SLOG: Your switch is also your load-generator

Marios Kogias Martin Weber * Edouard Bugnion

EPFL, Switzerland

1 Introduction

Today's webscale datacenter applications such as search, social networking, and e-commerce, communicate using Remote Procedure Calls (RPCs) in complex fan-in and fan-out patterns under strict µs-scale tail latency Service Level Objectives (SLOs). To support those applications, software and hardware-based Network Functions (NFs), such as Google's Maglev [3] need to process millions of packets per second.

Accurate and efficient load generators and latency measuring tools are crucial in designing high-performance datacenter applications and NFs. Those tools need to satisfy three main requirements: i) measuring latency accurately in a µs scale, without being part of the measuring loop. ii) generating load at a very high throughput to stress highly scalable applications, *e.g.,* based on kernel-bypassing. iii) accurately emulate deployment conditions be generating load with a realistic inter-arrival distribution, *e.g.,* Poisson.

There is a plethora of load generators and latency measuring tools that target either NFs or RPC services. Those tools can be implemented in software and depend on software timestamping such as Mutilate [6], and YCSB [2]. Others, such as Lancet [5], and Moongen [4] are implemented in software but depend on hardware timestamping capabilities that can be found in modern NICs. Finally, there are tools that use custom hardware appliances, *e.g.,* FPGAs, such as Spirent [8] or IXIA to generate load and measure latency. There is a obvious trade-off between the accuracy and performance of hardware solutions, versus the flexibility and ease of use of the software ones. Software tools that depend on hardware timestamping try to balance this trade-off but they are still limited by the CPU processing capabilities and NIC line rates.

Emerging programmable switches, such as Barefoot Tofino [1], constitute a very appealing building block as they provide an adequate level of programmability in languages such as P4 [7] without compromising line rate performance. Such appliances change the way we design systems and allow pushing functionality, traditionally implemented at the end-hosts, to the network. To do so they depend on a configurable set of match-action tables, dataplane registers that outlive packets, and a series of fixed function units.

In this work we propose SLOG (**S**witch **LO**ad **G**enerator), a programmable load generator and latency-measuring tool based on a programmable Tofino ASIC. SLOG leverages the programming capabilities and the fixed function units of Tofino to generate load and measure tail latency for both NFs and RPC services, while being able to generate a Poisson inter-arrival distribution. According to our knowledge, SLOG is the only hardware-based tool that is able to generate a randomized inter-arrival distribution, which is crucial for a realistic latency experiment.

2 Design

We design and build SLOG on top of a Barefoot Tofino ASIC, splitting the functionality between the control and the dataplane. In our design we had to answer five basic questions: i) How to use a programmable switch to generate and timestamp packets? ii) How to match the request with the reply timestamps? iii) How to maintain the latency distribution? iv) How to generate a Poisson inter-arrival distribution? v) How can a user configure SLOG without knowing P4? We further describe our solution to each of these design questions.

Switch packet generation and timestamping: Tofino ASICs are equipped with a *packet generation engine* as a fixed function unit. The engine can be configured to generate packets either on specific types of events, or periodically based on a predefined packet format. It can send packets individually or in batches, up to 100Gbps per pipe. Generated packets are injected into the packet processing pipeline, allowing them to be processed and forwarded like regular incoming packets. We use the packet generation engine to generate packets and the pipeline logic to parametrize them.

The Tofino ASIC is also equipped with timestamping capabilities with *ns*-scale granularity both in the ingress and the egress pipeline. We collect the Tx timestamps in the egress

*Work done as part of his EPFL Master thesis

pipeline, to avoid the queuing time in the switch buffers, and the Rx timestamps in the ingress pipeline.

Matching timestamps: To compute the end-to-end latency SLOG has to match Tx timestamps of outgoing packets with the Rx timestamps of packets coming from the device under test. Due to the limited dataplane memory Tx timestamps cannot be stored in the dataplane while waiting for Rx timestamp. Also, we don't want to keep the Tx timestamp in the packet payload, since the payload might need to be specific to the application under test. Thus, we decided to put the Tx timestamp in packet fields that are returned to the switch. Specifically, we can use the transport (UDP/TCP) source port to store 16 bits of the Tx timestamp. This field is returned as a destination port in the reply. Other fields that can be used to store the rest of Tx timestamp is the source IP, following a similar logic. Considering that the switch serves a hypothetical /16 subnet we can leverage the lower 16 bits of the source IP to store part of the Tx timestamp. There are also application specific fields that can be used for such purpose. For example, for DNS experiments we can use the *transaction id* to store another 16 bits of the Tx timestamp.

