FlowScope: Efficient Packet Capture and Storage in 100 Gbit/s Networks

Paul Emmerich, Maximilian Pudelko, Sebastian Gallenmüller, Georg Carle
Chair of Network Architectures and Services
Department of Informatics
Technical University of Munich
{emmericp|gallenmu|carle}@net.in.tum.de, maximilian.pudelko@tum.de

***Abstract*—Tools to capture and analyze traffic are found in every network operator's toolbox. Traffic dumps are essential to the process of debugging network issues and for network forensics. Capturing traffic is a performance-intensive and challenging task for high-speed networks. Therefore, network operators often rely on sampling a random subset of the traffic instead of capturing the network traffic in its entirety. Sampling is not always suitable, for example, network forensics applications require a full dump of the traffic to determine the source of an attack.**

We present FlowScope, a tool to continuously capture and store packets in an in-memory ring buffer. A filtered subset of the acquired packets can be dumped to disk if a specified trigger event occurs. We report benchmark results of 120 Gbit/s with 128 byte packets. This is achieved by using a novel ring buffer data structure that is optimized for high packet throughput. FlowScope is available as free software under the MIT license at https://github.com/emmericp/FlowScope.

I. Introduction

Traffic dumps are an essential tool for operators to trouble-shoot network issues. Acquiring such traffic dumps is a challenging task on high-speed links exceeding 10 Gbit/s. A common solution to cope with high bandwidths is taking random samples from the traffic. Sampling can sometimes even be done directly on the control plane of switches by configuring the control plane as a mirror port target [4]. Specialized protocols like sFlow [36] rely on hardware capabilities of switches to centrally collect packet samples to monitor traffic. These sampling-based approaches work sufficiently well for many scenarios. However, sometimes full traffic captures are needed to debug a network issue or for network forensics.

Traffic captures are usually started manually after a problem is noticed. Finding the root cause of a problem in a post-mortem analysis requires capturing packets before the problem occurs. Another scenario where starting the capture process after the fact is too late for network forensics to determine the source of an attack. Capturing and storing all traffic in persistent storage is expensive and hence generally not feasible in high-speed networks. One solution is continuously capturing packets in a large ring buffer in RAM and dump them to persistent storage on demand.

We present FlowScope, a tool to capture and store packets at rates above 100 Gbit/s. FlowScope is based on the idea of a digital storage oscilloscope: it continuously stores packets in a ring buffer and dumps a subset of them to disk on a trigger event. External sources such as an HTTP interface or observed packets matching a user-defined filter can act as trigger events. The user specifies two filters to FlowScope: a trigger filter and a dumping filter. The former specifies the start of the dumping process, the latter the kind of packets to be stored. FlowScope supports two different kinds of filter configurations: pcap filters or Lua-scriptable filters. For instance, the pcap dumping filter `"host $srcIP"` dumps packets from and to the source IP address of the initial trigger packet.

Usage scenarios for FlowScope are trouble-shooting and forensics applications in high-speed networks. We have already successfully deployed FlowScope to identify a rarely occurring anomaly on a 100 Gbit/s data center interconnect link. Other existing tools were not able to find the root-cause of this issue due to their limited performance.

Our main innovation over existing similar systems such as the Bro Time Machine [29] is our novel data structure QQ (queue of queues). QQ is an efficient multi-threaded queue implemented as a ring buffer optimized for packet storage. It scales to multiple producers and consumers and rates beyond 100 Gbit/s and 120 Mpps (million packets per seconds). This is 50 times faster than Bro Time Machine which only managed to process 2.4 Mpps in our tests. This is achieved by trading latency for throughput: FlowScope incurs a latency in the millisecond range but gains throughput and scalability. We also move the packet capturing backend to DPDK [23] which speeds up packet capture compared to the commonly used libpcap framework [42]. Further, we optimize the disk access: FlowScope can saturate multiple high-speed PCIe SSDs when persistently storing captured packets.

The remainder of the paper is structured as follows: We start by discussing the related work and technical background in Sections II and III. Section IV shows the architecture and design of FlowScope, Section V focuses on the implementation and evaluation of our queuing data structure. Section VI discusses how to run the published code to reproduce our evaluation of FlowScope. Both FlowScope and QQ are available as free software under the MIT license on GitHub [12].

II. Related Work

Capturing traffic in high-speed networks is the traditional domain of specialized hardware that is offered commercially.

ISBN 978-3-901882-94-4

Endace offers data acquisition PCIe adapters capturing up to 40 Gbit/s [16]. Standalone appliances that perform hardware-assisted packet capturing are provided by fmad [20]. Open source hardware solutions are available based on the NetFPGA platform [3].

The rise of high-speed packet processing frameworks such as DPDK [23], netmap [38], and PF_RING ZC [34] has enabled high-speed packet processing tasks in software running on commodity hardware. Multiple efficient software traffic capture and storage systems have been designed since 10 Gbit/s Ethernet became commodity over the past years. Important works are n2disk [9], [32], a traffic recorder that can write 10 Gbit/s traffic to disk with minimum sized packets using multiple capture threads. They recommend using at least 10 hard disks in parallel to store 10 Gbit/s of traffic. FloSIS [30] was evaluated at up to 30 Gbit/s using 24 hard disks in parallel.

The main bottleneck for even higher speeds is the storage back end: it needs to write a large amount of data fast. These are two conflicting optimization goals for a storage system, as they are typically either large, slow, and inexpensive (HDDs) or fast, small, and expensive (SSDs). Our approach of using an in-memory ring buffer as intermediate storage and only dump flows that are deemed interesting to persistent storage mitigates this bottleneck.

