffers. Note that the same arguments hold for other time-sensitive network experiments. Further, we show how one might alleviate some of these timing problems by tuning software components of the system. For example, by delaying the departure time of packets from a sender, a single traffic generator can produce traffic that appears to originate from multiple senders utilizing links of differing capacities.

Unfortunately, some of the problems associated with time-sensitive experiments cannot be addressed by simple adjustments to the software components. Higher levels of timing guarantees require hardware support, and the type of hardware support necessary depends on the experiment.

In the context of buffer sizing experiments, we show how we have been able to address a lot of timing issues (from traffic generation, to measurement) using NetFPGA boards. NetFPGA is a PCI-based programmable board containing an FPGA, four Gigabit Ethernet ports, an embedded CPU and memory. The board can be programmed to act as an Internet router [2], and we have modified the current implementation of NetFPGA to provide some of the time-sensitive functionalities that are required for our experiments. The changes support high-precision, queue-occupancy measurements, which can be used for analyzing traffic burstiness and network performance. This feature is not readily available in current commercial components and cannot be provided by software-based tools.

The organization of the paper is as follows. In section II we explain the main sources of time inaccuracy that may exist in an experimental testbed. Section III discusses the challenges in setting up a time-sensitive testbed and in generating and monitoring network traffic. Some solutions to overcome these challenges are proposed in section IV. The paper is concluded in section V.

II. Sources of Time Inaccuracy

There are three main sources of time inaccuracy in a testbed experimental setup.

1. Traffic Generators: Generating realistic traffic is one of the key challenges in modeling a network. Experimenting with a small number of traffic source-destination hosts is often done by using a number of Linux boxes as traffic generators. However, creating a large number of connections, in order to model traffic in networks closer to the core of the Internet, with thousands of flows sharing each link, is not a trivial task. The difficulty of such modeling becomes even more obvious when one desires to capture the heterogeneity in link capacities, with only a limited number of physical machines. The core traffic is a mix of packets coming from a broad range of access links; some packets are generated with modems capable of sending only a few kilo bytes per second, while others are generated at Gbps speed.

The accuracy of traffic generated by a generic Linux-based machine is bound by the system timer's resolution. A Linux kernel is not typically capable of providing resolutions higher than 1 ms. With links running at 1 Gbps, even a large packet of 1500 bytes has a transmission time of less than 12 µs. Alternatively, one can use the commercial traffic generators to create a large number of connections. The main problem with such approach would be the lack of accuracy in the implemented protocols. As an example, [3] shows differences between timings in a TCP Reno packet sequence generated by the Sprint's Avalanche traffic generator [1], and that of what is expected from the standard TCP Reno protocol.

2. Switches and Routers: The architecture of commercial routers is not transparent. Their layers of packet buffering and the existence of undocumented buffers make accurate control of the router's buffer size - and consequently the maximum delay that packets can experience inside the router - impossible. An example has been reported in [5] where reducing the buffer size of a Juniper T640 router to values below 1ms, resulted in increasing the packets' delays. Presumably, this is due to the activation of some *hidden* buffers when the size of the main buffer goes below a threshold.

3. Traffic Monitoring Systems: Routers are not typically capable of providing real-time information on per packet arrivals, departures and drops. For a wide range of experiments, however, it is crucial to measure such events accurately over time. Using today's commercial routers, the only way to have such information is to collect packet traces of all packets entering and leaving the router. This requires special purpose equipment, which is not always available and is usually expensive.

III. Challenges of Time-Sensitive Network Experiments

We start by explaining some of the problems associated with performing time-sensitive network experiments in a laboratory testbed with limited resources. Many interesting experiments may not directly map onto the hardware available in a testbed. If the context of an experiment is the Internet core with traffic coming from many endpoints and traversing a hierarchy of routers, then single machines and devices in the testbed will have to substitute for large parts of the emulated network

For time-sensitive experiments, and buffer sizing experiments in particular, we have identified several traffic characteristics that we would like to emulate in a testbed. If an experiment's results are intended to support changes to large scale networks, it is important to consider these affects. In the following sections we will describe the characteristics and their motivations. Section IV discusses several approaches for achieving these network conditions in a testbed.

