Technische Universität München

Department of Informatics

Bachelor's Thesis in Informatics

Comparison of Queuing Data Structures for Traffic Analysers

Maximilian Pudelko

TECHNISCHE UNIVERSITÄT MÜNCHEN
DEPARTMENT OF INFORMATICS

BACHELOR'S THESIS IN INFORMATICS

Comparison of Queuing Data Structures for Traffic Analysers

Vergleich von Warteschlangendatenstrukturen für Traffic-Analysers

Author	Maximilian Pudelko
Supervisor	Prof. Dr.-Ing. Georg Carle
Advisor	Paul Emmerich, Sebastian Gallenmüller
Date	July 15, 2016

Informatik VIII
Chair for Network Architectures and Services

Abstract

Modern 10-100 Gbit Ethernet adapters with steadily increasing data rates push the boundaries of the established work flows and make extensive on line traffic analysis a computational challenge. We show that existing data structures are not ideal for use in traffic analyzers because of different design goals. The root causes of the performance problems are analyzed and possible mitigations are shown. Based on that a set of requirements is worked out that a data structure should fulfill. A prototype named *Queue of Queues* (QQ) is developed following these guide lines and its performance is measured. It is demonstrated that *QQ* can handle data rates up to 150 Gbit/s, a factor 2-100 increase to existing ones, while keeping the latency within the 10 ms range.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Traffic analysis is an integral part of network administration and is used in many areas like flow monitoring, security enforcement, firewalls and intrusion detection systems (IDS). Researchers, programmers and administrators rely on the efficiency and reliability of traffic capturing and analysis tools like tcpdump and Wireshark. Usually these tools operate on a copy of the traffic stream, provided by the network stack of the operating system. However, with the common prevalence of 10 Gbit/s and upcoming 40 Gbit/s Ethernet devices, the internal stacks can not keep up with the high packet rates, resulting in missing packets and incomplete traces. But not only the Kernel is reaching its limits, other components which prior were fast enough, now become potential bottlenecks. Conventional hard disks are limited to around 125 MiB/s write bandwidth, a value easily exceeded by even two 1 Gbit/s NICs operating at full capacity. Another issue are the data structures used to organize and manage this many packets. Higher rates require more packets to be processed in the same time, while the clock frequency of CPUs stagnated since 2005 and Moore's Law is only fulfilled by fitting more cores on one chip. Spare computation power for analysis becomes increasingly scarce and the exploitation of modern multi-core architectures is a must.

As a result to these changes numerous software frameworks were developed, that bypass the stack and utilize new hardware features, promising to solve the problems. They are however not drop-in replacements and can only serve as a foundation for possible traffic analysis tool implementations by providing the basic building blocks. Products like the capture box SolarCapture and the software suite n2disk show that there is also commercial interest in reliable Gbit packet capturing. Both claim to be able to record at 10 Gbit/s wire rate without packet loss.

1.1 Goals of the thesis

The goal of this thesis is to compare existing data structures on their applicability as a core component of traffic analyzers. As they are found to be unsuitable, a set of requirements is worked out that such a structure has to fulfill to be ideal. Based on them a prototype named QQ is implemented. Lastly this prototype must be evaluated if it fulfills the set goals and how it can be utilized in a traffic analyzer.

1.2 Outline

The thesis starts in Chapter 2 with an example visualizing the general processing time problem with Gbit adapters and a usecase showcasing the possibilities a solution could provide.

Chapter 3 lists work related to the problem, which, in part, serves as the basis for the implementation.

In Chapter 4 several existing data structures of different origin are evaluated and the arising problems are discussed.

Further analysis on the problems is performed in Chapter 5 followed by the proposed solution that solves them.

Chapter 6 concentrates on the implementation of the novel data structure named QQ. Chapter 7 is dedicated to the evaluation of QQ to prove that it fulfills the set requirements in terms of throughput, latency and features. The chapter ends with an example that solves the previously constructed usecase.

