# NetFPGA-based Precise Traffic Generation

Geoffrey Salmon, Monia Ghobadi, Yashar Ganjali
Department of Computer Science
University of Toronto
{geoff, monia, yganjali}@cs.toronto.edu

Martin Labrecque, J. Gregory Steffan
Department of Electrical and Computer Engineering
University of Toronto
{martinl,steffan}@eecg.toronto.edu

## ABSTRACT

Generating realistic network traffic that reflects different network conditions and topologies is crucial for performing valid experiments in network testbeds. Towards this goal, this paper presents Precise Traffic Generator (PTG), a new tool for highly accurate packet injections using NetFPGA. PTG is implemented using the NetThreads platform, an environment familiar to a software developer where multithreaded C programs can be compiled and run on the NetFPGA. We have built the PTG to take packets generated on the host computer and transmit them onto a gigabit Ethernet network with very precise inter-transmission times. Our evaluations show that PTG is able to exactly reproduce packet inter-arrival times from a given, arbitrary distribution. We demonstrate that this ability addresses a real problem in existing software network emulators — which rely on generic Network Interface Cards for packet injections — and predict that the integration of PTG with these emulators would allow valid and convincing experiments which were previously difficult or impossible to perform in the context of network testbeds.

## 1. INTRODUCTION

Making any changes to the current Internet infrastructure is extremely difficult, if possible at all. Any new network component, protocol, or design implemented on a global scale requires extensive and accurate testing in sufficiently realistic settings. While network simulation tools can be very helpful in understanding the impact of a given change to a network, their predictions might not be accurate due to their simplified and restricted models and settings. Real network experiments are extremely difficult too: network operators usually do not like any modifications to their network, unless the proposed changes have been tested exhaustively in a large scale network. The only remaining option for testing the impact of a given change is using testbeds for network experiments.

To have meaningful experiments in a testbed, one must have realistic traffic. Generating high volumes of traffic is intrinsically difficult for several reasons. First, it is not always possible to use real network traces, as traces do not maintain the feedback loop between the network and traffic sources (for example the TCP closed-loop congestion feedback). Second, using a large number of machines to generate the required traffic is usually not an option, as it is extremely costly, and difficult to configure and maintain. Finally, depending on the purpose of the experiment, the generated traffic might have different sensitivity requirements. For example, in the context of testing routers with tiny buffers (e.g. 10-20 packets of buffering) even the slightest change in packet injection patterns can have major implications for the results of the experiment [1], whereas in the study of capacity planning techniques, the results are only sensitive to the aggregate bandwidth over relatively coarse timescales [2].

Commercial traffic generators are very useful for some experiments, but they have their own drawbacks. They are usually very expensive and their proprietary nature makes them very inflexible for research on new techniques and protocols. Also, it has been shown that their packet injection times are not accurate enough for time-sensitive experiments [3]. Additionally, commercial traffic generators do not always implement network protocols accurately: for example, Prasad *et al.* [4] describe differences observed between a TCP Reno packet sequence generated by a commercial traffic generator and the expected behavior of the standard TCP Reno protocol.

An alternative to a commercial traffic generator is open source packet generation software, where a small number of machines are used to generate high volumes of realistic traffic [2, 5]. These tools usually rely on generic hardware components that, depending on the vendor and model, can vary in their behavior; therefore, the output traffic might be inaccurate. For example, generic Network Interface Cards (NICs) usually provide little or no guarantees on the exact packet injection times. As a result, the actual traffic pattern depends on the NIC model and, depending on what model is used, significant differences in the generated traffic can be observed [1, 3].

Researchers at Stanford University have developed a packet generator that is capable of generating more precise traffic [6] (hereafter referred to as SPG), addressing the problems described above. The Stanford system is based on *NetFPGA*, a PCI-based programmable board containing an

FPGA, four gigabit Ethernet ports, and memory. The SPG system generates more accurate traffic by precisely replicating the transmission times recorded in a `pcap` trace file, similar to the operation of the `tcpreplay` software program; this method eliminates the dependence between the generated traffic and the NIC model.