This architecture and basic idea of FlowScope is similar to the Bro Time Machine (Bro TM) [29] published in 2005. It continuously stores packet captures into ring buffers of different sizes depending on a programmable classification. These ring buffers can then be dumped on triggers from an external intrusion detection system. Bro TM builds on a simple ring buffer that stores packet data back-to-back in one block of memory and maintains index pointers in a second ring buffer. Concurrency is achieved by using different ring buffers for different traffic classes. However, access to each ring buffer is controlled by a single lock for all reader and writer threads (cf. implementation in `FifoMem.cc` [8]). Traffic classes are typically defined by protocol in the example configurations, e.g. HTTP traffic would be stored in one ring buffer. This presents a serious bottleneck for traffic classes that receive a large amount of traffic, especially as a reader thread analyzing or dumping the data will also limit the data acquisition. Bro TM was designed for 1 Gbit/s networks [29] – state of the art technology in 2005. We found it unsuitable for monitoring links of capacities beyond 10 Gbit/s due to its design limitations.

Our main innovation and contribution is the queuing data structure queue of queues (QQ). Fast concurrent queues are a common building block for multi-threaded applications and many implementations of such data structures exist. These queues are typically written as *lock-free* code utilizing low-level memory synchronization primitives. A good introduction to general lock-free programming and a lock-free queue design was published by Sutter in 2008 [41]. Popular fast lock-free queues can be found in libraries such as Facebook's folly [19], Intel's TBB [24], and Boost [6]. Desrochers published a lock-free queue design in 2014 and provides a detailed performance comparison with other commonly used queues [11].

Lock-free queues also optimize for a low latency (in the nanoseconds range) between a producer and a consumer thread. This is often an important property, e.g., when used in the backend of a financial exchange platform [43]. However, we are not interested in such low latencies for our application. Detecting trigger events within milliseconds is sufficient since the history is available anyways. Hence, we only aim for a latency in the range of hundreds of milliseconds for QQ. This lowered requirement enables the use of code with traditional synchronization mechanisms like locks without sacrificing throughput. Such code leads to a simpler overall design and simplifies the implementation of features like random access to variable-length entries in an efficient manner with multiple producers and consumers, a feature that is not found in any lock-free queue design. Moreover, writing correct lock-free code is difficult, bugs due to subtle mistakes have been found in peer-reviewed published code [40].

III. Technical Background

Identifying the key technologies is crucial for understanding the performance of FlowScope and its underlying data structure QQ.

A. DPDK

DPDK (Data Plane Development Kit) is an open source project started by Intel providing efficient userspace drivers for common server network interface cards (NICs) by multiple vendors [23]. It maps the PCIe address space of NICs directly into a userspace process, thus eliminating overhead associated with the network stack of a general-purpose operating system. This allows receiving packets in a userspace process in less than 100 CPU cycles [21]. The drawback is that DPDK offers no protocol stack, it merely sends and receives packets without processing on any higher layer. FlowScope does not need a protocol stack, we are only interested in storing, analyzing and dumping raw packets – analysis is also handled by specialized libraries or user-defined code in FlowScope.

Moreover, DPDK provides utility functions commonly needed for multi-threaded packet processing applications. Of particular interest is the `rte_ring` data structure, a queue optimized for passing packets between threads. We evaluate this queue and compare it to our implementation.

B. libmoon

libmoon is a framework for packet processing in Lua [13], [14]. It combines DPDK with LuaJIT [35] providing a fast scripting environment for quick prototyping of packet processing tools. Furthermore, libmoon bundles other third-party libraries that are commonly required in packet processing applications. Relevant libraries used for this work are the Turbo webserver [1] and the pflua pcap filter compiler [5].

All high-level parts of FlowScope, i.e., handling triggers, the HTTP API, managing threads, and configuration, are written in Lua. Further, user-provided Lua code to analyze and filter

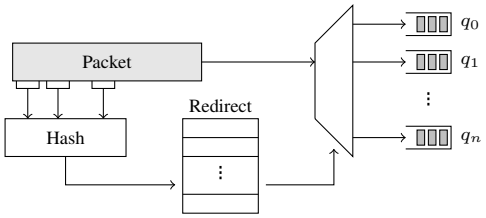

Figure 1. Receive side scaling

traffic can be embedded. These custom filters and triggers can utilize libmoon's fast and efficient protocol stack framework to dissect and analyze packets [39].

C. Offloading Features of NICs

Offloading capabilities are an integral part of modern NICs. Two features are particularly noteworthy here: VLAN offloading and Receive Side Scaling (RSS). VLAN offloading allows moving a IEEE 802.1Q tag [7] from the Ethernet header into the packet metadata. This ensures that the Ethernet header has a fixed length, speeding up the analysis. RSS sets up multiple receive queues on the NIC that can be used concurrently and independently from multiple threads. Traffic is distributed via hashing configurable protocol headers, ensuring that packets belonging to the same flow are directed to the same queue.

Figure 1 illustrates how RSS processes an incoming packet. The driver can configure which header fields are hashed and how many queues are used. Limitations like the header fields, hash function, and number of queues are hardware-specific. For example, the Intel 10 and 40 Gbit/s NICs used by us support 16 and 256 RSS queues respectively and both IPv4 and IPv6 with UDP or TCP ports for hashing [26], [27]. This is sufficient for our purposes.

D. Queues and Ring Buffers

The following definitions and concepts for queues will be used throughout this paper.

A *bounded queue* is a queue of a fixed size. Bounded queues are typically implemented as buffer that is allocated once and used in a circular manner, i.e., a *ring buffer*. Inserting elements into full ring buffers can either fail gracefully, block until space becomes available, or override the oldest entry efficiently. We use the term *ring buffer* to refer to a bounded queue that supports overriding old entries on insertion. Implementations for unbounded queues typically involve linked lists and do not support overriding the oldest entry, these queues are generally unsuitable for FlowScope.