Latency distribution: After collecting the Tx and Rx timestamps for a specific request, the dataplane has to keep the latency sample till the end of the experiment to identify the latency distribution. However, maintaining all the latency samples in the dataplane is not feasible. So, we resort to *histograms*. We leverage table counters to implement histograms in the dataplane. Counters count the number of hits that each table entry gets. They are automatically updated in the dataplane every time there is a table hit as part of the match-action pipeline. Counters are only read in the control plane. At the beginning of each experiment, we statically generate a table that stores a sorted set of numbers at a target granularity, that correspond to the histogram bucket boundaries. After computing a latency sample, the dataplane tries to range match the value to the bucket table with a nop action. A successful match corresponds to an increase to the equivalent bucket. At the end of the experiment, the control plane dumps the content of the buckets to generate the latency histogram.

Inter-arrival distribution: Unfortunately, the packet generation engine can only generate packets either at a fixed rate or after a specific dataplane event. None of the above, though, fits our needs to generate a Poisson inter-arrival distribution. Instead, we emulate this behavior by devising a mechanism to control the traffic rate in the dataplane. Rather than applying any inter-packet gap at the packet generator, we let it generate packets as fast as possible, and the dataplane transmits only a subset of the generated packets while dropping the rest.

Identifying the number of packets to be dropped is crucial for the inter-arrival distribution since it is translated to the inter-packet time interval. For a specific packet size, we compute the time the packet generator takes to generate a packet and inject it in the pipeline. This duration, $\Delta\tau$, is out

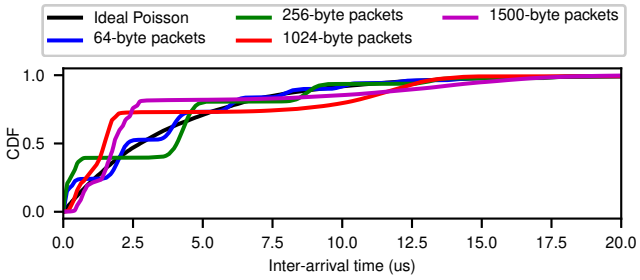


Figure 1: Approximation of a 2MPPS Poisson inter-arrival with different packet sizes observed at 1/8 target servers

granularity of control. At the beginning of the experiment the control plane generates a set of random inter-packet gaps according to the target inter-arrival distribution, transforms them to multiples of $\Delta\tau$, and stores them in a table to be accessible by the dataplane. Once the experiment starts, the dataplane randomly selects a number from this table, and drop the following N generated packets. To generate a random number in the dataplane we hash the current timestamp. Figure 1 shows how SLOG approximates a Poisson inter-arrival distribution under different packet sizes for a target load of 2M PPS.

User configuration: Users can use SLOG without changes to the P4 dataplane. We provide a convenient way to describe a test setup, by writing a configuration file. The configuration file is then read by the control plane application to set up and run SLOG. The format used is YAML and the configuration consists mainly of key-value options and lists of such options. Examples of configuration parameters are: the target host IP and port, the number of packets to be sent, the target rate, and the inter-arrival distribution.

References

[1] Barefoot Networks. Tofino product brief. https://barefootnetworks.com/products/brief-tofino/, 2018.

[2] Brian F. Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. Benchmarking cloud serving systems with YCSB. In *SOCC*, pages 143–154, 2010.

[3] Daniel E. Eisenbud, Cheng Yi, Carlo Contavalli, Cody Smith, Roman Kononov, Eric Mann-Hielscher, Ardas Cilingiroglu, Bin Cheyney, Wentao Shang, and Jinnah Dylan Hosein. Maglev: A Fast and Reliable Software Network Load Balancer. In *NSDI*, pages 523–535, 2016.

