Caliper: Precise and Responsive Traffic Generation using NetThreads*

Monia Ghobadi*, Geoffrey Salmon*, Martin Labrecque†, Yashar Ganjali*, J. Gregory Steffan†
*Department of Computer Science, †Department of Electrical and Computer Engineering
University of Toronto
{monia, geoff, yganjali}@cs.toronto.edu, {martinl, steffan}@eecg.toronto.edu

ABSTRACT

This paper presents (i) Caliper, a highly-accurate packet injection tool, and (ii) NetThreads, a new platform that dramatically simplifies the development of low-level network applications on the NetFPGA board. NetThreads provides a familiar environment to software developers where multithreaded `C` programs can be compiled and run on the NetFPGA. On top of NetThreads, we have built Caliper, a precise and responsive traffic generator that takes packets generated on a host computer and transmits them onto a gigabit Ethernet network with precise inter-transmission times. For packet injection, existing software traffic generators rely on generic Network Interface Cards which, as we demonstrate, do not provide high-precision timing guarantees. Hence performing valid and convincing experiments becomes difficult or impossible in the context of time-sensitive network experiments. Our evaluation shows that Caliper is able to reproduce packet inter-transmission times from a given arbitrary distribution while capturing the closed-loop feedback of TCP sources. Specifically, we demonstrate that Caliper provides three orders of magnitude better precision compared to commodity NIC: with requested traffic rates up to the line rate, Caliper incurs an error of 8 ns or less in packet transmission times. Furthermore, we explore Caliper's ability to integrate with existing network simulators to project simulated traffic characteristics into a real network environment. Caliper and NetThreads are both freely available online [1].

Categories and Subject Descriptors

C.2.1 [**Computer-Communication Networks**]: Network Architecture and Design

General Terms

Design, Experimentation, Measurement

Keywords

NetFPGA, Traffic Generation, Soft Processors

*A preliminary version of this work appeared in the NetFPGA Developers Workshop 2009.

1. INTRODUCTION

There are many challenges associated with performing valid experiments in network testbeds. Generating realistic and responsive traffic that reflects different network conditions and topologies is one of such key challenges. To perform network experiments, researchers often use a collection of commodity Linux machines as traffic generators. However, creating a large number of connections in order to accurately model the traffic shape in networks with thousands of flows is difficult for several reasons. First, it is not always possible to use real network traces since they do not maintain the feedback loop behavior between the network and traffic sources (for example the TCP closed-loop congestion feedback). Second, the complexity of traffic generation increases when trying to capture the heterogeneity of link capacities using only a limited number of physical machines. Finally, commodity hardware does not guarantee the precision of generated traffic, which is bounded by the system timer resolution.[1] In addition, there are differences in implementation and default settings (and sometimes lack of a way to change those settings) in commodity hardware. Hence, it is intrinsically difficult to perform *time-sensitive* network experiments with confidence on the accuracy of packet injection times. Time-sensitive experiments are those that need very high-precision timings for packet injections into the network. Experimenting with new congestion control algorithms, buffer sizing in Internet routers [2], and denial of service attacks which use low-rate packet injections [3] are all examples of time-sensitive experiments, where a subtle variation in packet injection times can change the results significantly [4].

As an alternative, commercial traffic generators are useful for some experiments, but they have their own drawbacks. They are usually very expensive and their proprietary nature makes them inflexible for research purposes. As an example, Prasad *et al.* [5] describe differences observed between a TCP Reno packet sequence

[1] A Linux kernel is typically capable of providing resolutions of 1 ms. In comparison, a packet of 1500 bytes on a 1 $Gbps$ link has a transmission time of less than 12 μs.

generated by a commercial traffic generator and the expected behavior of the standard TCP Reno protocol.

A more precise solution is using hardware based packet generators such as the traffic generator by Covington *et al.* [6] (hereafter referred to as the Stanford Packet Generator or SPG). SPG is based on *NetFPGA* [7, 8], a PCI-based programmable board containing an FPGA, four gigabit Ethernet ports, and memory. The SPG system generates more precise traffic by accurately 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 precise than many prior approaches, it has several limitations. The closed-loop feedback for TCP sources (and any other protocol that depends on the feedback from the system) is not kept because the trace files are based on past measurements. Furthermore, replaying a prerecorded trace on a link with different properties (such as capacity and buffer size) does not necessarily result in realistic traffic without performing non-trivial adjustments. 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 present the design, implementation, and evaluation of Caliper, a precise and responsive traffic generator based on the NetFPGA platform with highly-accurate packet injection times that can be easily integrated with various software-based traffic generation tools. Caliper has the same accuracy level as SPG, but provides a key additional feature that makes it useful in a larger range of network experiments: Caliper injects dynamically created packets, and thus, it can react to feedback and model the closed-loop behavior of TCP and other protocols. The ability to produce live traffic makes Caliper useful to explore a variety of what-if scenarios by tuning user, application, and network parameters. Note that characterizing real-traffic is not the goal of this work, instead, our objective is packet injection accuracy.

Caliper is built on *NetThreads*, a platform we have created 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 since it offers direct network I/O and allows the programmer to specify accurate timing requirements.

Our evaluation demonstrates that Caliper is able to reproduce packet inter-arrival times from a given arbitrary distribution with high accuracy while capturing the closed-loop feedback of TCP sources. We present the accuracy of Caliper with various packet arrival rates and demonstrate that with requested traffic rates up to the line rate, the maximum error that Caliper incurs is 8 *ns* which is the resolution of the measuring system clock. We also present measurements of an existing network emulator which clearly demonstrates the need that Caliper fulfills. We further demonstrate integration of Caliper with existing software-based traffic generators as well as the `ns-2` network simulator.

In summary, the contributions of this paper are two-fold: (i) we present and evaluate Caliper, a tool that can precisely control the inter-transmission times of packets created in the host computer; (ii) we introduce NetThreads, a new platform that dramatically simplifies the development of low-level network applications on the NetFPGA.