Chapter 2

Motivation

This chapter provides an introduction to the challenge of packet analysis in the context of multi-Gbit/s link speeds and how a suitable data structure can mitigate it.

2.1 Limitation of Processing Power

Budget	2.4 GHz / 14.88 Mpps	= 161.3 cycles per packet
Costs	Packet retrieval	$\approx$ 50 cycles per packet[1]
	Memory management	$\approx$ 17 cycles per packet[1]
	L1D reference (hit)	$\geq$ 4 cycles [1]
	L2 reference (hit)	$\geq$ 11 cycles [1]
	L3 reference (hit)	$\approx$ 34 cycles [1]

Table 2.1: Processing time costs at 10 Gbit/s (14.88 Mpps) line rate

One of the larger problems is the severe lack of processing power. In normal setups with only a few million packets per seconds (Mpps) throughput, one CPU core may easily handle the entire traffic while analyzing it at the same time. With continuously growing packet rates and the introduction of multiple queues per NIC, the load demand increased dramatically. In Table 2.1 the costs of various packet processing operations are listed for a commodity Intel Xeon E5-2620 v3 CPU with a (non-turbo) clock rate of 2.4 GHz. The first two costs are measured averages per packet over the entire 10 Gbit/s stream handled by an Intel 10G X710 Adapter, while the last three are per operation resulting in the listed reference. The link was completely saturated at the maximum packet rate of 14.88 Mpps, resulting in a budget of 161.3 cycles per packet on the receiving core, which must be adhered to or packet loss will occur. This assumes the ideal case, where the core is not interrupted by the scheduler or occupied with other tasks. Each processing loop starts with the packet retrieval from the NIC. While this is batched in modern drivers

[1]DPDK, 64 byte packets, 64 batch size

to reduce overhead, an average time of 50 cycles per packet[1] is still spend in the driver. Due to the use of memory pools in DPDK, the objects storing the packets have to be marked as free by the caller, so they can be reused later. This took another 17 cycles per packet in our measurements. These fixed costs already take up 40% of the cycle budget and arise independent from any traffic analysis.

The remaining 60% or 94 cycles can then be used for only the most basic analysis: Even a packet counter will take an considerable amount of time. The value first has to be read into a register, which takes at least 4 cycles as the data will most likely remain in the L1 data cache due to frequent access. The increment operation is computational negligible, but the following store will take a similar amount of time as the read. Reading actual packet data is also not free. With technology like Intel's Data Direct I/O (DDIO) modern CPUs already optimize for fast access by writing the packets directly into the shared last level cache (LLC) of the CPU, skipping the slow RAM. Reads from the cache are generally much faster and will result in delay of around 34 cycles until the bytes of one cacheline[2] arrive. Analyzers filtering e.g. by TCP ports or tracking state, have to expect such costs.