[4] Paul Emmerich, Sebastian Gallenmüller, Daniel Raumer, Florian Wohlfart, and Georg Carle. MoonGen: A Scriptable High-Speed Packet Generator. In *IMC*, pages 275–287, 2015.

[5] Marios Kogias, Stephen Mallon, and Edouard Bugnion. Lancet: A self-correcting Latency Measuring Tool. In *USENIX ATC*, pages 881–896, 2019.

[6] Jacob Leverich and Christos Kozyrakis. Reconciling high server utilization and sub-millisecond quality-of-service. In *EUROSYS*, pages 4:1–4:14, 2014.

[7] The P4 Language Specification. https://p4.org/p4-spec/p4-14/v1.0.4/tex/p4.pdf. Accessed on 20.09.2018.

[8] Spirent Communications. Spirent test modules and chassis. https://www.spirent.com/Products/TestCenter/Platforms/Modules.

SLOG: Your switch is also your load generator

Marios Kogias, Martin Weber, Edouard Bugnion

EPFL

Goal: Build a high-throughput, accurate tool for Network Function and RPC-service testing

Requirements

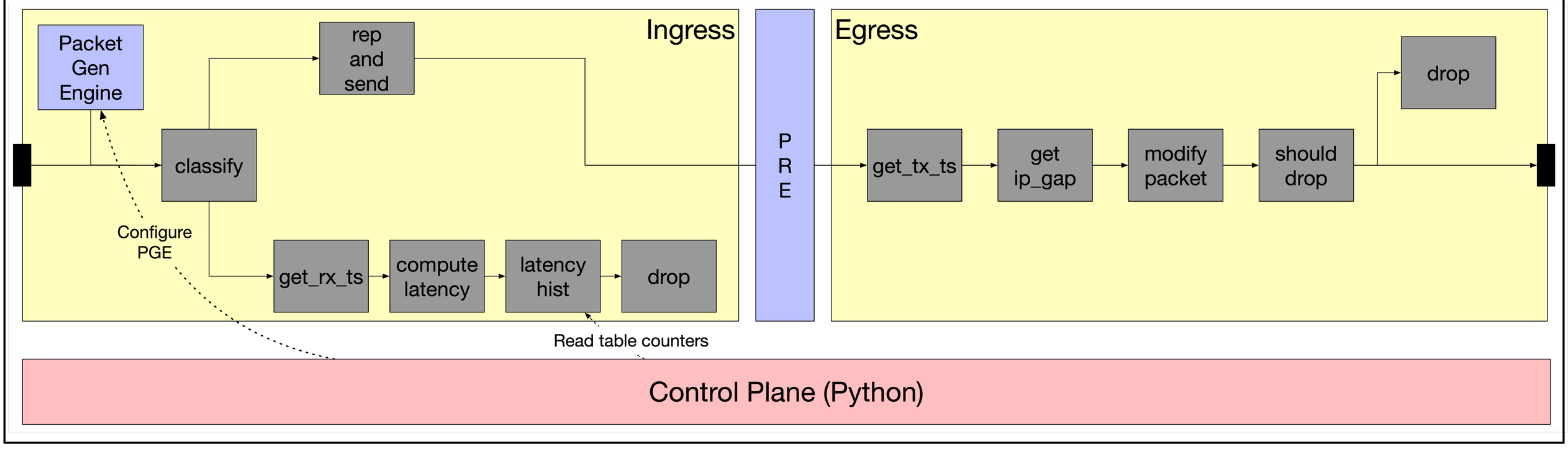
- High Throughput
- Accuracy
- Flexibility:
  - Different protocols
  - Inter-arrival time

Tool	Dataplane	Timestamp	Flexibility
Mutilate	Software	Software	High
Moongen/Lancet	Software	Hardware	High
IXIA/Spirent	Hardware	Hardware	Low
SLOG	Hardware	Hardware	Medium