2. NETTHREADS

In this section, we describe NetThreads [9], the programmable platform that enables Caliper and leverages an FPGA-based network card. FPGAs are increasingly used in packet processing systems [10–12] due to several advantages that they provide: (i) ease of design and fast time-to-market compared to building custom chips; (ii) the ability to connect to a number of memory channels and network interfaces, possibly of varying technologies; (iii) the ability to fully exploit parallelism and custom accelerators; and (iv) the ability to field-upgrade the hardware design.

To avoid implementing a packet generator in low-level hardware-description language (how FPGAs are normally programmed), we instead implement *soft processors* – processors composed of programmable logic on the FPGA. Despite the raw performance drawbacks, a soft processor has several advantages: it is easier to program (e.g., using `C`), portable to different FPGAs, flexible (i.e., can be customized), and can be used to communicate with other components/accelerators in the design. In this work, our FPGA resides on a NetFPGA board and communicates through DMA on a PCI interface to a host computer. This configuration is particularly well suited for packet generation: (i) the load of the soft processors is isolated from the load on the host processor, (ii) the soft processors suffer no operating system overheads, (iii) they can receive and process packets in parallel, and (iv) they have access to a high-resolution system timer (much higher than that of the host timer).

The rest of this section describes our infrastructure, including the system architecture and our base platform, and how we do compilation, timing and validation.

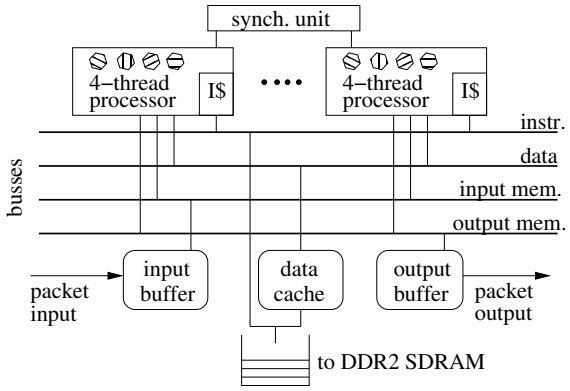

Figure 1: The architecture of our soft multithreaded multiprocessor. We implement two processors, although the architecture can be scaled to larger numbers of processors.

System architecture: As shown in Figure 1, our hardware design is sufficiently general to accommodate a large variety of packet processing workloads. The memory system is composed of a private instruction cache for each processor, and three data memories that are shared by all processors; this design is sensitive to the two-port limitation of block RAMs available on FPGAs. The first memory is an input buffer that receives packets on one port and services processor requests on the other port via a 32-bit bus, arbitrated across processors. The second is an output memory buffer that sends packets to the NetFPGA Ethernet controllers on one port, and is connected to the processors via a second 32-bit arbitrated bus on the second port. Both input and output memories are 16KB, allow single-cycle random access and are controlled through memory-mapped registers; the input memory is read-only and is logically divided into ten fixed-sized packet slots. The third memory is a shared memory managed as a cache, connected to the processors via a third arbitrated 32-bit bus on one port, and to a DDR2 SDRAM controller on the other port. For simplicity, the shared cache performs 32-bit line-sized data transfers with the DDR2 SDRAM controller (similar to previous work [13]), which is clocked at 200MHz. The SDRAM controller services a merged load/store queue of 16 entries in-order; since this queue is shared by all processors it serves as a single point of serialization and memory consistency, hence threads need only block on pending loads but not stores. Our base soft processor is a single-issue, in-order, 5-stage, 4-way multithreaded processor [14] and each processor has a dedicated connection to a synchronization unit that implements 16 mutexes and to control registers that manage packet I/O. Those control registers provide the option of sending a packet by default as soon as possible, or at a given future time to avoid random latencies introduced by bus contention and cache misses when executing software on the processors. Non-blocking software commands manage the hardware that fills the input buffer with received packets and sends packets from the output buffer. In the NetThreads `C` programming model, the input and output memories, system clock, mutexes and control registers to send and receive packets are all memory-mapped.

Platform: Our processor designs are inserted inside the NetFPGA 2.1 `Verilog` infrastructure [15] that manages four 1GigE Media Access Controllers (MACs). We added to this base framework a memory controller configured through the Xilinx Memory Interface Generator to access the 64 Mbytes of on-board DDR2 SDRAM. The system is synthesized, mapped, placed, and routed under high effort to meet timing constraints by Xilinx ISE 10.1.03 and targets a Virtex II Pro 50 (speed grade 7 ns).

Compilation: Our compiler infrastructure is based on modified versions of `gcc` 4.0.2, `Binutils` 2.16, and `Newlib` 1.14.0 (a C library for embedded systems) that target variations of the 32-bit MIPS-I ISA. This toolchain allows to compile ordinary `C` code and NetThreads provides an API for memory-mapped operations [9]. Integer division and multiplication are both implemented in software. Because network experiments often need to be performed with a specific protocol stack (e.g. a Linux kernel 2.6 TCP/IP stack), Caliper's program on NetThreads is protocol-agnostic and only controls the timing of packets as generated from the host computer (see Section 3.1.1).

Timing: Our processors run at the clock frequency of the Ethernet MACs (125MHz) because there are no free PLLs (a.k.a. Xilinx DCMs) after merging-in the NetFPGA support components. Due to these stringent timing requirements, and despite some available area on the FPGA, (i) the private instruction caches and the shared data write-back cache are both limited to a maximum of 16KB, and (ii) we are also limited to a maximum of two processors. These limitations are not inherent in our architecture, and would be relaxed in a system with more PLLs and a more modern FPGA.

Validation: At runtime in debug mode and in RTL simulation (using `Modelsim` 6.3c) the processors generate an execution trace that has been validated for correctness against the corresponding execution by a simulator built on MINT [16].