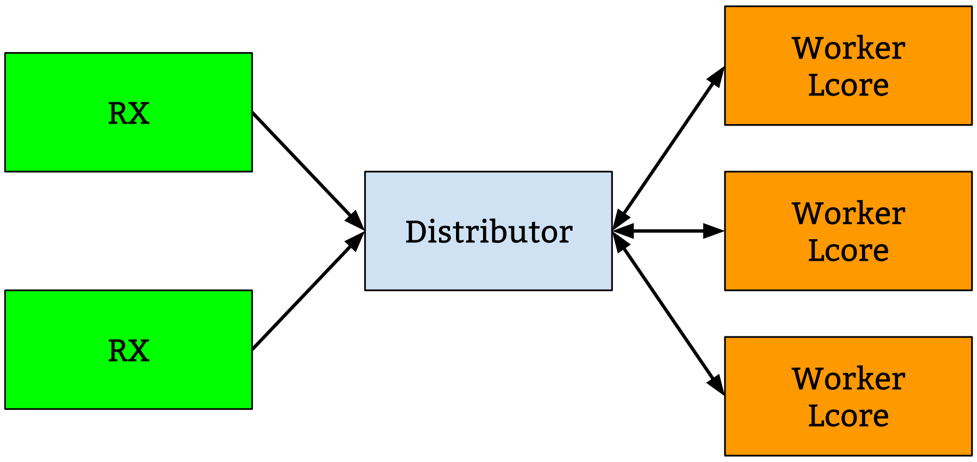

Figure 2.1: Load sharing scheme

While modern CPUs can coalescence many types of delays, it can be seen that overall there is very little time left to do extensive packet analysis on the same core the packets are received on. Therefore, they are usually distributed to other cores via queuing data structures, e.g. the DPDK ringbuffer, but generic data structures may also be used. In Figure 2.1 this is illustrated where a number of RX inputs, usually the RX-queues of NICs, insert packets into the distributor. Worker tasks bound to logical cores fetch and process packets in parallel, so that the load is shared among the CPU cores and meaningful analysis can be done.

[2] 64 byte on x86-64

2.2 Example Usecase - High Performance Filtering

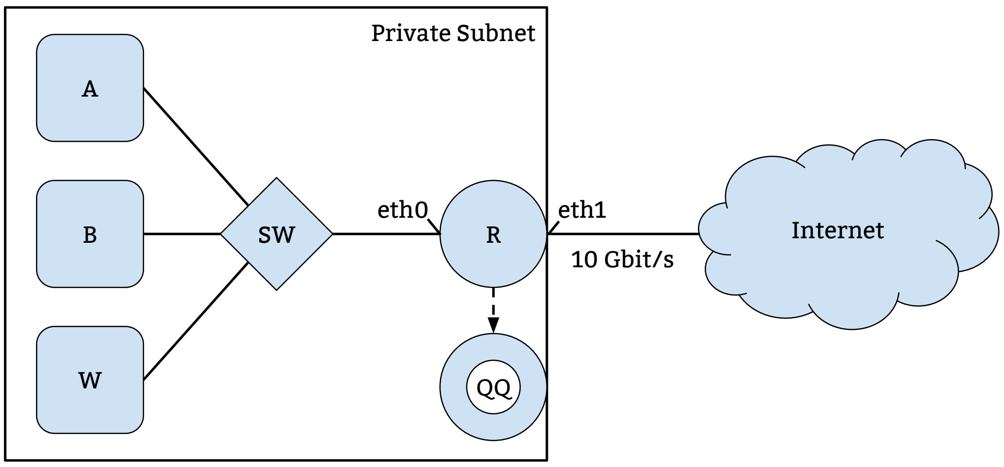

Figure 2.2: Example topology

In Figure 2.2 a possible use case is constructed in which such a queue can be used. A private subnet with a number of host is connected to the Internet via a gateway router *R* over a 10 Gbit/s link. Hosts *A* and *B* stand exemplary for the many clients connecting to the Internet causing large amounts of traffic. Additionally a webserver *W* is also located within the subnet and provides services to outside clients. The router forwards all traffic and also runs some kind of IDS to protect the clients and in particular the server from outside attacks.

In a usual setup the IDS would log the specific connection informations, if an attack or abnormal activity is detected. It is not feasible to capture an entire 10 Gbit/s link with standard tools like *TCPDump* in advance [2]. So all previous packets from the potential attacker, apart from the ones at the moment of detection, would be lost and not available for analysis.

With a suitable queuing data structure all traffic could be buffered temporarily and the analysis work load could be shared. Upon detection the specific analyzer then has access to past and future packets, can filter out interesting flows and store only these to disk. In Chapter 4 several existing data structures are evaluated if they satisfy the requirements of a traffic analyzer and Chapter 7.6 demonstrates how our new queue can be used in the presented a scenario.