While the traffic that SPG generates is more realistic than many prior approaches, it has several limitations. Because the trace files are based on past measurements, the closed-loop feedback for TCP sources (and any other protocol that depends on the feedback from the system) is not accurately captured. Furthermore, replaying a prerecorded trace on a link with different properties (such as capacity and buffer size) does not necessarily result in realistic traffic. Finally, SPG can only (i) replay the exact packet inter-arrival times provided by the trace file, or (ii) produce fixed inter-arrival times between packets (i.e., ignoring the variation of packet timings from the original trace).

In this paper, we introduce *Precise Traffic Generator* (PTG), a NetFPGA-based packet generator with highly-accurate packet injection times that can be easily integrated with various software-based traffic generation tools. PTG has the same accuracy level as SPG, but provides two key additional features that make it useful in a larger variety of network experiments: (i) packets in PTG are created dynamically and thus it can model the closed-loop behavior of TCP and other protocols; (ii) PTG provides the ability to follow a realistic distribution function of packet inter-arrival times such as the probability distributed functions presented by Katabi *et al.* [7][1].

PTG is built on *NetThreads* [8], a platform for developing packet processing applications on FPGA-based devices and the NetFPGA in particular. NetThreads is primarily composed of FPGA-based multithreaded processors, providing a familiar yet flexible environment for software developers: programs are written in `C`, and existing applications can be ported to the platform. In contrast with a PC or NIC-based solution, NetThreads is similar to a custom hardware solution because it allows the programmer to specify accurate timing requirements.

# 2. PRECISE TRAFFIC GENERATOR

In this section we present PTG, a tool which can precisely control the inter-transmission times of generated packets. To avoid implementing a packet generator in low-level hardware-description language (how FPGAs are normally programmed), we use NetThreads instead. We generate packets on the host computer and send them to the NetFPGA over the PCI bus. NetThreads provides eight threads that prepare and transmit packets. This configuration is particularly well-suited for packet generation: (i) the load of the threads' execution is isolated from the load on the host processor, (ii) the threads suffer no operating system overheads, (iii) they can receive and process packets in parallel, and (iv) they have access to a high-resolution system clock (much higher than that of the host clock).

[1]Here, we assume the distribution of different flows and the packet injection times are known a priori, and our goal is to generate traffic that is as close as possible to the given distribution.

In our traffic generator, packets are sent out of a single Ethernet port of the NetFPGA, and can have any specified sequence of inter-transmission times and valid Ethernet frame sizes (64-1518 bytes). PTG's main objective is to precisely control the transmission times of packets which are created in the host computer, continually streamed to the NetFPGA, and transmitted on the wire. Streaming packets is important because it implies the generator can immediately change the traffic in response to feedback. By not requiring separate load and send phases for packets, the PTG can support closed-loop traffic. PTG can easily be integrated with existing traffic generators to improve their accuracy at small time scales.

Let us start by going through the life cycle of a packet through the system, from creation to transmission. First a userspace process or kernel module on the host computer decides a packet should be sent at a particular time. A description of the packet, containing the transmission time and all the information necessary to assemble the packet is sent to the NetFPGA driver. In the driver, multiple packet descriptions are combined together and copied to the NetFPGA. Combining descriptions reduces the number of separate transfers required and is necessary for sending packets at the line rate of 1Gb/s. From there, the packet descriptions are each given to a single thread. Each thread assembles its packet in the NetThreads' output memory. Next, another thread sends all of the prepared packets in the correct order at the requested transmission times. Finally, the hardware pipeline of the NetFPGA transmits the packets onto the wire.

In the rest of this section we explain each stage of a packet's journey through the PTG in greater detail. We also describe the underlying limitations which influence the design.

**Packet Creation to Driver:** The reasons for and context of packet creation are application-specific. To produce realistic traffic, we envision a network simulator will decide when to send each packet. This simulation may be running in either a userspace process, like `ns-2` [9], or a Linux kernel module, as in ModelNet [10]. To easily allow either approach, we send packets to the NetFPGA driver using Linux NetLink sockets, which allow arbitrary messages to be sent and received from either userspace or the kernel. In our tests and evaluation, we create the packets in a userspace process.