3. PRECISE TRAFFIC GENERATION

Caliper'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. The generated packets are sent out of a single Ethernet port of the NetFPGA, according to any given sequence of requested inter-

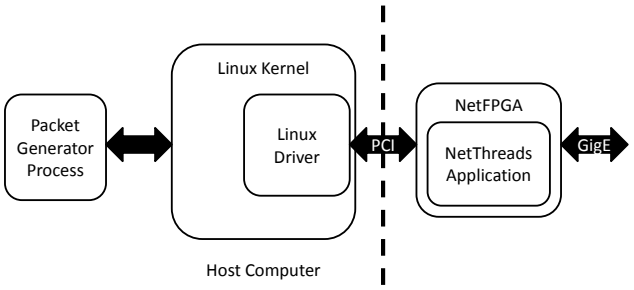

Figure 2: Components of Caliper packet generator.

transmission times. It is important to note that because packets are streamed, the generator can immediately change the traffic in response to feedback. Unlike previous works that replay packets with prerecorded transmission times from a trace file, Caliper generates live packets and supports closed-loop traffic. Therefore, Caliper can easily be coupled with existing traffic generators (such as Iperf [17] and Netperf [18]) to improve their accuracy at small time scales.

3.1 Caliper's Components

We have built Caliper on the NetThreads platform. Caliper's components are illustrated in Figure 2 showing the life cycle of a packet through the system, from creation to transmission. First, a user space process or a kernel module on the host computer determines when a packet should be sent. 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 and copied to the NetFPGA card. Combining descriptions reduces the number of separate transfers required and is necessary for sending packets at the line rate of 1 *Gbps*. From there, packet descriptions are processed in software on the NetThreads soft multithreaded processors. Each software thread assembles packets in the NetThreads' output memory (shown in Figure 1). Next, a selected 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 Caliper in detail. We also describe the underlying limitations and challenges that influenced our design. Note that users can edit all parts of Caliper to modify and extend its functionality [1].

3.1.1 Packet Creation to Driver

The reasons and context of packet creation are application-specific. To produce realistic traffic, we envision that a network simulator, such as Swing [19], will decide when to send each packet. This simulation may be running in either a user space process, like `ns-2` [20], or a Linux kernel module, as in ModelNet [21]. 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 user space or the kernel. In Section 4 we describe examples of using Caliper with the `ns-2` network simulator, Iperf traffic generator, as well as our own user space program.

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 zeroed when the packet is eventually transmitted. In Section 5, we mention a work-around that may be useful when the contents of packet payloads are important to an experiment.

3.1.2 Driver to NetThreads

We modified the driver provided with NetFPGA to support Caliper. 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 33-MHz 32-bit bus with a top theoretical transfer rate of 1056 *Mbps*, 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 260 *Mbps* 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 *Gbps*. Instead we work around this problem by sending only the packet headers across the PCI bus and rebuilding the packets inside the NetFPGA. Currently, NetFPGA fills the payload with zeros, which is sufficient both for our evaluation and for many of the tests we are interested in performing with Caliper.

To obtain the line rate 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 (shown in Figure 1). 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.

3.1.3 NetThreads to Wire

This last part of Caliper 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. Then, Caliper prepares these packets for transmission and sends them at the appropriate times.

NetThreads takes advantage of multithreaded processors, which have been shown to outperform single threaded processor on parallel tasks [14,22]. To achieve high throughput in NetThreads, it is therefore important to maximize parallelism and the processor pipeline utilization, meaning that all eight available threads must be put to contribution (each processor is 4-way multithreaded as explained in Section 3.2).

3.2 Leveraging the hardware threads in NetThreads

As explained in Section 2, NetThreads provides 8 hardware threads that implicitely all execute the same program, but with a private stack space. In Caliper, we implement thread-specific behavior by reading each thread's identifier with an API call [9]. We designate one thread to be in charge of sending all the packets. This way, packets are not reordered and we can easily control their transmission time without using synchronization and sorting. Each of the other seven threads continually process jobs from a work queue. There are two types of jobs: 1) interpreting a message from the driver and scheduling further jobs to prepare each outgoing packet, and 2) preparing an outgoing packet and notifying the sending thread when completed.

When preparing outgoing packets, most of the work performed by the threads involves copying the requested packet headers from the input memory to the output memory. As described in Section 2, buses to the input and output memories are arbitrated between both processors; i.e., on a given clock cycle, only one of the processors can access the input memory, and only one can access the output memory. Fortunately, these details are hidden from the software, and the instructions squashed by an arbiter will be retried without impacting the other threads [14]. Because of the fine-grained interleaving of load and store instructions across the two processors, threads from both processors will be able to make forward progress. Additionally, multithreading inside each processor prevents pipeline stalls (e.g. the dependence from a load to a store in a copy operation) by interleaving multiple threads (4 in this case), thus providing an increased pipeline efficiency.

3.3 Integration with Existing Tools

Caliper is intended to be integrated with software in charge of creating packets: in this section, we explain the integration of Caliper with existing Linux-based software traffic generators and in Section 4.5, we describe and evaluate a prototype that we develop to allow packets from the `ns-2` simulator to be sent on a real network using Caliper.

Caliper is able to receive packets directly from the Linux network stack as well as over NetLink sockets. Since Caliper is acting as a network card device in the kernel, it can transmit packets generated within the Linux kernel with any software packet generator (i.e., ping, Iperf, Netperf, etc.) according to user-specified inter-arrival times. By using TCP sources, Caliper can transmit live TCP connections and closed-loop sessions. As a result, the generated traffic becomes "responsive" to changing network conditions or competing application traffic by capturing the congestion feedback of TCP sources and any other Linux implemented protocols. Another desirable advantage of using Caliper for TCP sources is the ability to define an inter-packet gap to provide TCP pacing [23] in hosts. TCP pacing addresses the burstiness of TCP traffic by minimizing the possibility of overflow in router buffers [2]. Towards this goal, Caliper is able to adjust precisely the interval between outgoing packets and produce smoothed and stable traffic. In Sections 4.1 and 4.3, we demonstrate that Caliper provides three orders of magnitude better precision compared to commodity NIC when using Iperf to generate TCP and UDP traffic.