A. *Packet Delay*

The round trip time (RTT) between two endpoints is an important property of the network. It measures the minimum time it takes to send a packet and receive a reply. In particular, the RTT is important in protocols such as TCP that rely on positive acknowledgements to control the data transmission rate. The RTT directly influences how soon a sender can expect to receive a positive acknowledgement for any sent packet. Round trip packet delays that occur in a testbed do not reflect those in the core of the Internet. In a testbed that physically resides in several rooms with short wires connecting computers, the observed RTT can be less than 200µs. Whereas RTTs on the Internet vary widely and can be as large as a few hundred milliseconds.

To emulate the long Internet paths in a testbed it is necessary to add delay to every packet. Also, the RTT of connections between different pairs of endpoints in the Internet obviously varies. If one machine in a testbed is standing in for many machines, then different packets sent by that machine may need to be delayed different amounts.

B. *Flow Mixture*

The context of an experiment affects the expected ordering of packets from different flows. Consider the packets of a single flow as they traverse the sender's access link, the Internet core and the receiver's access link. Near the edges of the Internet, close to the sender or receiver, the packets of the flow may be back to back with other packets of the same flow. However in the Internet core, where there are many flows, packets of one flow will be intermixed with packets of other flows. Therefore, if the testbed traffic is to emulate the Internet core traffic, it should consist of multiplexed flows with their packets being intermixed. Since there are a limited number of physical links, each carrying a large number of flows in the testbed, this mixing will not happen naturally at the switches and routers. The traffic generating hosts must be configured to inject packets belonging to different emulated users in a mixed way to the output physical links.

In the buffer sizing experiments, the ordering of packets can greatly affect the throughput and the fairness. If packets of an individual flow arrive back to back in a burst, then a buffer overflow may result in a large number of packet drops for a single flow, while other flows might not experience any drops.

C. *Packet Pacing*

In addition to the relative ordering of packets from different flows, an experiment's context will also affect the interarrival time of packets belonging to the same flow. As in Section III-B, consider packets belonging to a flow as they traverse the sender's access link, the Internet core and the receiver's access link. In this case, the disparate link bandwidths encountered by the packets have an important effect on the packets' pacing.

Given that backbone networks typically run at 2.5 Gbps or 10 Gbps, and will run even faster in the future, almost every access link runs much slower than the backbone network. For most flows (e.g. where the user is connected to the network via modem, DSL, cable or from a 10Mbps or 100Mbps Ethernet), packets will be naturally spaced out by the network. As packets of a flow cross the boundaries from slower to faster links, the spacing between them should grow. At these boundaries are store and forward routers, which wait for a packet to completely arrive before sending it. Ignoring any potential queueing, the router will be able to send the previous packet before the next one has completely arrived. Packets arriving back to back on the slow link will depart with a gap between them on the fast link. Without queueing, as packets of a given flow traverse faster and faster links the gaps between them will grow.

If a testbed is meant to approximate the Internet core, then (a fraction of) the generated flows must be made paced in order to emulate the difference between the core and the access bandwidths.

D. *Measurement*

Obtaining meaningful experimental results requires accurate measurements. In our testbed there are 3 different domains with properties that need to be measured: inside end hosts, inside network devices like routers, and on the wire itself. Each presents different difficulties and measuring some performance metrics require more precise timing information than others.

The easiest properties to measure are those within the end hosts. Properties such as the number and size of flows and size of congestion windows are relatively easy to obtain. Operating Systems, like Linux, provide statistics about their network stacks. Note that these properties do not depend on precise timings.

Measuring the internal state of a router is more difficult. Commercial routers are usually black boxes exposing few internal details. For buffer-sizing experiments, precise queue occupancy measurements are critical and require support from the router itself. To obtain the necessary transparency and flexibility we are using routers based on NetFPGA, which is discussed in more detail in Section IV-D and the Appendix.

The final accurate measurements we need are from packets travelling on the wire. Although OSs running on commodity hardware can provide information about each packet sent and received on a network link, the timing accuracy is often not sufficient. There are too many other considerations in the designs of such systems to also provide accurate timings. For example, to save CPU cycles, network cards can coalesce multiple packet arrivals together and signal a single arrival to the system. From the OS's perspective, the packets will have all arrived nearly simultaneously and the true timing information is lost.

To obtain accurate timings of packets on the wire hardware support is necessary. Here we use the reporting capabilities of the NetFPGA again. Precise timings for arrivals to and departures from the router's output queues also provide good timings for the packets on the wire. More details are in Section IV-D.