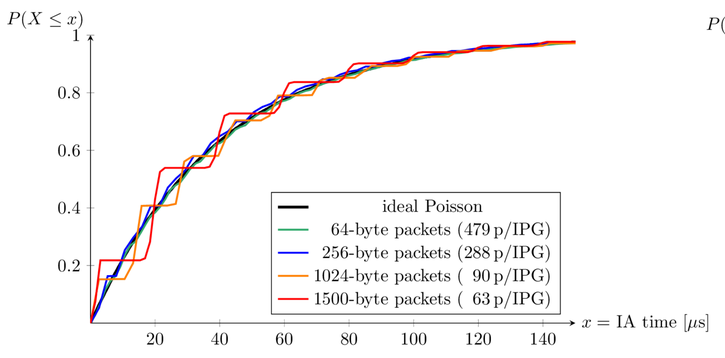
Challenges

1. **Packet generation**
   - ☞ Use Tofino's packet generation engine fixed unit
2. **Packet timestamping**
   - ☞ Use egress timestamp for Tx and ingress timestamp for Rx
3. **Timestamp matching**
   - ☞ Include Tx timestamp as part of the packet headers, e.g. UDP src port
4. **Latency statistics**
   - ☞ Maintain a latency histogram in the P4 dataplane
5. **Inter-arrival distribution**
   - ☞ Generate packets at line rate and drop in the dataplane for Poisson
6. **User configuration**
   - ☞ Configure the P4 dataplane using a YAML cfg file

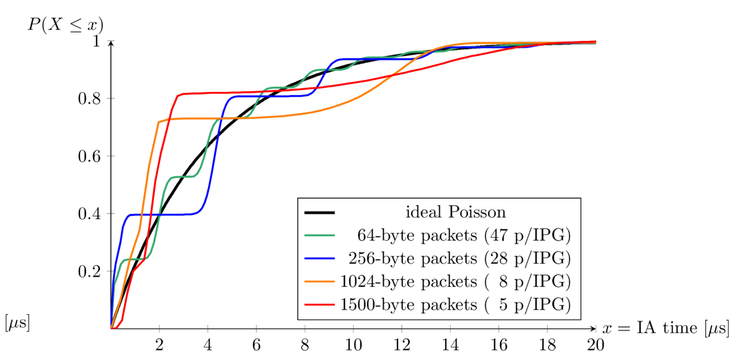
Design



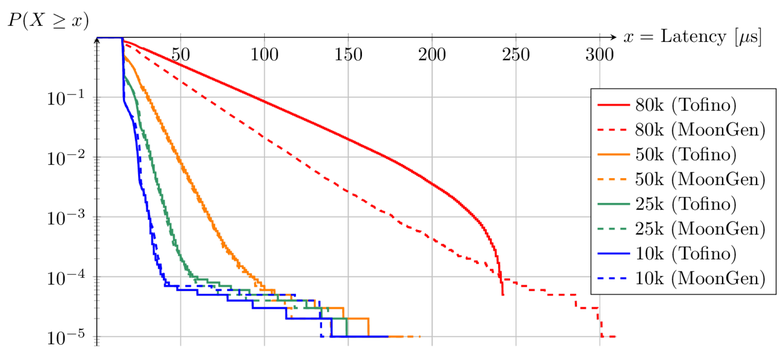
Evaluation



200 Kp/s



2 Mp/s



Synthetic server
max throughput: 100kps
Comparison with Moongen

YAML Configuration

```yaml
ia_distribution: [fixed, poisson]  # Interarrival distribution

histogram_ranges:                  # List of ranges in the histogram
  - [{numeric}, {numeric}]         # - Defines start and end (both inclusive) of
  - [{numeric}, {numeric}]         #   a range.
  ...

pktlen: {numeric}                  # Ethernet frame length
                                   # Ignored for `pkttype` == 'dns'

pktcount: {numeric}                # Number of packets to be transmitted to each
                                   # target

pktrate: {numeric}                 # Total packet transmission rate (split in
                                   # case of multiple targets) in packets/second

local:                             # Network parameters for the switch itself
  - macDstAddr: {MAC address string}
    ipDstAddr: {IPv4 address string}
    udpPort: {numeric}

hosts:                             # Network parameters for target devices
  - switchPort: {numeric}          # Output port on the switch for this target
    macDstAddr: {MAC address string}
    ipDstAddr: {IPv4 address string}
    udpPort: {numeric}
    dummy: {boolean}               # Not a real host (packets dropped in egress)
  - switchPort: {numeric}
    ...
```