In order to preserve the closed-loop sessions of TCP based traffic generators, Caliper needs to receive packets from the wire, process them in the NetFPGA, and copy them to the host computer over the PCI bus. Because the PCI bus is highly contented (§3.1.2), the receiving operations can interfere with the scheduled transmissions of packets. Since the main focus of Caliper is providing precision in transmission times, in the current implementation of Caliper, we eliminate this potential interference by using a separate physical PCI Express NIC to receive the incoming acknowledgment packets from the wire and reserving the NetFPGA for transmitting packets. Caliper kernel module routes packets appropriately, and as a result, the process controlling Caliper is unaware that it is receiving packets from one network interface and sending packets out of a different one.[2]

4. EVALUATION

In this section we evaluate the performance of Caliper by focusing on its accuracy and flexibility features. We

[2]The path that the sent and received packets follow in the network does not have to be completely disjoint. Only the closest hop to Caliper needs to be duplicated.

Figure 3: Topology of experiments.

set-up our experiments to reflect Caliper's intended use: to complement existing traffic generators by allowing them to precisely control when packets are transmitted. Hence, we present our experiments where the most important metric is the accuracy of packet transmission times. We also present measurements of an existing network emulator which clearly demonstrate the need that Caliper fulfills. We further present a prototype that integrates Caliper with the `ns-2` simulator.

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 card. The topology of our experiments is illustrated in Figure 3. In each test, there is a single server sending packets and a single server receiving packets via a NetFPGA-based router in the middle that measures the packets inter-arrival times. In the experiment described in Section 4.4, the router is replaced with a server running a software network emulator which routes packets between the sender and receiver.

Since Caliper's main goal is to transmit packets exactly when requested, the measurement accuracy is vital to the evaluation. Measuring arrival times in software using `tcpdump` or similar applications is imprecise: generic NICs combined with OS overheads are intrinsically inaccurate at the level we operate [4]. Therefore, we use a NetFPGA router to measure packet inter-arrival times at the middle node (router). The NetFPGA router is configured with the "event capturing module" of the NetFPGA router design [2,24] which supports instrumenting the router's output queues:[3] when a packet arrives, departs or is dropped, the system clock time of the NetFPGA, which has an 8 ns granularity, is recorded. To reduce overhead, the NetFPGA router batches multiple events in packets that are periodically sent out and we obtain the packet inter-arrival times at the router by subtracting successive timestamps in the payload of those event packets.

[3]To increase the accuracy of the timestamps even more, we removed two parts of the router pipeline that could add a variable delay to packets before they reach the output queues. This is possible because we are only interested in measuring packets that arrive at a particular port and the routing logic is unnecessary.

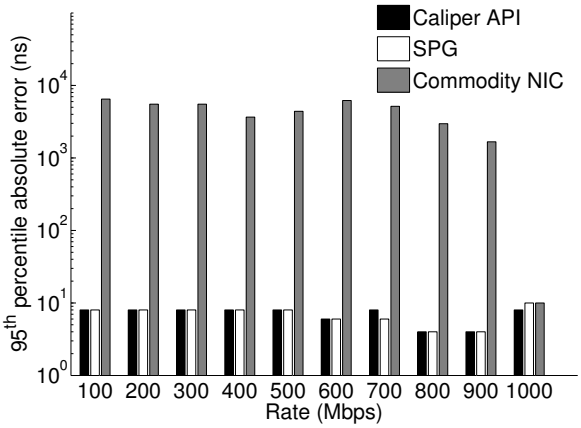

Figure 4: Comparing the 95^{th} percentile of absolute error ($|D_R - D_M|$) between Caliper, SPG, and commodity NIC when injecting UDP packets.

4.1 Sending UDP Packets at Fixed Intervals

The simplest test case for Caliper is to generate UDP packets with a fixed inter-transmission time. Comparing the requested inter-transmission time with the observed inter-arrival times demonstrates Caliper's degree of precision. As explained in Sections 3.2 and 3, Caliper leverages software running on what has previously been a hardware-only network device, the NetFPGA. Even executing software, NetThreads should provide sufficient performance and control for precise packet generation.

To evaluate the above criteria we compare Caliper's transmission times against those of Stanford's Packet Generator (SPG), which is implemented on the NetFPGA solely in hardware. Moreover, we demonstrate the lack of precision when using a commodity NIC transmitting Iperf traffic. Figure 4 shows the 95^{th} percentile of absolute error between the measured inter-arrival times (D_M) and the requested inter-transmission times (D_R) corresponding to various packet transmission rates (T_R). It is important to note that the 95^{th} percentile error is a more conservative metric than the average error as it captures the 5% largest errors. For each transmission rate, we send 1,500,000 UDP packets of size 1518 bytes (including Ethernet headers) using Caliper, SPG, or an Intel commodity Ethernet NIC. To generate constant bit rate traffic over the commodity NIC we use the Iperf traffic generator with rate T_R. We then capture a portion of traffic in a trace file and replay it with SPG while configuring SPG with the exact same packet inter-arrival time that we used with Caliper, D_R.

As Figure 4 illustrates, for all range of transmission times up to 1 *Gbps*, the 95^{th} percentile absolute error

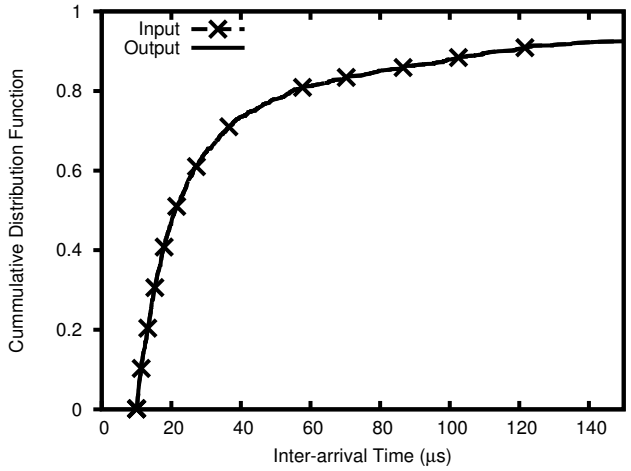

Figure 5: CDF of measured inter-arrival times compared with an input Pareto distribution. Only a single curve is visible since the two plots match entirely.