IV. Overcoming the Challenges

The following sections suggest possible modifications to components of a testbed to obtain a more realistic environment for time-sensitive experiments. We examine a number of settings in a commodity Linux server which can benefit from tuning. We focus on settings that could alter the measured metrics in time-sensitive experiments.

A. Delay

We have tried two different approaches for simulating network delay: NIST Net [4] and Netem [6]. Both are software for simulating different network conditions within Linux. They can be used to add delay to any packet matching filters based on IP addresses and port numbers.

NIST Net delays packets as they are arriving at the system, while Netem usually delays packets as they are departing but can be configured to delay arriving packets as well. Where the software should be installed in a testbed depends on which packets are meant to be delayed. In our experiments we are delaying the ACK packets because delaying the larger data packets would require more memory. Also, we added the delay in a machine that is acting as a router in the middle of the network path to try to avoid any unknown interactions between sending or receiving TCP packets and introducing delay.

With either software, setting a sufficient queue size is important. The queue must be large enough to hold all of the packets that can arrive in a delay period, otherwise packets can be dropped. Both methods can also simulate other network conditions by reordering, duplicating or even corrupting packets.

NIST Net is an older solution and will run on older Linux kernels, but is a little more difficult to install. Conversely, Netem is included in Linux kernels and requires no installation, but it only began using high resolution timers as of kernel 2.6.22 (released July 8, 2007), according to its website [8]. Prior to that kernel, the minimum granularity of the delay was only 1 ms which could significantly affect the interarrival times of delayed packets on a 1 Gbps link. In our experiments, we usually use NIST Net to add delay because we have observed some machines lock up and become unresponsive when delaying with Netem under high network loads. For future experiments, we suggest evaluating the stability and performance of both Netem and NIST Net before choosing one.

B. Tuning End Hosts

During our experimental studies, we observed that modifying TCP parameters such as the congestion control algorithm, the receive and congestion window sizes and the TCP acknowledgement options is important to create a more realistic setup for time-sensitive experiments. In Linux these parameters are configured using either the sysctl interface or the proc file system. The hosts in our testbed setup are Dell Power Edge 2950 machines running Debian GNU/Linux 4.0r3 (codename Etch) with kernel 2.6.18.

In addition to adjusting software parameters, there are some important hardware related factors that we must pay attention to and tune if needed.

TCP segmentation offload (TSO): If supported, this feature allows the OS to send segments much larger than the supported maximum transmission unit (MTU) of the physical layer to the network interface card. Once there, the packet is segmented into multiple MTU sized packets. This option frees the CPU from segmentation overhead but can cause bursts of back to back packets on the wire. Disabling TSO is especially important if any type of sender-side packet pacing is used, as explained in Section IV-C. We performed a number of experiments to assess TSO's effect. We were mainly interested in the resulting flow mixture and burstiness, which led us to choose the following distance-probability metric: For every packet in the aggregate traffic, determine its TCP flow and look at the subsequent packets to find those belonging to the same flow. Calculate the probability of this match occurring for each offset ahead in the packet sequence. In practice we looked 100 packets ahead.