At this stage, the messages sent to the NetFPGA driver do not contain the entire packet as it will appear on the wire. Instead, packets are represented by minimal descriptions which contain the size of the packet and enough information to build the packet headers. Optionally, the descriptions can also include a portion of the packet payload. The parts of the payload that are not set will be zeroes when the packet is eventually transmitted. In Section 4, we mention a work-around that may be useful when the contents of packet payloads are important to an experiment.

**Driver to NetThreads:** We modified the driver provided with NetFPGA to support the PTG. Its main task is to copy the packet descriptions to the NetFPGA card using DMA over the PCI bus. It also assembles the packet headers and computes checksums.

Sending packets to the NetFPGA over the PCI bus introduces some challenges. It is a 33MHz 32-bit bus with a top theoretical transfer rate of 1056 Mb/s, but there are significant overheads even in a computer where the bus is not shared. Most importantly, the number of DMA transfers between the driver and NetFPGA is limited such that the total throughput is only 260Mb/s when individually transferring 1518 byte packets. Limitations within the NetFPGA hardware pipeline mean we cannot increase the size of DMA transfers to the NetFPGA enough to reach 1 Gb/s. Instead we settle for sending less than 1 Gb/s across the PCI bus and rebuilding the packets inside the NetFPGA. Currently, the packet payloads are simply zeroed, which is sufficient both for our evaluation and for many of the tests we are interested in performing with the PTG.

To obtain the desired throughput, the driver combines the headers of multiple packets and copies them to the NetFPGA in a single DMA transfer. Next, the NetFPGA hardware pipeline stores them into one of the ten slots in the input memory of the NetThreads system. If there is no empty slot in the memory then the pipeline will stall, which would quickly lead to dropped packets in the input queue of the NetFPGA. To avoid this scenario, the software running on the NetThreads platform sends messages to the driver containing the number of packets that it has processed. This feedback allows the driver to throttle itself and to avoid overrunning the buffers in the NetFPGA.

**NetThreads to Wire:** The PTG runs as software on the NetThreads platform inside the NetFPGA. The driver sends its messages containing the headers of multiple packets and their corresponding transmission times, and the PTG needs to prepare these packets for transmission and send them at the appropriate times.

To achieve high throughput in NetThreads, it is important to maximize parallelism and use all eight threads provided by NetThreads. In the PTG, one thread is the sending thread. By sending all packets from a single thread we can ensure packets are not reordered and can easily control their transmission time. Each of the other seven threads continually pop a job from a work queue, performs the job and returns to the queue for another job. There are currently two types of jobs: 1) receive and parse a message from the driver and schedule further jobs to prepare each packet, and 2) prepare a packet by copying its header to the output memory and notifying the sending thread when complete.

When preparing outgoing packets, most of the work performed by the threads involves copying data from the input memory to the output memory. As described in [8], the buses to the input and output memories are arbitrated between both processors. For example, in a single clock cycle only one of the processors can read from the input memory. Similarly only one processor can write to the output memory in a given cycle. Fortunately, these details are hidden from the software, and the instructions squashed by an arbiter will be retried without impacting the other threads [11]. At first, it may appear that only one processor can effectively copy packet headers from the input memory to the output memory at any given time. However, the instructions that implement the `memcpy` function contain alternating loads and stores. Each 32-bit value must be loaded from the input memory into a register and then stored from the register into the output memory. Therefore, if two threads are copying packet headers, their load and store instructions will naturally interleave, allowing both threads to make forward progress.

# 3. EVALUATION

In this section we evaluate the performance of PTG by focussing on its accuracy and flexibility and also present measurements of an existing network emulator which clearly demonstrate the need that PTG fulfills. By contrast, previous works presenting traffic generators usually evaluate the realism of the resulting traffic [2,12]. While their evaluations present large test runs and often attempt to replicate the high-level properties seen in an existing network trace, our evaluation of PTG reflects its intended use; PTG is meant to complement existing traffic generators by allowing them to precisely control when packets are transmitted. Thus, we present relatively simple experiments where the most important metric is the accuracy of packet transmission times.