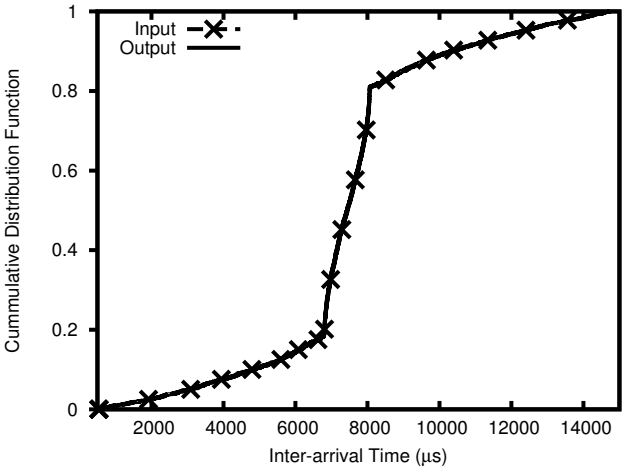

Figure 6: CDF of measured inter-arrival times when both packets sizes and requested inter-arrival times vary. Only a single curve is visible since the two plots match entirely.

is around 8 *ns* for both Caliper and STG. The clock period of the sending and measuring NetFPGA systems is 8 *ns*, and hence an error of 8 *ns* implies that most of the inter-transmission times are within one clock cycle (the measurement resolution). This shows that even though NetThreads is executing software, it still allows precise control on when packets are transmitted. On the other hand, note that the commodity NIC's error is almost three orders of magnitude higher than both Caliper and STG. At 1 *Gbps* rate, we notice that the error is minimum for Caliper, STG as well as the commodity NIC case because the network is operating at its maximum utilization and packets are sent and received back-to-back.

Although both Caliper and STG packet generators are of similar accuracy, SPG has a limitation that makes it unsuitable for the role we intend for Caliper. 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.[4] Although SPG can optionally replay the same short trace multiple times, it can not dynamically 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. Caliper, on the other hand, can be used to improve the precision of packet transmissions streamed by any existing packet generation software.

[4]The largest memory on the board is 64 *MB* which is only about 0.5 seconds of traffic at the 1 *Gbps* line rate.

4.2 Variable Inter-arrival Times and Packet Sizes

Another advantage of Caliper is its ability to generate packets with an arbitrary sequence of inter-arrival times and sizes that are both essential parts of performing realistic large scale experiments. Figure 5 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). As we will demonstrate in Section 4.4, this property of Caliper is exactly the component that the network emulators need. Caliper can take a list of packets and transmission times and send the packets when requested. The crucial difference between Caliper and SPG is that SPG has a separate load phase that prevents it from being used by network emulators.

As another example, Figure 6 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.* [25]. In this distribution, a flow traverses a low bandwidth bottleneck with an inter-arrival of 8 *ms* followed by a high bandwidth bottleneck. Moreover, to demonstrate Caliper's ability in generating packets with variable sizes, we choose the packet sizes according to another realistic distribution from the same study: 50% are 1518 bytes, 10% are 612 bytes, and 40% are 64 bytes. Note that, again, Caliper generates the traffic exactly

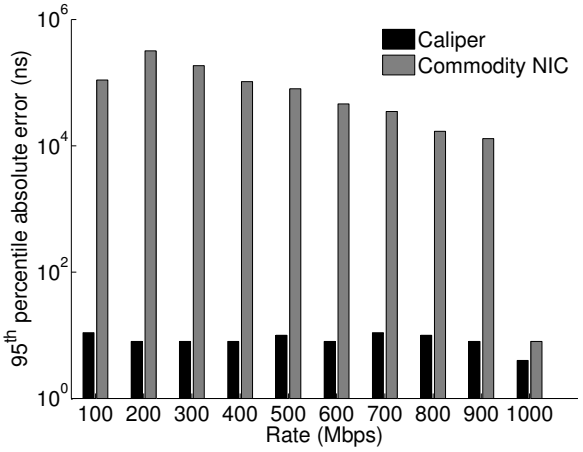

Figure 7: Comparing the 95^{th} percentile of absolute error ($|D_R - D_M|$) between Caliper and a commodity NIC when injecting TCP packets.

as expected and hence only one curve is visible.

4.3 Generating Responsive Traffic

As explained in Section 4.5, Caliper has the ability to receive packets from the Linux network stack and hence it can be used to produce live TCP connections and closed-loop sessions. In this section, we evaluate the performance of Caliper to inject smoothed TCP packets (paced TCP) at precise time intervals and compare the precision of using TCP Iperf traffic with Caliper and a commodity NIC.[5] As in [2], we use the Precise Software Pacer (PSPacer) [26] package as a loadable kernel module to enforce pacing while using the commodity NIC. The challenges to accomplish precise packet pacing are discussed in [24]. PSPacer 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 controls the timing of packets in software without using timers. The trade-off is that packets are being sent at the line rate through a regular NIC even when the data rate has been limited by pacing. In both experiments, we use an unmodified version of TCP, as implemented by the Linux network stack.

Similar to experiments in Section 4.1, we calculate the absolute error ($|D_R - D_M|$) between the measured inter-arrival times (D_M) and the requested inter-transmission times (D_R) corresponding to the requested packet transmission rate (T_R) of Iperf. As illustrated in Figure 7, Caliper improves the 95^{th} percentile of absolute error by almost three orders of magnitude compared to the commodity NIC. Hence, Caliper's accuracy enables researchers to perform live and time-sensitive network experiments with confidence on the accuracy of packet injection times. As in Figure 4, at 1 *Gbps* rate, the error of both Caliper and the commodity NIC is minimal because packets are sent and received back-to-back.

[5]Note that in this set of experiments we are unable to include SPG due to its open-loop limitation.

4.4 Accuracy of Existing 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 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 [21, 27]. To minimize overhead, they 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.