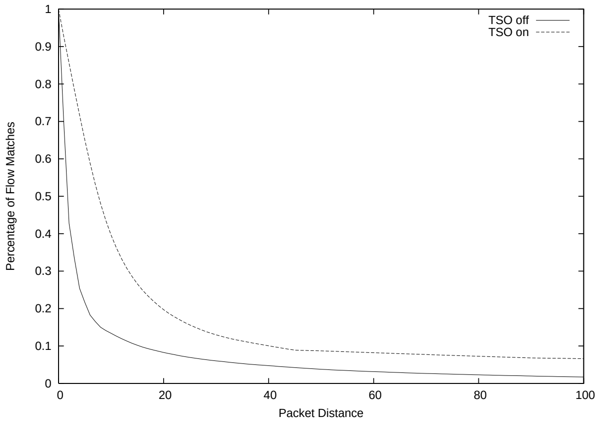


Fig. 1. The Effect of TSO on traffic burstiness

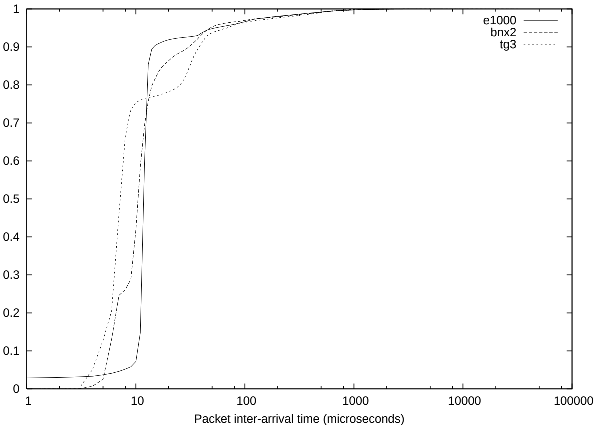


Fig. 2. CDF of packet interarrival times for different NIC cards

Although it is difficult to interpret the resulting data from a single trace, this metric is useful for comparing different traces.

Figure 1 illustrates the effect of TSO on traffic burstiness by comparing the distance-probability metric described above for two experiments *TSO on* and *TSO off*. Both experiments involve 400 long-lived flows on a single network path with RTT of 100 ms. The only difference between the two experiments is whether TSO is enabled on the network cards. The figure shows that enabling TSO here increases the likelihood that a subsequent packet will be from the same flow. In other words, packets of a particular flow are more likely to be followed closely by packets of the same flow. However, the TSO off experiment shows that disabling TSO would notably decrease the probability of having two subsequent packets of the same flow being close to each other and hence increases the flow mixture.

Interrupt Coalescing (IC): High-bandwidth network interfaces usually use Interrupt Coalescing. Rather than raising interrupts for every single arrival and departure, packets are collected and one single interrupt is generated for multiple packets. IC decreases the per-packet interrupt processing overhead. However, IC also introduces queueing delays and alters the pacing of packets. It has been shown in [10] that enabling IC can make several packets appear as if they form a burst, even though that may not be the case and can affect the correlation structure of packet interarrivals in short timescales. IC can also cause bursty delivery of TCP ACKs to the sender transmission which leads to bursty transmission of data segments.

Turning off coalescing appears to be easier said than done in some cases. Changing the IC setting of the NIC card using ethtool appears to have different outcome depending on the brand and driver of the NIC card. As an example, Figure 2 shows the difference between CDF of interarrival times of packets when IC is turned off using ethtool for Intel NIC card with e1000 driver, Broadcom NIC card with bnx2 driver and Broadcom NIC card with tg3 driver. It can be inferred from this figure that Broadcom NIC cards have more variations in the interarrival times than the e1000 card. The e1000 has a spike at 12µs which is the expected interarrival time for a 1 Gbps line with RTT 100ms and packet size 1500B. Thus, we concluded that tuning IC is dependent on the hardware type and also driver implementation.

Another major difference between different NIC card manufacturers that we observed was the setup of IEEE 802.3x Ethernet flow control standard. The intent of Ethernet flow control is to prevent loss in the network by providing back pressure to the sending NIC cards that are going too fast to avoid loss. While this sounds like a good idea on the surface, higher-layer protocols like TCP were designed to rely on loss as a signal that they should send more slowly. We noticed that, by default, Intel cards have flow control enabled whereas Broadcom cards disable it. Therefore, when performing time-sensitive network experiments, one should be extra cautious regarding the setting of Ethernet flow control. For experiments on traffic burstiness and flow mixture, we recommend turning off the Ethernet flow control.

C. Packet Pacing

As described in Section III-C, the interarrival time between packets of a TCP flow needs to be simulated in the testbed during certain experiments. We tried a number of approaches to achieve satisfactory pacing of packets. The following paragraphs detail those attempts

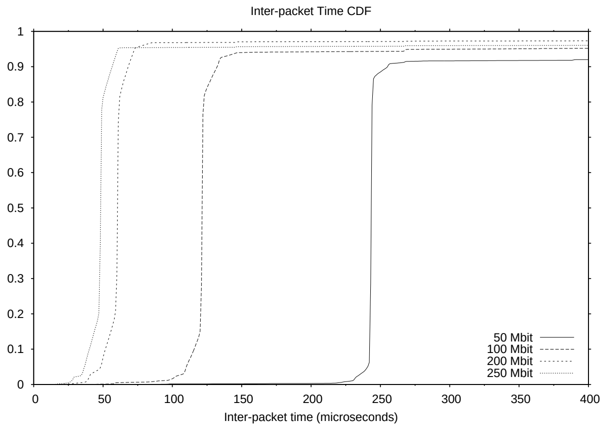


Fig. 3. Packet Pacing Without TSO.