Threads writing to a queue are called *producers*, threads reading from it *consumers*. Concurrent queues can support a single or multiple producers and consumers. The most common variants are single-producer/single-consumer (*SPSC*) and multi-producer/multi-consumer (*MPMC*) queues.

IV. FlowScope

Figure 2 shows a high-level overview over the different components in FlowScope. The analyzer and dumper are

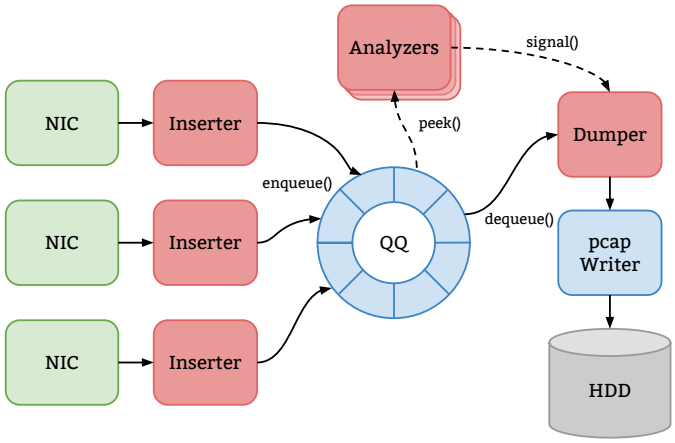

Figure 2. Architecture of FlowScope

written in Lua as they run filters which are either user-defined Lua code or pcap filters compiled to Lua. All other components are implemented in C++.

QQ is the central ring buffer data structure, discussed and evaluated in detail in Section V. *Inserters* act as producers reading traffic from an assigned RSS queue from a NIC and copying packets into QQ. Inserters can also trim large packets if only headers are required. A zero-copy solution where the NIC places the packets directly into QQ is not possible: commodity NICs copy packets into fixed-size DMA buffers. Hence, packets must be copied to store them back-to-back in the queue.

QQ offers two different kinds of consumers - *analyzers* and *dumpers*. An analyzer peeks at packets in the queue without removing them from the queue. Multiple analyzers are supported: QQ provides a `peek()` method that shares a pointer between the workers to distribute work. The analyzer that detects a trigger sends a signal to the dumper thread containing the packet that matched the filter. The dumper thread listens to trigger signals that can either come from the analyzer thread or from external sources such as the HTTP API.

Note that this design does not preserve packet order if multiple inserters are used. Intra-flow order is preserved by RSS on the NIC, cf. Section III-C. Analyzers and dumpers will only see packets of different flows out of order. Inserters take a timestamp for each received packets this allows interleaving flows to create an inter-flow order if desired, e.g., in the dumper to produce pcap files with monotonic timestamps.

A. Triggers

Simple triggers are provided in the libpcap filter language [42]. We use pflua [5] to compile and execute the filters with LuaJIT, this method is faster than other common libpcap filter compilers [22].

More complex filters can be defined as Lua code with full support for the libmoon protocol framework [39] in the filter functions. Listing 1 shows an example based on the example `trigger-payload.lua` from the examples in the

```lua
local pktLib = require "packet"
local ffi    = require "ffi"
local eth    = require "proto.ethernet"
local ip     = require "proto.ip4"

local searchStr = "EXAMPLE"
return function(pkt)
  local udpPkt = pktLib.getUdp4Packet(pkt)
  if udpPkt.eth:getType() == eth.TYPE_IP
  and udpPkt.ip4:getProtocol() == ip.PROTO_UDP then
    local data = ffi.string(udpPkt.payload.uint8,
      #searchStr)
    return data == searchStr
  end
end
```

Listing 1: Filtering with custom Lua code

FlowScope repository. It checks whether the packet is an IPv4 UDP packet and then raises the trigger if the payload begins with a specific string. These Lua functions can be arbitrarily complex, e.g., a fully featured deep-packet inspection (DPI) library can be attached here. A good candidate is nDPI [33] with the ljndpi [2] LuaJIT interface.

Triggers can also be raised externally: A custom HTTP REST API provides an endpoint to start the dumper with a custom libpcap filter.

B. Dump filters

The dumper thread is notified of every trigger event and receives the packet that raised the trigger. A dump filter is a function that derives a filter from a given packet, this filter will then be applied to decide whether to include a packet in the resulting pcap file. A simple example is the libpcap filter `"host $srcIp"` which dumps all packets coming from or going to the source of the trigger packet. The full documentation of available replacements can be found in our GitHub repository [12]. Moreover, a user-provided Lua function can be used for the dump filter. This function receives the trigger packet and returns a predicate function for packets.

Packets within the configurable time interval $[t_s, t_e]$ depicted in Figure 3 are evaluated by the filter. The dumper first jumps back in the queue to search for the first packet that arrived after t_s, it then starts filtering and saving packets until t_e is reached, waiting for new packets if necessary. t_s should be chosen large enough that the dumper does not have to jump back to the very beginning of the queue to prevent conflicts with the inserters that are at that position. The dumper is prioritized over inserters, i.e., packets after the trigger are lost if the inserters catch up to the dumper. Note that the dumper is initially faster as it writes to the disk cache, mitigating this problem.