To quantify the scheduling granularity problem, we focus on the transmission times generated by NIST Net [27], a representative network emulator. Here, we generate MTU-sized UDP packets at a fixed arrival rate using Caliper. The packets are received by a server running 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 100 *ms* 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 8 is a CDF of the measured intervals between packet arrivals in NIST Net's input and output traffic where the requested packet inter-transmission time is 70 μs. As shown, Caliper accurately delivers packets to NIST Net in the intermediate sever, but NIST Net is unable to preserve the precision. Here a packet is sent by Caliper to NIST Net, and thus should depart from NIST Net, every 70 μs. This interval is smaller than the fixed timer interval used by NIST Net, which has a period of 122 μs [27], thus NIST Net will either send the packet immediately or in the next timer interval. Consequently, in Figure 8, 40% of the packets are received back-to-back if we consider that it takes just

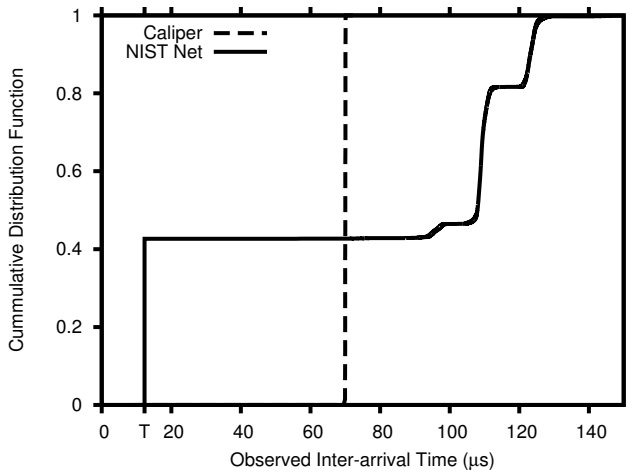

Figure 8: Effect of NIST Net adding delay to packets sent 70 μs apart. $T = 12.304\mu s$ is the time it takes to transmit a single 1472-byte packet at 1 *Gbps*.

over 12 μs 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 μs interval between them. Most of the remaining intervals are between 100 μs and 140 μs. Note that the server running NIST Net is using a commodity NIC which also plays a negative role in preserving the packet inter-transmission times.

Even when the interval between arriving packets is larger than NIST Net's bucket size, the actual packet transmission times are still incorrect. We repeated the same experiment with inter-transmission times of 640 μs and 700 μs arrivals and observed that in both cases, 70% of the intervals are actually either 610 μs or 732 μs, which are multiples of NIST Net's 122 μs bucket size. It is only possible for NIST Net to send packets either back-to-back or with intervals that are multiples of 122 μs. When we vary the inter-arrival time of the input traffic between 610 μs and 732 μs, it only varies the proportion of the output intervals that are either 610 μs or 732 μs.

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 [21] is the software network emulator providing the finest scheduling granularity of 100 μs 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.

4.5 Network Emulation by Integrating Caliper with `ns-2`

Simulation and testbed construction represent the two most important methodologies available to network researchers for the design and evaluation of both novel and existing networking elements. Employing an emulation capability in network simulation provides the ability for real-world traffic to interact with a simulation. Since many researchers are already familiar with the Network Simulator `ns-2`, this is a useful tool to test real network devices together with simulated networks. Such integrations will enable researchers to repeat simulation experiments under different link and environment conditions.

Compared to previous attempts to connect `ns-2` to a real network [28], our integration of Caliper with `ns-2` enables generating real packets with transmission times that match the `ns-2` simulated times even on very small time scales. In our integration, we mark a particular link in `ns-2`'s simulated network to be mapped to physical link(s). When the simulation starts, the simulated packets are built and traverse the simulated links and nodes until they reach the specific marked queue (*caliper_queue*) connecting the simulation and physical worlds. At this point, Caliper builds real packets and transmits them according to their simulated inter-arrival times. The mapping between simulated node IDs and physical IP/MAC addresses and port numbers is also specified in the simulation configuration file. On the other end of the *caliper_queue*, there is another simulation running, which receives the physical packet, and fires the appropriate simulation event indicating that the corresponding simulated packet has been received.

Our implementation only requires a few additional commands to a simulation program written in the Tcl language which makes it extremely convenient to work with. As an example, Figure 9 illustrates a simple topology expressed in our extension. Lines 1 to 6 define the topology and the nodes conventionally in `ns-2` language. In line 8, we define the link between node 2 (n_2) and node 3 (n_3) as *caliper_queue* so that when packets depart the link's queue the simulator will send the packet to Caliper's driver. In line 10, we create a mapping table for the physical IP and MAC addresses corresponding to the physical ends of wire between n_2 and n_3. Finally in line 14 we assign a traffic source to


```
1.#Caliper's interval in seconds:
2.set caliper_interval 0.001
3.#define the nodes n0, n1, n2, and n3
4.#define the links (n0, n2), (n2, n3), and (n3, n1)
5.#obtain the queue of the specific caliper queue:
6.set caliper_queue_ [$ns simplex-link-op $n2 $n3 queue]
7.#call use-caliper function:
8.$cliper_queue_ use-caliper
9.#set the physical IP/MAC addresses mapping table:
10.$ns insert_nat IP_N2 IP_N3 PORT_N2 PORT_N3 MAC_N2 MAC_N3
11.#Create a UDP agent and attach it to node n0
12.#Create a CBR traffic source and attach it to udp0
13.#set the rate of the CBR source:
14.$cbr0 set interval_ $caliper_interval
```

Figure 9: Illustration of a simple topology expressed in our integration of `ns-2` with Caliper.

n_0 with the packet inter-arrival time defined in line 2 (1 ms).

Figure 10 compares the CDF of measured physical packets' inter-transmission times with the simulated packet logs from the `ns-2` trace file. The `ns-2` sources are set to send UDP packets with 1 ms inter-transmission times and our integration of Caliper with `ns-2` is able to preserve the inter-arrival times of 80% of the packets. One of the challenges to integrate Caliper with simulation software is synchronizing the simulation time and real-time. In our approach, we make the intuitive assumption that the simulation world is always faster than the real world. In order to keep the difference between real-time and simulation time minimum, we pause the simulation scheduler and as a result, we maintain the same simulation clocks between the simulated world and the real world. We envision that the main cause for the inaccuracy in 20% of the packets is due to our inefficiency in synchronizing real and simulated times.