Chapter 3

Related Work

This chapter presents a short overview over software and technologies related to packet processing and capturing on commodity hardware.

TCPDump/libpcap—One of the most commonly used tools for packet capture is *TCPDump* implemented on top of *libpcap*. It relies on the Kernel interface to the NIC and does not put it in exclusive mode like most other drivers do. This makes the tool more versatile as it can monitor traffic under normal operation, but also limits its performance compared to other frameworks. Table 3.1 lists the maximum capture rates achieved with *TCPDump* on Linux with a 10 Gbit/s Intel NIC as measured by Brown [2]. For his assessment he used a dedicated capture cube build with high end, but commodity hardware. Each test run consisted of unicast UDP packets and ended once *TCPDump* dropped a packet. The highest rate without failure marks the final result.

Packet size [Byte]	64	128	256	512	1024	1500
Rate [Gbit/s]	0.6	1.0	1.7	2.8	4.4	5.9

Table 3.1: TCPDump lossless capture rates on Linux

It is evident that these rates are far to slow for our purpose, but the widespread use of the pcap capture file format and the availability of processing tools already supporting it, make it the file format of choice for our developed data structure.

nCap/n2disk—In a publication from 2005 Deri [3] presents a novel approach for packet capture by removing the control of the NIC from the kernel. His design is split into two components: A new device driver managing the NIC exclusively with two circular buffers for send/received packets and an userspace library as the interface for regular C programs. In particular, the API is compatible to *libpcap* so that existing programs can benefit from the speedup without much rewriting. By mapping the NIC's memory directly into the address space of the process *nCap* can capture at maximum Gbit speed, while using less CPU cycles than a setup using the NAPI combined with *PF_RING*. Measurements on 10 Gbit/s or faster adapters are not available, but the tool was developed further by Deri and is now available as the commercial software product *n2disk* [4], which claims to be able to capture up to 10 Gbit/s.

Data Plane Development Kit/MoonGen—The DPDK project is a collection of libraries and drivers for fast packet processing tasks. It has explicit support for multi core architectures and is one of the most mature frameworks in terms of performance and reliability as shown by Emmerich et al [5]. The written drivers require the NICs to be bound to DPDK, which makes them invisible to the operating system even when not used in an experiment. The MoonGen project [6] started as a scriptable packet generator built on top of DPDK and accelerated by LuaJit, but developed into general purpose wrapper around the DPDK API. In [7] the same researchers demonstrated that MoonGen combines the ease of access and flexibility of scripting languages with the performance of compiled code.

The focus of the DPDK and *PF_RING* frameworks lies more on fast and precise packet transmission, than on packet capture or analysis. So they do not provide specialized data structures beyond basic building blocks like the tested ringbuffer *rte_ring* and static filter modules.

Chapter 4

Evaluation of Existing Queuing Data Structures and Problem Analysis

In Chapter 2 we showed that a concurrent data structure is required to distribute packets to different tasks. As data sharing is not a new problem, such structures already exist. In this chapter we present a comparison between a few general purpose ones and one from the packet processing framework DPDK by evaluating their features relevant to traffic analysis tasks. Subsequently, the found problems of them are analyzed. The newly developed data structure *QQ* is also included for reference.

4.1 Tested Data Structures

The following outlines the evaluated data structures. Implementation details, origin or remarks not fitting into the general problem categories are mentioned.