We perform our evaluations using Dell Power Edge 2950 servers running Debian GNU/Linux 5.0.1 (codename Lenny) each with an Intel Pro/1000 Dual-port Gigabit network card and a NetFPGA. In each test, there is a single server sending packets and a single server receiving packets and measuring their inter-arrival times. In the experiment described in Section 3.2, there is an additional server running a software network emulator which routes packets between the sender and receiver. The servers' network interfaces are directly connected – there are no intermediate switches or routers.

Since PTG's main goal is to transmit packets exactly when requested, the measurement accuracy is vital to the evaluation. As we discussed before, measuring arrival times in software using `tcpdump` or similar applications is imprecise; generic NICs combined with packet dumping software are intrinsically inaccurate at the level we are interested in. Therefore, we use a NetFPGA as the NIC to measure packet inter-arrival times at the receivers. Those receiving NetFPGAs are configured with the "event capturing module" of the NetFPGA router design [1] which provides timestamps of certain events, including when packets arrive at the router's output queues. To increase the accuracy of the timestamps, we removed two parts of the router pipeline that could add a variable delay to packets before they reach the output queues. This simplification is possible because we are only interested in measuring packets that arrive at a particular port and the routing logic is unnecessary. The timestamps are generated in hardware from the NetFPGA's clock and have a granularity of 8ns. We record these timestamps and subtract successive timestamps to obtain the packet inter-arrival times.

## 3.1 Sending Packets at Fixed Intervals

The simplest test case for the PTG is to generate packets with a fixed inter-transmission time. Comparing the requested inter-transmission time with the observed inter-arrival times demonstrates PTG's degree of precision.

As explained in Section 2, PTG is implemented as software running on what has previously been a hardware-only net-

work device, the NetFPGA. Even executing software, NetThreads should provide sufficient performance and control for precise packet generation. To evaluate this, we compare PTG's transmission times against those of SPG, which is implemented on the NetFPGA directly in hardware.

| Requested Inter-arrival (ns) | PTG mean error (ns) | SPG mean error (ns) |
|---|---|---|
| 1000000 | 8.57 | 8.46 |
| 500000 | 4.41 | 5.12 |
| 250000 | 3.89 | 3.27 |
| 100000 | 3.87 | 1.49 |
| 50000 | 1.87 | 0.77 |
| 25000 | 1.04 | 0.42 |
| 20000 | 0.82 | 0.35 |
| 15000 | 0.62 | 0.35 |
| 13000 | 0.54 | 0.27 |

**Table 1: Comparing the mean error in ns between the Precise Traffic Generator (PTG) and Stanford's packet generator (SPG)**

Table 1 shows the mean absolute error between the observed inter-arrival times and the requested inter-transmission times for various requested intervals. For each interval, we transmit 100000 packets of size 1518 bytes with both PTG and SPG. For inter-transmission times less than 50 $\mu$s, the average absolute error is less than 2ns for both packet generators. Note that clock period of 1000BASE-T gigabit Ethernet is 8ns, so an average error of 2ns implies most of the inter-transmission times are exactly as requested. This shows that even though NetThreads is executing software, it still allows precise control of when packets are transmitted.

Although both packet generators are of similar accuracy, SPG has a limitation that makes it unsuitable for the role we intend for the PTG. The packets sent by SPG must first be loaded onto the NetFPGA as a `pcap` file before they can be transmitted. This two-stage process means that SPG can only replay relatively short traces that have been previously captured[2]. Although it can optionally replay the same short trace multiple times to generate many packets, it can not continually be instructed to send packets by a software packet generator or network emulator using a series of departure times that are not known a priori. PTG, on the other hand, can be used to improve the precision of packet transmissions sent by any existing packet generation software.

## 3.2 Accuracy of Software Network Emulators

The goal of network emulators is to allow arbitrary networks to be emulated inside a single machine or using a small number of machines. Each packet's departure time is calculated based on the packet's path through the emulated network topology and on interactions with other packets. The result of this process is an ordered list of packets and corresponding departure times. How close the actual transmission times are to these ideal departure times is critical for the precision of the network emulator.