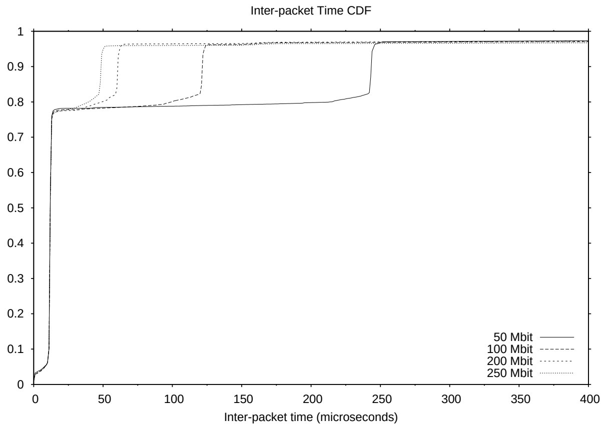


Fig. 4. Packet Pacing With TSO.

and some of the problems encountered.

Initially we tried to create pacing by using slower access links. The simplest method appeared to be to limit the speed that our network cards negotiated. With a 1 Gbps link as our core link, we forced our endpoints' network cards to perform at either 100 Mbps or 10 Mbps. The ratio of 1 Gbps core to 100 Mbps access links is only 10X which does not reflect the Internet today. Endpoints with 10 Mbps access links improve the ratio, but the testbed would now require 100 machines to saturate the 1 Gbps link. In most testbeds this is impractical.

Our next attempt used facilities provided by Linux to shape the traffic leaving on a network interface. The queueing discipline used by the transmit queue leading to a network device can be modified, and it is possible to add a token bucket filter to shape outgoing traffic. In the simple case this shaping will affect all traffic leaving the network device and should simulate flows from a single endpoint with a slow access link. To simulate many hosts we require different pacing for different TCP flows, which can be achieved with a hierarchical token bucket with multiple classes and filters matching against masked portions of the TCP port number. Unfortunately, using a hierarchical token bucket caused stability problems. With 16 token bucket classes, under high network load, running the 2.6.24 Linux kernel, a sending machine locked up and required rebooting. We have not determined an exact cause for this yet.

While investigating Linux's support for token buckets we also experimented with PSPacer [11], which paces packets by injecting gap packets between the real packets. By knowing the speed of the link and controlling the number and size of the gap packets, PSPacer can precisely control the timing of packets without using timers. The trade-off is packets are being sent at the line rate even when the data rate has been limited by pacing. Also it is important to note that the gap packets are actually Ethernet PAUSE frames, so Ethernet flow control must be turned off. With fast testbed machines, neither sending at the line rate or turning off flow control should pose a problem.

Figures 3 and 4 shows the packet timings that result from pacing packets. The MTU of the network is 1500 bytes which can be transmitted in roughly 12 μs at 1 Gbps speeds. In this example, the machine sending the packets uses PSPacer to simulate four different links with the speeds 50 Mb, 100 Mb, 200 Mb and 250 Mb, which can transmit 1500 bytes in roughly 240, 120, 60 and 48 μs, respectively. Each of the 100 TCP flows involved is assigned to a speed based on the last 2 bits of its destination port, which distributes them approximately equally between each of the simulated speeds. The figure shows the interarrival times of the fours sets of packets received at another computer.

Without TSO, described in Section IV-B, the majority of the packets have the expected interarrival time for their speed, so we can conclude the pacing is working. However, with TSO enabled most of the interarrival times are 12 μs, which is the minimum transmission time for MTU-sized packets. Disabling TSO is very important when employing packet pacing inside Linux. If enabled then the packets that are paced can actually contain multiple packets. Even though their arrival at the network card is paced, individual packets can then appear back to back on the wire after being segmented in hardware. As the figure shows, the TSO and this form of pacing are not compatible.