C. Writing pcaps

We rewrote the pcap writer in libmoon for FlowScope, it previously relied on appending to files with the `write()` syscall which proved to be too slow. Our updated implementation sparsely allocates the file with `fallocate()` and then calls `mmap()` to map it into memory. The allocated file size is grown exponentially. Incoming packets are then simply copied into the memory area. The operating system handles writing

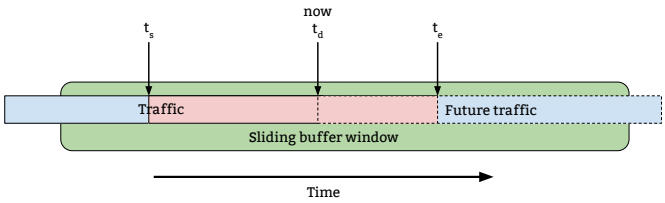

Figure 3. Traffic stored by the dumper thread

out the data to disk transparently and asynchronously in the background.

We test this approach with two Samsung SM951 NVMe PCIe SSDs configured as software raid 0. Our pcap writer manages to write out small packets at 1900 MB/s in this configuration after the file system cache was completely filled when writing a file larger than the system memory. Linux reports 100% utilization of both disks, i.e., the disks are the bottleneck.

D. Evaluation

We evaluate FlowScope on a server with two 10-core Intel Xeon E5-2630 v4 CPUs, 128 GiB DDR4 RAM, two Samsung SM951 NVMe PCIe SSDs. The used NICs are two dual-port Intel XL710 40 Gbit/s NICs, a dual-port Intel X520, and a dual-port Intel X540. The total network bandwidth is limited to 120 Gbit/s at 128 byte packets due to limitations of the Intel XL710 NICs: they cannot saturate both ports at the same time due to hardware limitations and require a minimum packet size of 128 byte to achieve the full 40 Gbit/s [28]. MoonGen [14] on a second host (same NIC configuration, directly connected) is used to generate packets at full line rate, i.e., 101.35 Mpps with 128 byte packets.

We compare the capturing performance of FlowScope with Bro Time Machine (TM) [29] and tcpdump [42] in Figure 4. Note that the figure has a discontinuous y axis as FlowScope is over 6 times faster than Bro TM measured in data rate. Both Bro TM and tcpdump are particularly slow when faced with a large number of small packets: Bro TM managed 2.4 mpps and tcpdump only 0.3 Mpps when capturing 128 byte packets. FlowScope is able to capture the full 120 Gbit/s to memory at all tested packet rates.

tcpdump was configured to write to `/dev/null` to avoid hitting disk bottlenecks but it still falls behind the specialized in-memory capturing tools. Bro Time Machine is used with a different number of traffic classes (TCs) with traffic evenly distributed between them. Adding traffic classes initially increases the capturing performance as more ring buffers and threads are used. However, more than 3 TCs lead to diminishing returns.

Unlike the other two contenders, Bro Time machine supports tracking of individual flows. We experimented with different numbers of flows per traffic class and found that a single flow performed best in this scenario.

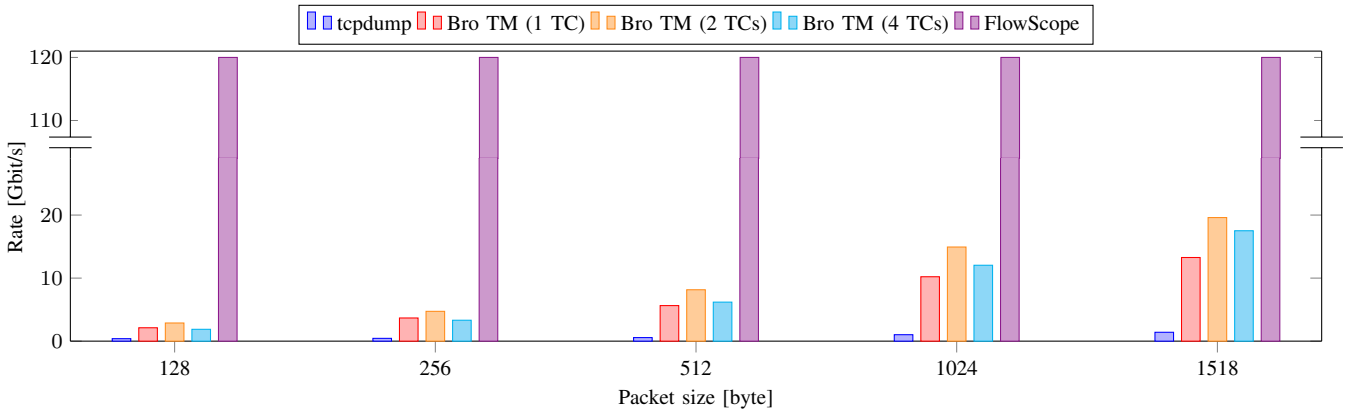

Figure 4. FlowScope vs. Bro Time Machine and tcpdump, note the discontinuous y axis

V. QQ: Queue of Queues

This section describes the design of QQ and compares the features and performance with other queueing data structures. All benchmarks performed in this section are executed on a 6-core Intel Xeon E5-2620 v3 CPU.

A. Requirements

Design goals of queues are typically a high throughput measured in messages per second and a low latency. However, as mentioned in Section II, we are not interested in low latency here. FlowScope only requires a latency in the millisecond-range, so we can make a trade-off between latency and throughput. Our requirements are summarized as follows:

- R1 MPMC support.
- R2 Efficient storage of variable-length objects.
- R3 Bulk operations for efficiency.
- R4 Ring-buffer operation, producers implicitly override the oldest entries.
- R5 Different priorities for consumers: dumpers must block the producers, analyzers that fall behind must be advanced implicitly by the producers.
- R6 Iterating through elements without consuming them.
- R7 Nice-to-have but not strictly necessary is support for random access to efficiently start writing out packets at a specified time.

B. Design

QQ is implemented as a queue of queues, i.e., it is a large *outer queue* containing multiple small *inner queues*.