Existing software network emulators have been built on Linux and FreeBSD [10, 13, 14]. To minimize overhead, all three process packets in the kernel and use a timer or interrupt firing at a fixed interval to schedule packet transmissions. They effectively divide time into fixed-size buckets, and all packets scheduled to depart in a particular bucket are collected and sent at the same time. Clearly, the bucket size controls the scheduling granularity; i.e., packets in the same bucket will essentially be sent back-to-back.

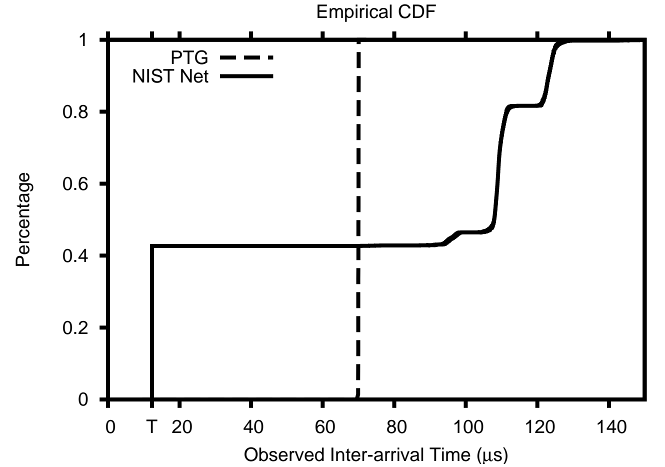


**Figure 1: Effect of NIST Net adding delay to packets sent 70 $\mu$s apart. $T = 12.304\mu s$ is the time it takes to transmit a single 1518-byte packet at 1 Gb/s: packets with that inter-arrival are effectively received back-to-back.**

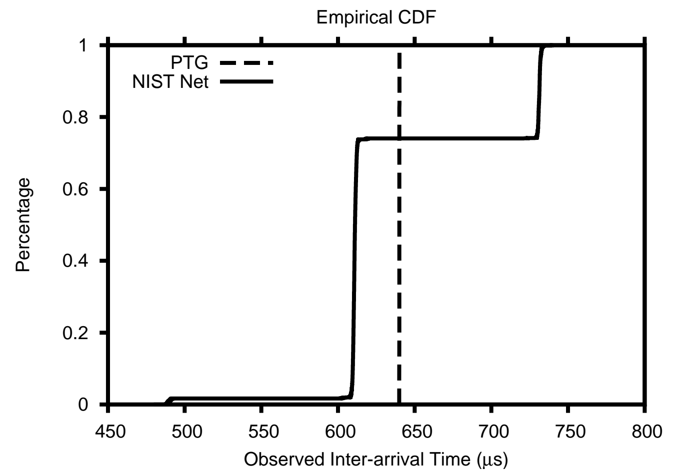


**Figure 2: Effect of NIST Net adding delay to packets sent 640 $\mu$s apart.**

To quantify the scheduling granularity problem, we focus on the transmission times generated by NIST Net [13], a representative network emulator. Here, we generate UDP packets with 1472 byte payloads at a fixed arrival rate using the PTG. The packets are received by a server run-

[2]The largest memory on the board is 64MB which is only about 0.5 seconds of traffic at the 1 Gb/s line rate.