When applying pacing, it is important to realize that the RTT of the flows on the paced link may increase. Using either PSPacer or token buckets to emulate multiple, slower link speeds will creates multiple queues for outgoing packets to wait on. By default, these queues are the same size as the single outgoing queue used by the network interface when not pacing packets. If these queues fill up, the effective RTT will increase more than it would without pacing because the queues are being served at slower rates and will take longer to empty. To maintain a relatively fixed RTT across different pacing rates one should increase or decrease both the queue size and the sending rate together.

D. Precise Timing Measurements

As discussed in Section III-D, a general purpose computer cannot provide the precise measurements required for some experiments. For example, to adequately understand the results of an experiment involving small packet buffers we need accurate timing information. Specifically, we need to know the arrival time of packets to the router and the state of the router's packet queues. Gathering this information requires hardware support.

We have modified a NetFPGA router design to provide the necessary measurement support. The output queues of the router can be instrumented to monitor packets arriving, departing and dropping. When any of those three events occur, the size of the packet and the current clock of the NetFPGA, which has a 8 ns granularity, are recorded. Multiple events are then collected in a single event packet which is periodically sent out a specified router port. The event packets contain enough information to reconstruct the queue evolution in time.

Figure 5 shows a sample of the resulting occupancies and packet drop information for a single NetFPGA router queue. In this example, the queue size has been limited to 128 packets, and drops are evident whenever the queue occupancy reaches that value. Setting queue sizes and measuring timings with this level of granularity is essential for small buffer experiments, but it is not supported by commercial routers. Also, the transparency of the NetFPGA hardware allows us to be confident that queue size is set exactly and the reported measurements are precise.

In the current implementation, per-flow metrics are very difficult to obtain. Simply adding the source-destination flow tuple to each event record will increase bandwidth requirements dramatically. Future work will explore classifying events by the flow they belong to and building flow-specific event packets to study how flows interact in a router and how they affect queue occupancies. The appendix provides more information on the implementation of the queue monitoring logic.

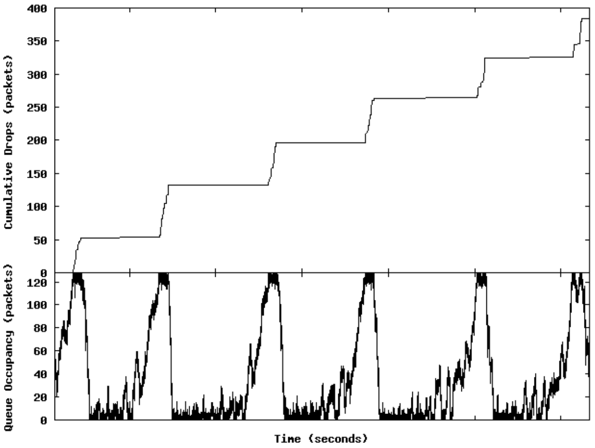


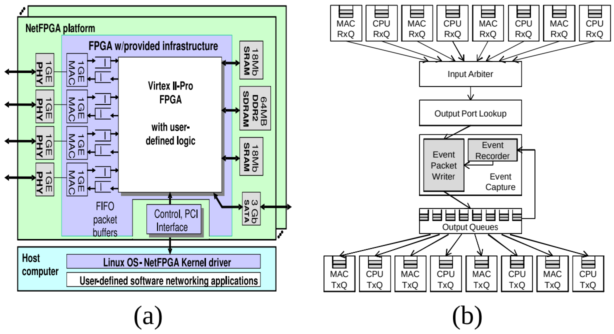
Fig. 5. Precise queue occupancy and packet drops monitored with the NetFPGA

V. Conclusion

Network experiments are an inseparable part of design and deployment of new techniques and protocols in any network, especially the Internet. In this paper, we study some of the challenges associated with time-sensitive networking experiments. We bring attention to the minor software/hardware related details that one needs to consider in order to get realistic results in a time-sensitive network experiment. We also show how some of these timing issues can be alleviated by tuning system components, and how some of these issues need support and accuracy that can only be provided by hardware components of the system. This shows the importance of having configurable/programmable networking hardware that can be modified quickly and tuned for specific experiments. We employ one such component (NetFPGA) throughout the paper and show how we have used it to address time-sensitivity issues that we were not able to solve by modifying software parts of the system.