Inner queues are simple containers for packets with accompanying metadata and a mutex for synchronization. All inner queue elements are allocated from a large memory area that is created by QQ. Producers acquire exclusive ownership of inner queue elements from this area, fill them, and eventually release them. As every producer has its own inner queue with exclusive access, writing to these inner queues does not need to be synchronized between different producers, hence accelerating write performance. However, the inner queues can only be consumed after the producer releases them.

The outer queue is a ring buffer consisting of inner queue elements, MPMC support is limited by a single mutex for the outer queue. Note that the outer queue is accessed rarely: each inner queue stores a large amount of data and only acquiring and releasing an inner queue requires locking the mutex. Hence, there are few accesses to the outer queue per second, so the mutex on the outer queue does not pose a bottleneck.

This design imposes a maximum time that an inner queues element can be hold by a producer. A producer that owns a queue element indefinitely blocks a consumer: all inner queues are stored directly and consecutively in the outer queue, there is no additional indirection. Consumers that are directly behind producers in the queue receive an element that is locked by the producer and have to block. The maximum hold time is checked by the inner queue every time an element is inserted using the low-overhead `RDTSC` instruction. Moreover, this sets a limit on the time required for a packet to traverse the queue.

C. Inner queue size

Tuning the size of the inner queue is crucial for QQ: Large inner queues lead to high latencies, small inner queues affect the throughput as the outer queue becomes the bottleneck. Inner queue elements should be multiples of the CPU's page size to improve utilization of per-core TLB (translation lookaside buffer) caches [25]. A common and recommend optimization when handling large consecutive data elements is using a larger page size than the default 4 KiB of our CPU [25], [44]. Modern Intel CPUs support 2 MiB and 1 GiB page sizes beside the default 4 KiB. Transparent huge pages (THP) in the Linux kernel make 2 MiB pages particularly easy to use [44], 1 GiB pages require a more elaborate setup. Hence, we support a configurable multiple of 2 MiB for the inner queue size.

We test exponentially increasing queue sizes from 2 MiB to 128 MiB in a configuration with 4 producers and a single consumer utilizing the queue with 64 byte packets. The performance difference between 2 MiB queues and 128 MiB sizes is only 1.5%. Therefore, we use a default size of 2 MiB to optimize latency, especially at lower throughputs.

Table I
BOTTLENECK IMPOSED BY THE OUTER QUEUE

Threads	1	2	3	4	5	6	7	8
Ops/s	2.1 M	800 k	340 k	240 k	200 k	190 k	175 k	170 k

2 MiB pages can contain $167.8\,\mu s$ worth of data at 100 Gbit/s. This translates to 11.9 k operations on the outer queue per 100 Gbit/s of traffic, a potential bottleneck. We stress-test the outer queue operations to quantify this bottleneck, Table I shows that our configuration is feasible even at higher data rates.

D. Latency and capacity

The best-case latency is achieved at high packet rates and determined by the inner queue size. Worst-case latency occurs at lower packet rates and is determined by the configurable maximum hold time for producers. As with any queue, latency is even worse if consumers fall behind the producers. We only discuss the case where the consumers can keep up with the producers here, latency in the other case is simply the total queue size.

To reduce resource contention, we impose a minimum distance between producers and consumers equal to the number of producers. This increases the total throughput by 59% in a test case with 4 producers and 4 consumers. Taking this into account, the latency l can be derived from the inner queue size $s_i = 2\,\text{MiB}$, the data rate r, the number of producers n_p, and the maximum hold time t_{max}.

$$l = \min(\frac{s_i}{r}, t_{max}) \cdot n_p$$

Assuming one inserter per 10 Gbit/s of traffic, the latency is 1.67 ms if the link is fully utilized (t_{max}, if chosen correctly, is not hit at full rate). Lower rates pose an optimization problem between storage capacity and latency: Choosing the t_{max} too high affects latency at lower rates. A hold time set too low leaves unused space in the fixed-size inner queues as producers are forced to give up their inner queue before it is completely filled. t_{max} can be selected to match the lowest expected data rate r_{min}:

$$t_{max} = \frac{s_i}{r_{min}}$$

For example, in a setup with 10 producers, the default 2 MiB inner queue size and an expected data rate between 1 Gbit/s and 100 Gbit/s, t_{max} should be set to 16.77 ms. This leads to an overall latency between 1.67 ms and 167.7 ms. The storage capacity of QQ is best measured in time at a specific data rate $c(r)$ depending on the total amount of memory allocated s_t.

$$c(r) = \min(\frac{s_t}{r}, \frac{t_{max} \cdot s_t}{s_i})$$

Using the previous guideline for choosing t_{max} leads to an upper bound for the stored time equal to the amount of data that can be stored at r_{min}.

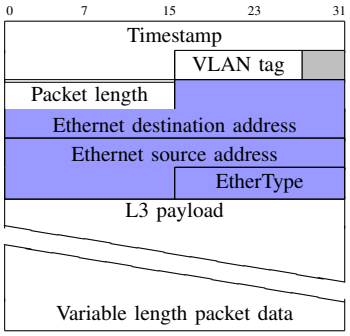

Figure 5. Packet layout used by the capture threads

E. Packet storage

Naïvly filling up the inner queues as densely packed as possible leads to poor data alignment of packet headers, slowing down the analyzers, especially on older CPUs [25]. Aligning packets at a 4 byte or 8 byte boundary is also not optimal: the 14 byte Ethernet header leads to poorly aligned header fields in layer 3 and layer 4 protocols. Metadata must also be stored in the queue: a timestamp (48 bits), the length of the packet (16 bits), and the 802.1q VLAN tag (12 bits) extracted by the NIC. A 48 bit timestamp is sufficient since it only needs to be unique for all packets currently in the queue. 48 bits at 1 ns resolution are sufficient for over 3 days worth of packets in memory.