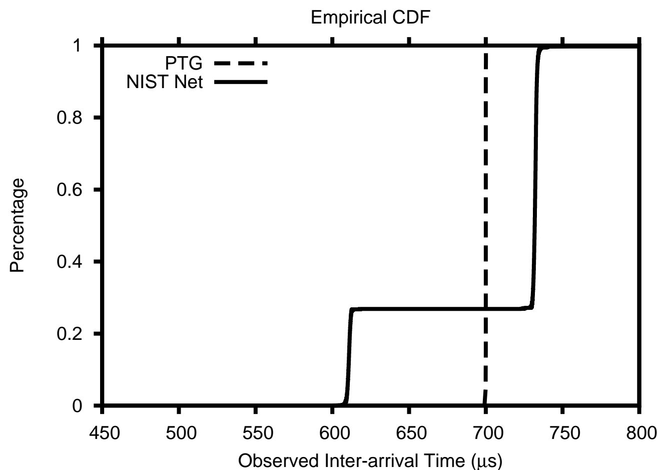


**Figure 3: Effect of NIST Net adding delay to packets sent 700 $\mu$s apart.**

ning NIST Net, pass through the emulated network, and are routed to a third server which measures the resulting packet inter-arrival times. NIST Net is configured to add 100ms of delay to each packet. Although adding a delay to every packet is a simple application of a network emulator, by varying the input packet inter-arrival times, NIST Net's scheduler inaccuracy is clearly visible.

Figure 1 is a CDF of the measured intervals between packet arrivals in NIST Net's input and output traffic. To measure the arrival times of the input traffic we temporarily connect the generating server directly to the measuring server. Here a packet is sent by the PTG to NIST Net, and thus should depart from NIST Net, every 70 $\mu$s. This interval is smaller than the fixed timer interval used by NIST Net, which has a period of 122 $\mu$s [13], so NIST Net will either send the packet immediately or in the next timer interval. Consequently, in Figure 1, 40% of the packets are received back-to-back if we consider that it takes just over 12 $\mu$s to transmit a packet of the given size on the wire (the transmission time of a single packet is marked with a "T" on the x-axis). Very few packets actually depart close to the correct 70 $\mu$s interval between them. Most of the remaining intervals are between 100 $\mu$s and 140 $\mu$s.

Even when the interval between arriving packets is larger than NIST Net's bucket size, the actual packet transmission times are still incorrect. Figures 2 and 3 show the measured arrival intervals for 640 $\mu$s and 700 $\mu$s arrivals, respectively. Note that in both figures, most of the intervals are actually either 610 $\mu$s or 732 $\mu$s, which are multiples of NIST Net's 122 $\mu$s bucket size. It is only possible for NIST Net to send packets either back-to-back or with intervals that are multiples of 122 $\mu$s. When we vary the inter-arrival time of the input traffic between 610 $\mu$s and 732 $\mu$s, it only varies the proportion of the output intervals that are either 610 $\mu$s or 732 $\mu$s.

The cause of the observed inaccuracies is not specific to NIST Net's implementation of a network emulator. Any software that uses a fixed-size time interval to schedule packet transmissions will suffer similar failures at small time scales, and the generated traffic will not be suitable for experiments that are sensitive to the exact inter-arrival times of packets. The exact numbers will differ, depending on the length of the fixed interval. To our knowledge, Modelnet [10] is the software network emulator providing the finest scheduling granularity of 100 $\mu$s with a 10KHz timer. Although higher resolution timers exist in Linux that can schedule a single packet transmission relatively accurately, the combined interrupt and CPU load of setting timers for every packet transmission would overload the system. Therefore, our conclusion is that an all-software network emulator executing on a general-purpose operating system requires additional hardware support (such as the one we propose) to produce realistic traffic at very small time scales.

## 3.3 Variable Packet Inter-arrival Times

Another advantage of PTG is its ability to generate packets with an arbitrary sequence of inter-arrival times and sizes. For example, Figure 4 shows the CDFs of both the requested and the measured transmission times for an experiment with 4000 packets with inter-arrival times following a Pareto distribution. Interestingly, only a single curve is visible in the figure since the two curves match entirely (for clarity we add crosses to the figure at intervals along the input distribution's curve). This property of PTG is exactly the component that the network emulators mentioned in Section 3.2 need. It can take a list of packets and transmission times and send the packets when requested. The crucial difference between PTG and SPG is that SPG has a separate load phase and could not be used by the network emulators.

As another example, Figure 5 shows the CDFs of the requested and the measured transmission times when the requested inter-arrival of packets follows the spike bump pattern probability density function observed in the study on packet inter-arrival times in the Internet by Katabi *et al.* [7]. Here 10000 packets are sent with packet sizes chosen from a simple distribution: 50% are 1518 bytes, 10% are 612 bytes, and 40% are 64 bytes. Note that, again, PTG generates the traffic exactly as expected and hence only one curve is visible.