Acknowledgements

We would like to thank Amin Tootoonchian for his help with our experiments. We are grateful to Sara Boluki for her help with the NetFPGA queue monitoring module, as well as other Stanford HPNG members who supported the NetFPGA system. University of Toronto's part of this work was supported by NSERC Discovery, NSERC RTI, Cisco Systems, as well as DoD-N Award No. W911NF-07-1-002431.



(a) (b)

Fig. 6. (a) NetFPGA 2.1 block diagram, (b) Router with event capture subsystem.

APPENDIX

NetFPGA Router

The NetFPGA is platform that was designed for network hardware teaching and research [9]. It consists mainly of a PCI form-factor board that has an FPGA, four 1-Gigabit Ethernet ports, and memory in SRAM and DRAM. Figure 6(a) shows the components in more details. Several reference designs have been implemented on NetFPGA: An IPv4 router, a learning Ethernet switch, and a NIC.

All the designs run at line-rate and follow a simple five-stage pipeline. The first stage, the *Rx Queues*, receives packets from the Ethernet and from the host CPU via DMA and handles any clock domain crossings. The second stage, the *Input Arbiter*, selects which input queue in the first stage to read a packet from. This arbiter is currently implemented as a packetized round-robin arbiter.

The third stage, the *Output Port Lookup*, implements the design specific functionality and selects the output destination. In the case of the IPv4 router, the third stage will check the IP checksum, decrement the TTL, perform the longest prefix match on the IP destination address in the forwarding table to find the next hop, and consult the hardware ARP cache to find the next hop MAC address. It will then perform the necessary packet header modifications and send the packet to the fourth stage.

The fourth stage is the *Output Queues* stage. Packets entering this stage are stored in separate SRAM output queues until the output port is free. At that time, a packet is pulled from the SRAM and sent out either to the Ethernet or to the host CPU via DMA. The fifth stage, the *Tx Queues*, is the inverse of the first stage and handles transferring packets from the FPGA fabric to the I/O ports.

The Buffer Monitoring design augments the IPv4 router by inserting an *Event Capture* stage between the Output Port Lookup and the Output Queues. It allows monitoring the output queue evolution in real-time with single cycle accuracy. This stage consists of two main components: an *Event Recorder* module and a *Packet Writer* module.

The Event Recorder captures the time when signals are asserted and serializes the events to be sent to the Packet Writer, which aggregates the events into a packet by placing them in a buffer. When an event packet is ready to be sent out, the Packet Writer adds a header to the packet and injects it into the Output Queues. From there the event packet is handled just like any other packet. Figure6(b) shows the router's five stage pipeline augmented with the Event Capture stage to monitor the output queues.

REFERENCES

[1] Avalanche traffic generator. http://www.spirentcom.com.

[2] The netfpga project. http://www.netfpga.org/.

[3] N. Beheshti, Y. Ganjali, M. Ghobadi, N. McKeown, and G. Salmon. Experimental study of router buffer sizing. Technical Report TR08-UT-SNL-05-09-00, University of Toronto, May 2008.

[4] M. Carson and D. Santay. Nist net: a linux-based network emulation tool. *SIGCOMM Comput. Commun. Rev.*, 33(3):111–126, July 2003.

[5] Y. Ganjali. *Buffer Sizing in Internet Routers*. PhD thesis, Stanford University, Department of Electrical Engineering, March 2007.

[6] S. Hemminger. Network emulation with netem. In *Linux Conf Au*, April 2005.

[7] A. Kuzmanovic and E. Knightly. Low-rate tcp-targeted denial of service attacks (the shrew vs. the mice and elephants). *In Proceedings of ACM SIGCOMM*, August 2003.

[8] The linux foundation website: http://www.linux-foundation.org/en/net:netem, 2008.

[9] J. W. 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 *MSE '07: Proceedings of the 2007 IEEE International Conference on Microelectronic Systems Education*, pages 160–161, Washington, DC, USA, 2007. IEEE Computer Society.

[10] R. Prasad, M. Jain, and C. Dovrolis. Effects of interrupt coalescence on network measurements. *Passive and Active Measurements (PAM) conference*, April 2004.

[11] R. Takano, T. Kudoh, Y. Kodama, M. Matsuda, H. Tezuka, and Y. Ishikawa. Design and evaluation of precise software pacing mechanisms for fast long-distance networks. *3rd Intl. Workshop on Protocols for Fast Long-Distance Networks (PFLDnet)*, 2005.