Figure 5 shows how QQ stores a packet and its metadata. The whole structure is aligned to 4 byte boundaries leading to correct alignment of 4 byte layer 3 header fields such as IPv4 addresses.

F. Performance

We configure QQ with 1 to 5 producers that insert packets sized between 64 byte and 1518 byte into the queue. A single consumer continuously reads from QQ and counts the packets before freeing the inner queues, this single consumer is fast enough to keep up with all five producers. The producers copy the same packet data over and over into QQ.

Figure 6 visualizes the results and compares the throughput to the memory bandwidth measured with the STREAM benchmark [31]. QQ scales well with the number of cores when using small packets, larger packets see lower increases from a higher number of producers. This can be attributed to memory bandwidth as the memory throughput measured by the STREAM benchmark behaves in a similar way when more threads are added.

G. Comparison with other queues

We compare the features and performance of popular queues: Facebook's ProducerConsumerQueue [18] and MPMCQueue [17], Desrochers' ReaderWriterQueue [10] and ConcurrentQueue [11], and DPDK's rte_ring [23]. We omit implementations found Intel's TBB [24] and in Boost [6] because evaluation done by Desrochers [11] demonstrates that they are slower than other available alternatives. Table II is

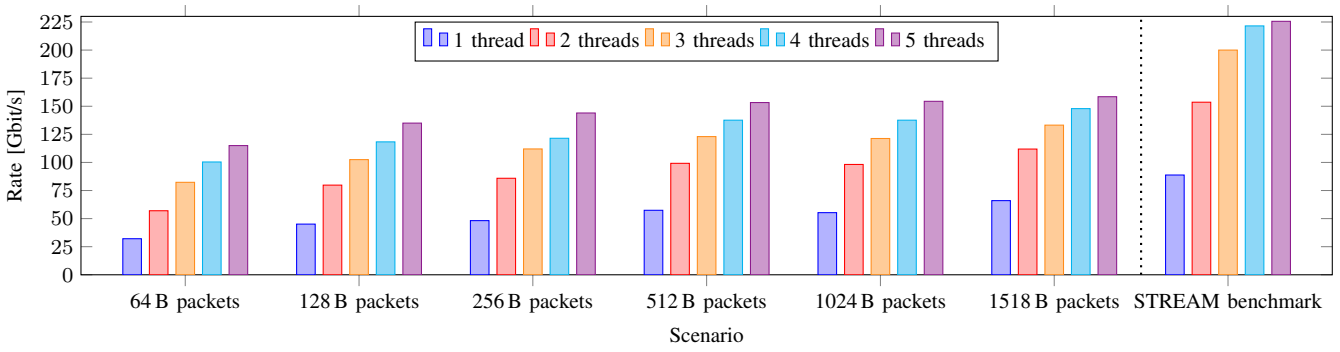

Figure 6. Throughput of QQ with different packet sizes and inserters compared to STREAM memory benchmark

Table II
QUEUE COMPARISON

Feature	ReaderWriterQueue [10]	ConcurrentQueue [11]	ProducerConsumerQueue [18]	MPMCQueue [17]	rte_ring [23]	QQ
MPMC support (R1)	✗	✓	✗	✓	✓	✓
Variable length objects (R2)	✗	✗	✗	✗	✗	✓
Bulk operations (R3)	✗	✓	✗	✗	✓	✓
Ring buffer (R4)	✗	✗	✗	✗	✗	✓
Special functions (R5-R7)	✗	✗	✗	✗	✗	✓
Growable	✓	✓	✗	✗	✗	✗
Low latency	✓	✓	✓	✓	✓	✗

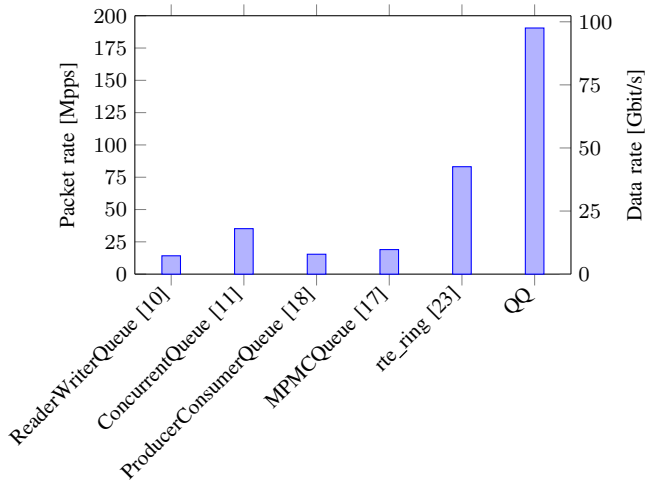

Figure 7. Throughput comparison with 64 byte packets

a feature comparison matrix showing that QQ only fails to support low-latency and dynamic memory allocation – two features that are not required for FlowScope.

Support for variable length objects can be achieved with all queues: simply allocate memory of the correct size for each packet. This approach is inefficient: memory allocation becomes a bottleneck, memory fragmentation occurs, and reading data leads to random accesses instead of sequential accesses that can be prefetched. QQ stores all data directly and sequentially in a large ring buffer: the outer queue contains the inner queues directly with no additional indirection.

Figure 7 compares the throughput achieved by the different queues with 4 producers and 4 consumers sending 64 byte packets through the queue. SPSC queues only use a single producer and consumer and are hence significantly slower. To make this a fair comparison, we emulate how a properly optimized application would use these third-party queues: we insert message buffers containing 64 packets into them. Further, we avoid allocations by re-using the same buffers over and over again, i.e., this is synthetic benchmark comparable to the ”heavy concurrent“ benchmark from [11]. QQ proves to be the fastest queue by a factor of two, however, the latency of all other queues is better by several orders of magnitude.