# 4. DISCUSSION

In this section, we describe the limitations of PTG's current implementation. As PTG is intended to be integrated into existing traffic generators and network emulators, we also briefly describe a prototype we are developing that allows packets from the popular network simulator `ns-2` [9] to be sent on a real network using PTG.

**Limitations:** The limitations of the PTG stem from copying packets between the host computer and the NetFPGA over the 32 bit, 33 MHz PCI bus, which has a bandwidth of approximately 1Gb/s. As explained in Section 2, the payloads of packets sent by the PTG are usually all zeros, which requires sending only the packet headers over the PCI bus. This is sufficient for network experiments that do not involve packet payloads. A larger body of experiments ignore most of the packet payloads except for a minimal amount of application-layer signaling between sender and receiver. To support this, arbitrary custom data can be added to the start of any packet payload. This additional data is copied

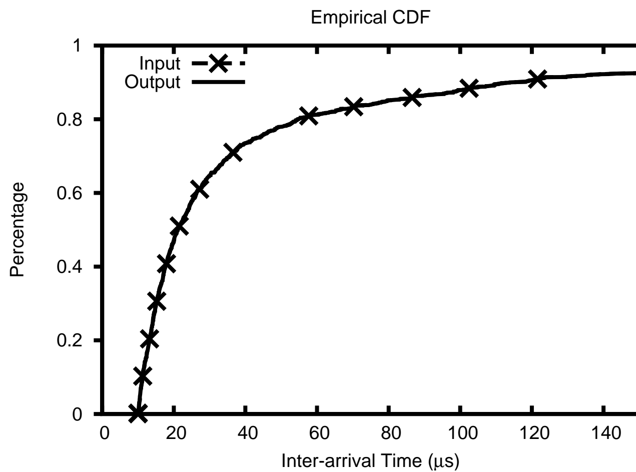


**Figure 4: CDF of measured inter-arrival times compared with an input pareto distribution.**

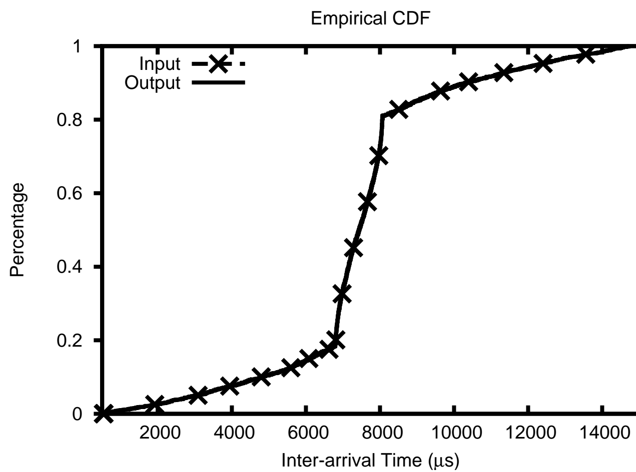


**Figure 5: CDF of measured inter-arrival times compared with an input distribution.**

to the NetFPGA card and is included in the packet. In the future, we plan to allow a number of predefined packet payloads to be copied to the NetFPGA in a preprocessing phase. These payloads could then be attached to outgoing packets without the need to repeatedly copy them over the PCI bus. We envision this feature would support many experiments where multiple flows send packets with the same or similar payloads.

The current PTG software implementation does not yet handle received packets from the network. For experiments with a high traffic volume, it would not be possible to transfer all of the received packet payloads from the 4 Gigabit Ethernet ports of the NetFPGA to the host computer over the PCI bus. Only a fraction of the packets could be transfered or the packets could be summarized by the NetFPGA.

**Integration with `ns-2`:** Because many researchers are already familiar with `ns-2`, this is a useful tool to test real network devices together with simulated networks. Compared to previous attempts to connect `ns-2` to a real network [15], the integration of PTG with `ns-2` will enable generating real packets with transmission times that match the `ns-2` simulated times even on very small time scales. For example, a particular link in `ns-2`'s simulated network could be mapped to a link on a physical network and when simulated packets would arrive at this link, they would be given to the PTG to be transmitted based on the requested simulated time.