5. DISCUSSION AND FUTURE WORK

The limitations of Caliper 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 1 *Gbps*. As explained in Section 3, the payloads of packets sent by Caliper 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-

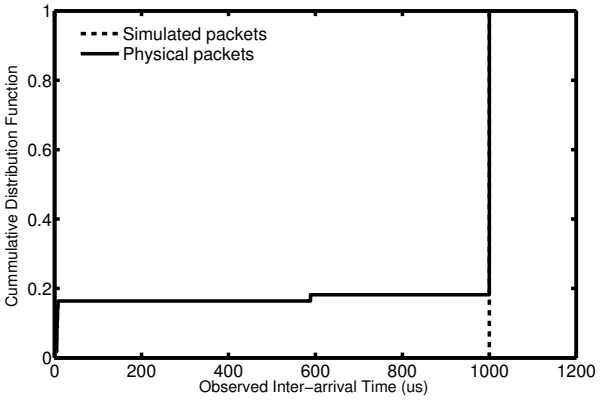

Figure 10: CDF of simulated packets' and physical packets' inter-transmission times when integrating Caliper with `ns-2`.

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 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 to later 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.

We are working on extending the Caliper software on NetThreads to enable more features while preserving the precision of the traffic. For NetThreads applications in general, the maximum achievable packet rate depends on the amount of computations done per packet and hence it also is a function of the packet size (the shortest packets are the worst case leaving less cycle budget per packet: the 125MHz clock allows for 1 cycle per packet byte per processor). Finally, we made both NetThreads and Caliper available as free software to download [1].

6. RELATED WORK

There have been many software- and hardware-based packet generators presented in the literature. Some of the software workload generators try to characterize network traffic by empirically deriving models for web traffic [29, 30], or other network applications, such as TELNET, SMTP, NNTP, and FTP [31]. Cao *et al.* [32] model the HTTP traffic and parameterize the network characteristics such as the round-trip times at the clients rather than capturing it empirically. `Netspec` [33] builds source models to generate traffic for TELNET, FTP, HTTP, voice and video.

One popular way to generate traffic for testbeds is

through packet traces from existing networks. RAMP [34] generates high bandwidth traces using a simulation environment involving source-level models for HTTP and FTP. Rupp *et al.* [35] introduce a packet trace manipulation framework for testbeds. They present a set of rules to manipulate a given network trace, for instance, stretch the duration of existing flows, add new flows, change packet size distributions, etc. Hernandez *et al.* [36] generate realistic TCP workloads using a one-to-one mapping of connections from the original trace to the test environment. Swing [19] can create responsive, closed-loop traffic with similar burstiness characteristics on multiple time scales to existing traces by estimating wide-area network characteristics.

Another popular way to create realistic traffic is to use network emulators. For example, Swing uses ModelNet [21] to emulate a network of links each with its own bandwidth, delay and drop probability. Unfortunately, relying on network emulators has its own limitations. The network is emulated in software, and the position of packets within the network is only updated e.g. 10000 times per second or once every 100 μs for ModelNet. Thus, the packets sent do not have high precision timings as roughly 8 MTU sized packets can be transmitted at 1 *Gbps* in 100 μs. Using the discrete event simulator `ns-2` as the network emulator also suffers from similar timing issues. Mahrenholz *et al.* [37] recommend several modifications to `ns-2` to improve the accuracy of its network emulation feature.

The NetFPGA-based packet generator introduced by Covington *et al.* [6] can reliably replay a trace file and capture packets at *Gbps* line rate. However, as mentioned in Section 1 the traces are based on prior recording and it would be difficult to extrapolate them to closed-loop traffic and other workload/topology scenarios.

There are a number of other available software tools for traffic generation such as `Harpoon` [38] and `tcplib` [39]. However, they are all designed to match the property distributions of a trace at a coarse granularity and none attempt to guarantee the behavior of traffic at short time scales. They ignore the unavoidable timing issues introduced by the users' hardware and OS choices. Our efforts are complementary to the above mentioned works as we focus only on constructing real packets and providing exact transmission times. To the best of our knowledge, we present the first framework for generating precise closed-loop traffic providing guarantees for inter-transmission times at very short time scales.

Soft processors have been previously used for networking [40] and provide an easy path to leverage the close integration of FPGAs with the network controllers. While a number of soft processors are available (e.g. Vespa [41]), NetThreads cores provide an increased pipeline efficiency for control-flow intensive programs through the use of multithreading [14].

7. CONCLUSIONS

Generating realistic traffic in network testbeds is challenging yet crucial for performing valid experiments. Software network emulators schedule packet transmission times in software, incurring unavoidable inaccuracy for inter-transmission intervals in the sub-millisecond range – hence they are insufficient for experiments sensitive to the inter-arrival times of packets. In this paper we present the NetThreads platform and Caliper, a precise and responsive traffic generator built on NetThreads. NetThreads allows network devices to be quickly developed for the NetFPGA card in software while still taking advantage of the hardware-level resolution. Caliper allows packets generated on the host computer to be sent with extremely accurate inter-transmission times and is designed to be integrated with existing software traffic generators and network emulators. We demonstrate Caliper's precision and integration with existing softwares to generate traffic that is realistic and accurate at almost all time scales. In our experiments, the maximum error that Caliper incurs is around 8 *ns* which is the NetFPGA's clock cycle time and also our measurement resolution. Overall, Caliper allows researchers to perform experiments that were previously infeasible.

8. REFERENCES

[1] Caliper's wiki page. http://netfpga.org/foswiki/bin/view/NetFPGA/OneGig/PreciseTrafGen.

[2] N. Beheshti, Y. Ganjali, M. Ghobadi *et al.*, "Experimental study of router buffer sizing," in *IMC '08*, 2008, pp. 197–210.