Low latency implies that a large amount of data needs to be moved between different CPU cores. This leads to resources contention in the memory subsystem – accessing data that is currently owned by a different core is a slow operation [25]. QQ enforces a minimum distance between producers and consumers. This trade-off between latency and throughput increases throughput by 59%, this is both due to reduced locking and fewer cache conflicts. The latency impact is in the range of tens of milliseconds, the exact value depends on the number of producers and data rate, cf. calculations in Section V-D.

VI. REPRODUCIBLE RESEARCH

Both FlowScope and QQ are available as free software under the MIT license on GitHub [12]. Readers interested in validating of the performance and functionality of FlowScope can use a MoonGen script published on a separate GitHub repository [15]. Our evaluation at 120 Gbit/s with 128 byte packets in Section IV-D can be reproduced with this setup.

`test-high-background-traffic.lua` in the repository is a MoonGen script that generates random flows within the subnet 10.0.0.0/16 on random ports in the range 1000 to 10000. One of these flows will generate a single packet to UDP port 60000 with a different payload after a configurable delay. This packet can be used as a FlowScope

trigger, by matching either on the port or on the payload. An appropriate dumping filter can be used to dump only traffic of this 5-tuple flow or all packets from/to the IP address that generated the trigger packet.

Follow these steps to test FlowScope with this setup:

- Directly connect two servers A and B with several high-speed links, server A is the device under test (DuT), server B the packet generator.
- Install FlowScope [12] on server A.
- Install MoonGen [14] with the script found in our test repository [15] on server B.
- Run `./libmoon/build/libmoon flowscope.lua 0 --trigger-expr 'udp port 60000' --dumper-expr 'host $srcIP'` on server A.
- Run `./MoonGen test-high-background-traffic.lua 0 -t 10` on server B, this will generate the trigger packet after 10 seconds of traffic.

Run FlowScope and the MoonGen scripts with `--help` to see further options, e.g., to use multiple ports and threads. Matching on payload instead of the UDP port to benchmark the analyzer under higher load can be done by using the trigger example `trigger-payload.lua` that comes with FlowScope. FlowScope will report the current throughput of all capturing threads, the analyzer and the performance of the pcap writer thread once the trigger packet has been detected. We last tested this procedure with git revision `3678ebc` of FlowScope. `README.md` in the FlowScope repository [12] will be kept up-to-date if anything in this demo setup changes.

VII. Conclusions and Future Work

Previous in-memory ring buffer systems were bottlenecked both by inefficient packet IO libraries and by inefficient data structures. The performance optimizations discussed in this paper can help others building similar high-speed concurrent data structures to analyze or capture traffic on commodity hardware. FlowScope is a demonstrator for the possibilities of capturing solutions that build on QQ, our queueing data structure optimized for packet capturing and analysis. The clear separation between the implementation of the QQ data structure and the FlowScope demonstrator applications allows others to integrate QQ into their systems. We demonstrate the feasibility of capturing and analyzing traffic in networks at rates beyond 100 Gbit/s by building on the fast packet IO backend DPDK. Our work presents an increased performance of factor 50 over the state of the art.

QQ was implemented as part of a thesis by one of the authors. Readers interested in further implementation details of this part are referred to his thesis [37].

We plan to apply FlowScope to both high-speed Internet uplinks and Internet exchange points where we found existing tools to be unsuitable due to performance limitations. In particular, we have previously been monitoring a high-speed internet uplink and identified flows that show anomalous behavior seconds or minutes after they start. We have been unable to capture the full flow as the anomaly cannot be detected when the flow starts.

We are currently working on integrating efficient flow tracking to simplify integration of analyzers operating on data streams, as well as more a more comprehensive API. Refer to our GitHub repository [12] for documentations and examples of the latest features.

Acknowledgments

The authors would like to thank First Colo GmbH for helpful real-world applications of FlowScope and Johannes Reifferscheid for coming up with the name QQ and contributing to its design. This research is supported by the German BMBF projects X-CHECK (16KIS0530), DecADe (16KIS0538), and SENDATE-PLANETS (16KIS0472). The authors alone are responsible for the content of the paper.

References

[1] J. Abrahamsen. Turbo. `https://github.com/kernelsauce/turbo`.

[2] Adrian Perez. ljndpi: The SnabbWall sidekick, Jan. 2016. `https://perezdecastro.org/2016/ljndi-snabbwall-sidekick.html`. Last visited 2017-01-16.

[3] G. Antichi, S. Giordano, D. J. Miller, and A. W. Moore. Enabling Open-Source High Speed Network Monitoring on NetFPGA. In *Network Operations and Management Symposium*, pages 1029–1035. IEEE, 2012.

[4] Arista, A. Dacquay. Introduction to Managing EOS Devices – Troubleshooting. `https://eos.arista.com/introduction-to-managing-eos-devices-troubleshooting/`. Last visited 2017-01-15.

[5] K. Barone-Adesi, A. Wingo, D. Pino, and J. Muño. pflua. `https://github.com/Igalia/pflua`.

[6] T. Blechmann. Boost Lockfree. `https://www.threadingbuildingblocks.org/`. Last visited 2017-01-15.

[7] Bridges and Bridged Networks. *IEEE Std. 802.1q*, 2014.

[8] Bro Time Machine Source Code. FifoMem.cc. `https://github.com/bro/time-machine/blob/6f90b9f/src/FifoMem.cc`.