# 5. CONCLUSION

Generating realistic traffic in network testbeds is challenging yet crucial for performing valid experiments. Software network emulators schedule packet transmission times in software, hence incurring unavoidable inaccuracy for inter-transmission intervals in the sub-millisecond range. Thus, they are insufficient for experiments sensitive to the inter-arrival times of packets. In this paper we present NetFPGA-based Precise Traffic Generator (PTG) built on top of the NetThreads platform. NetThreads allows network devices to be quickly developed for the NetFPGA card in software while still taking advantage of the hardware's low-level timing guarantees. The PTG allows packets generated on the host computer to be sent with extremely accurate inter-transmission times and it is designed to integrate with existing software traffic generators and network emulators. A network emulator that uses PTG to transmit packets can generate traffic that is realistic at all time scales, allowing researchers to perform experiments that were previously infeasible.

## Acknowledgments

This work was supported by NSERC Discovery, NSERC RTI as well as a grant from Cisco Systems.

# 6. REFERENCES

[1] N. Beheshti, Y. Ganjali, M. Ghobadi, N. McKeown, and G. Salmon, "Experimental study of router buffer sizing," in *IMC'08: Proceedings of the 8th ACM SIGCOMM conference on Internet measurement*, 2008.

[2] K. V. Vishwanath and A. Vahdat, "Realistic and responsive network traffic generation," in *SIGCOMM'06: Proceedings of the 2006 conference on Applications, technologies, architectures, and protocols for computer communications*, 2006.

[3] N. Beheshti, Y. Ganjali, M. Ghobadi, N. McKeown, J. Naous, and G. Salmon, "Performing time-sensitive network experiments," in *ANCS'08: Proceedings of the 4th ACM/IEEE Symposium on Architectures for Networking and Communications Systems*, 2008.

[4] R. Prasad, C. Dovrolis, and M. Thottan., "Evaluation of Avalanche traffic generator," 2007.

[5] A. Rupp, H. Dreger, A. Feldmann, and R. Sommer, "Packet trace manipulation framework for test labs," in *IMC'04: Proceedings of the 4th ACM SIGCOMM conference on Internet measurement*, 2004.

[6] G. A. Covington, G. Gibb, J. Lockwood, and N. McKeown, "A packet generator on the NetFPGA platform," in *FCCM'09: Proceedings of the 17th annual IEEE symposium on field-programmable custom computing machines*, 2009.

[7] D. Katabi and C. Blake, "Inferring congestion sharing and path characteristics from packet interarrival times," Tech. Rep., 2001.

[8] M. Labrecque, J. G. Steffan, G. Salmon, M. Ghobadi, and Y. Ganjali, "NetThreads: Programming NetFPGA with threaded software," in *NetFPGA Developers Workshop'09, submitted*.

[9] The Network Simulator - ns-2. http://www.isi.edu/nsnam/ns/.

[10] A. Vahdat, K. Yocum, K. Walsh, P. Mahadevan, D. Kostić, J. Chase, and D. Becker, "Scalability and accuracy in a large-scale network emulator," *SIGOPS Operating Systems Review archive*, vol. 36, no. SI, pp. 271–284, 2002.

[11] M. Labrecque, P. Yiannacouras, and J. G. Steffan, "Scaling soft processor systems," in *FCCM'08: Proceedings of the 16th annual IEEE symposium on field-programmable custom computing machines*, April 2008.

[12] J. Sommers and P. Barford, "Self-configuring network traffic generation," in *IMC'04: Proceedings of the 4th ACM SIGCOMM conference on Internet measurement*, 2004.

[13] M. Carson and D. Santay, "NIST Net: a Linux-based network emulation tool," *SIGCOMM Computer Communication Review*, vol. 33, no. 3, pp. 111–126, 2003.

[14] Dummynet. http://info.iet.unipi.it/ luigi/ip_dummynet/.

[15] "Network emulation in the Vint/NS simulator," in *ISCC'99: Proceedings of the The 4th IEEE Symposium on Computers and Communications*, 1999, p. 244.