[3] A. Kuzmanovic and E. W. Knightly, "Low-rate TCP-targeted denial of service attacks," in *Proceedings of ACM SIGCOMM*, 2003, pp. 75–86.

[4] N. Beheshti, Y. Ganjali, M. Ghobadi *et al.*, "Performing time-sensitive network experiments," in *ANCS '08*, 2008, pp. 127–128.

[5] R. Prasad, C. Dovrolis, and M. Thottan., "Evaluation of Avalanche traffic generator," Georgia Institute of Technology, Tech. Rep., 2007.

[6] G. A. Covington, G. Gibb, J. W. Lockwood, and N. Mckeown, "A packet generator on the NetFPGA platform," in *FCCM '09*, 2009, pp. 235–238.

[7] J. Naous, G. Gibb, S. Bolouki, and N. McKeown, "NetFPGA: reusable router architecture for experimental research," in *PRESTO'08*, 2008, pp. 1–7.

[8] The NetFPGA project. http://www.netfpga.org/.

[9] NetThreads wiki. http://www.netfpga.org/foswiki/bin/view/

NetFPGA/OneGig/NetThreads.
[10] CommAgility Limited, "AMC-2C87W3 LTE/WiMAX Card," http://www.commagility.com, 2010.
[11] INVEA-TECH, "COMBOv2 Cards," http://www.invea-tech.com/, 2010.
[12] AllowTech, "100G Network Processor," http://www.allowtech.com, 2010.
[13] R. Teodorescu and J. Torrellas, "Prototyping architectural support for program rollback using FPGAs," in *FCCM '05*, 2005, pp. 23–32.
[14] M. Labrecque and J. G. Steffan, "Fast critical sections via thread scheduling for FPGA-based multithreaded processors," in *FPL*, 2009.
[15] J. W. Lockwood, N. McKeown, G. Watson *et al.*, "NetFPGA - an open platform for Gigabit-rate network switching and routing," in *MSE'07*, June 3-4 2007, pp. 160–161.
[16] J. Veenstra and R. Fowler, "MINT: a front end for efficient simulation of shared-memory multiprocessors," in *MASCOTS'94*, January 1994.
[17] Iperf. http://sourceforge.net/projects/iperf/.
[18] Netperf. http://www.netperf.org/netperf/.
[19] K. V. Vishwanath and A. Vahdat, "Realistic and responsive network traffic generation," in *SIGCOMM '06*, 2006, pp. 111–122.
[20] The Network Simulator - ns-2. http://www.isi.edu/nsnam/ns/.
[21] A. Vahdat, K. Yocum, K. Walsh *et al.*, "Scalability and accuracy in a large-scale network emulator," *SIGOPS Operating Systems Review archive*, vol. 36, no. SI, pp. 271–284, 2002.
[22] M. Labrecque and J. G. Steffan, "Improving pipelined soft processors with multithreading," in *FPL*, August 2007.
[23] L. Zhang, S. Shenker, and D. D. Clark, "Observations on the dynamics of a congestion control algorithm: The effects of two-way traffic," in *ACM Computer Communication Review*, 1991, pp. 133–147.
[24] N. Beheshti, Y. Ganjali, M. Ghobadi *et al.*, "Performing time-sensitive network experiments," University of Toronto, Tech. Rep. TR08-UT-SNL-09-10-00, 2008.
[25] D. Katabi and C. Blake, "Inferring congestion sharing and path characteristics from packet interarrival times," Massachusetts Inst. of Technology, Tech. Rep., 2001.
[26] R. Takano, T. Kudoh, Y. Kodama *et al.*, "Design and evaluation of precise software pacing mechanisms for fast long-distance networks," in *In Proceedings of PFLDNet 2005*, 2005.
[27] 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.
[28] K. Fall, "Network emulation in the Vint/NS simulator," in *ISCC'99*, 1999, p. 244.
[29] P. Barford and M. Crovella, "Generating representative web workloads for network and server performance evaluation," *SIGMETRICS Performance Evaluation Review*, vol. 26, no. 1, pp. 151–160, 1998.
[30] B. A. Mah, "An empirical model of HTTP network traffic," in *INFOCOM '97*, 1997, p. 592.
[31] V. Paxson, "Empirically derived analytic models of wide-area TCP connections," *IEEE/ACM Transactions on Networking*, vol. 2, no. 4, pp. 316–336, 1994.
[32] J. Cao, W. S. Clevel, Y. Gao *et al.*, "Stochastic models for generating synthetic HTTP source traffic," in *INFOCOMM*, 2004.
[33] B. O. Lee, V. S. Frost, and R. Jonkman, "NetSpec 3.0 source models for Telnet, FTP, voice, video and WWW traffic," University of Kansas, Tech. Rep. ITTC-TR-10980-19, 1997.
[34] S. Sen and J. Wang, "Analyzing peer-to-peer traffic across large networks," in *IEEE/ACM Transactions on Networking*, 2002, pp. 219–232.
[35] A. Rupp, H. Dreger, A. Feldmann, and R. Sommer, "Packet trace manipulation framework for test labs," in *IMC'04*, 2004.
[36] F. Hernndez-campos, F. Donelson, and S. K. Jeffay, "Generating realistic TCP workloads," in *Computer Measurement Group International Conference*, 2004, pp. 273–284.
[37] D. Mahrenholz and S. Ivanov, "Real-time network emulation with ns-2," in *DS-RT'04*, 2004.
[38] J. Sommers and P. Barford, "Self-configuring network traffic generation," in *IMC '04*, 2004, pp. 68–81.
[39] P. B. Danzig and S. Jamin, "tcplib: A library of TCP internetwork traffic characteristics," University of Southern California, Tech. Rep. USC-CS-91-495, 1991.
[40] K. Ravindran, N. Satish, Y. Jin, and K. Keutzer, "An FPGA-based soft multiprocessor system for IPv4 packet forwarding," in *FPL*, 2005, p. 487492.
[41] P. Yiannacouras, J. G. Steffan, and J. Rose, "Fine-grain performance scaling of soft vector processors," in *CASES*, 2009.