[9] L. Deri, A. Cardigliano, and F. Fusco. 10 Gbit Line Rate Packet-to-Disk Using n2disk. In *Computer Communications Workshops (INFOCOM WKSHPS)*, pages 441–446. IEEE, 2013.

[10] C. Desrochers. A Fast Lock-Free Queue for C++, 2013. `http://moodycamel.com/blog/2013/a-fast-lock-free-queue-for-c++`. Last visited 2016-07-01.

[11] C. Desrochers. A fast general purpose lock-free queue for C++, 2014. `http://moodycamel.com/blog/2014/a-fast-general-purpose-lock-free-queue-for-c++`. Last visited 2016-07-01.

[12] P. Emmerich. FlowScope. `https://github.com/emmericp/FlowScope`.

[13] P. Emmerich. libmoon. `https://github.com/libmoon/libmoon`.

[14] P. Emmerich. MoonGen. `https://github.com/emmericp/MoonGen`.

[15] P. Emmerich. MoonGen Scripts to Test FlowScope. `https://github.com/emmericp/flowscope-tests`.

[16] Endace Technology Limited. Introducing Endace DAG Cards. `https://www.endace.com/introducing-dag.pdf`, 2016. Last visited 2017-01-15.

[17] Facebook. A high-performance bounded concurrent queue that supports multiple producers, multiple consumers. `https://github.com/facebook/folly/blob/master/folly/MPMCQueue.h`. Last visited 2017-01-15.

[18] Facebook. A one producer and one consumer queue without locks. `https://github.com/facebook/folly/blob/master/folly/ProducerConsumerQueue.h`. Last visited 2017-01-15.

[19] Facebook. Folly: Facebook open-source library, 2017. `https://github.com/facebook/folly`. Last visited 2017-01-15.

[20] fmad engineering llc. Cost Effective Line Rate 10G Packet Capture to Disk. `http://fmad.io`, 2016. Last visited 2017-01-15.

[21] S. Gallenmüller, P. Emmerich, F. Wohlfart, D. Raumer, and G. Carle. Comparison of Frameworks for High-Performance Packet IO. In *Architectures for Networking and Communications Systems (ANCS)*, pages 29–38, Oakland, CA, 2015. ACM.

[22] Igalia. Benchmarking implementations of pflang, the pcap filter language. `https://github.com/Igalia/pflua-bench/blob/6787b55d/README.md`.

[23] Intel. Data Plane Development Kit. `http://dpdk.org`. Last visited 2017-01-15.

[24] Intel. Thread Building Blocks (TBB). `https://www.threadingbuildingblocks.org/`. Last visited 2017-01-15.

[25] Intel. *Intel 64 and IA-32 Architectures Optimization Reference Manual*, 2016.

[26] Intel 82599 10 GbE Controller Datasheet Rev. 2.76. Intel, 2012.

[27] Intel Ethernet Controller XL710 Datasheet Rev. 2.1. Intel, December 2014.

[28] Product Brief - Intel Ethernet Controller XL710 10/40 GbE. Intel, 2014.

[29] S. Kornexl, V. Paxson, H. Dreger, A. Feldmann, and R. Sommer. Building a Time Machine for Efficient Recording and Retrieval of High-Volume Network Traffic. In *Internet Measurement Conference 2005 (IMC'5)*, 2005.

[30] J. Lee, S. Lee, J. Lee, Y. Yi, and K. Park. Flosis: A highly scalable network flow capture system for fast retrieval and storage efficiency. In *2015 USENIX Annual Technical Conference (USENIX ATC 15)*, pages 445–457, Santa Clara, CA, 2015. USENIX Association.

[31] J. D. McCalpin. STREAM: Sustainable Memory Bandwidth in High Performance Computers. `https://www.cs.virginia.edu/stream/`. Last visited 2017-01-19.

[32] ntop. n2disk. `http://www.ntop.org/products/traffic-recording-replay/n2disk/`. Last visited 2017-01-15.

[33] ntop. nDPI – Open and Extensible LGPLv3 Deep Packet Inspection Library. `http://www.ntop.org/products/deep-packet-inspection/ndpi/`. Last visited 2017-01-16.

[34] ntop. PF_RING ZC (Zero Copy). `http://www.ntop.org/products/packet-capture/pf_ring/pf_ring-zc-zero-copy/`. Last visited 2017-01-15.

[35] M. Pall. LuaJIT. `http://luajit.org/`. Last visited 2017-01-15.

[36] P. Phaal and M. Lavine. sFlow Version 5, July 2004.

[37] M. Pudelko. Comparison of Queuing Data Structures for Traffic Analysers. *Bachelor's thesis*, 2016. Available at `https://www.net.in.tum.de/fileadmin/bibtex/publications/theses/pudelko-2016-queues.pdf`.

[38] L. Rizzo. netmap: A Novel Framework for Fast Packet I/O. In *USENIX Annual Technical Conference*, pages 101–112, 2012.

[39] D. Scholz, P. Emmerich, and G. Carle. Efficient Handling of Protocol Stacks for Dynamic Software Packet Processing. *Under submission to IFIP NETWORKING 2017*, 2017.

[40] H. Sutter. Lock-Free Code: A False Sense of Security. *Dr. Dobb's Journal*, Sept. 2008.

[41] H. Sutter. Writing Lock-Free Code: A Corrected Queue. *Dr. Dobb's Journal*, Sept. 2008.

[42] Tcpdump & libpcap. `http://www.tcpdump.org/`. Last visited 2017-01-16.

[43] M. Thompson, D. Farley, M. Barker, P. Gee, and A. Stewart. Disruptor: High performance alternative to bounded queues for exchanging data between concurrent threads. *White Paper*, May 2011.

[44] Transparent Hugepage Support. *Linux Kernel 4.4 Documentation*, 2016.