Moodycamel: ReaderWriterQueue and ConcurrentQueue—Both queues from Cameron [8, 9] are written for and in generic C++ and are completely lock-free. While the *Reader-WriterQueue* (RWQ) is limited to exactly one thread inserting and one thread removing elements from it compared to the *ConcurrentQueue* (CQ), both share the same underlying concept. The queue is a contiguous circular buffer of blocks, which can be allocated in advance to speed up inserts later. An index keeps track of the front and tail block respectively. Within each block there is an additional set of indices tracking its fill level. The performance benefit comes from the fact that each thread works on a block independent from others. The outer indices of the circular buffer are only moved once a block is full or empty and a thread moves on to the next block. One downside of this approach is, that for the CQ the dequeue order of elements, enqueued by different threads, is not necessarily the same as the enqueue order: I_1 inserts elements $\{A, B, C\}$ while I_2 inserts $\{1, 2, 3\}$. An reader might dequeue any interleaved combination like

$\{A, 1, B, 2, C, 3\}$ or $\{1, 2, 3, A, B, C\}$. Only the intra thread ordering is preserved, while no guarantees can be made about the inter thread order. The RWQ with facilitating a single inserter does not suffer from this problem.

Folly: ProducerConsumerQueue and MPMCQueue—The second set of queues origins from the *Facebook Open-source Library* (folly). Like the first the first presented data structures, there are two variations with different scopes. Similar to the RWQ folly provides the single header file *ProducerConsumerQueue* [10] (PCQ) with the identical limitations to 1:1 producer:consumer relationships. The *MultiProducerMultiConsumerQueue* [11] (MPMC) depends on the boost library and claims to be faster than any other queue present in both folly and Intel's *Threading Building Blocks* library. To be more robust under heavy contention, i.e. a high number of concurrent workers, the ticket dispenser uses an atomic increment instead of a Compare and Swap (CAS) loop to manage access. As CAS iterations fail when the value is modified by another thread at the same time, the loop becomes nondeterministic and potentially slow. Unlike the CQ it is linearizable and guarantees the same enqueue/dequeue order of elements.

Both queues are fixed in size, allocate all memory upfront and support optional blocking.

DPDK: rte_ring—DPDK [12] comes with its own lock-free ringbuffer implementation used internally and made available to programmers via the API. It allocates all memory up front and stores pointers to objects. In the default configuration *rte_ring* is single producer, single consumer queue, but at creation it can be changed to support more threads. We tested only the later setting. The number of slots is fixed once set and limited to powers of 2. Additionally we found that the maximum number of slots is $2^{27} = 134217728 \approx 134$ M [13], severely limiting its usefulness for our purpose: At a packet rate of 50 Mpps *rte_ring* can only hold $134\text{ M}/50\text{ Mpps} = 2.68$ sec worth of traffic until the ring is full, which does not fulfill our goal of buffering longer periods into the past. Enqueue calls will not overwrite old packets, but fail until slots are available again. This requires at least one worker always dequeues packets to make space for new ones, even when not processing or analyzing anything.

To store variable length packets without allocating every packet explicitly, DPDK provides a message buffer class (*rte_mbuf*) to be used in conjunction with memory pools (*rte_mempool*). While these provide high performance object creation routines, they are unsuitable for anything else than short term storage, as a single mempool is limited to the same restrictions as the ring. The message buffer also store several bytes of device specific flags and metainformation and so inhibit a constant memory overhead per packets. Bundling multiple ringbuffer together to reach the required storage capacity would be possible, but complicates usage and setup.

Due to their internal design with per-core caches, all *rte* data structures can not be used outside of the DPDK environment, in particular not by threads created with the pthreads library.

We included *rte_ring* despite these shortcomings as it is not clear if the slot limits are genuine design limitations or just precautionary limits set by the developers that could be increased easily. DPDK may be used anyway, so that the additional requirements do not always apply.

4.2 Tested Features

Throughput—Throughput measures the average sustainable rate of packets and is an important factor for a traffic analyzer. The bars in Figure 4.1 show the measured rates of each queue, depending on the used allocation method further explained in Section 4.3.2. The left y axis lists the packet rate in Mpps, while the right y axis shows the equivalent data rate, if the packets were transmitted over a NIC[1]. For the test four threads are inserting elements into a queue, while four other threads dequeue them as fast as possible. To only measure the theoretical limits, no other work is performed on the packets. With exception of *rte_ring* all queues are configured to allocate 16 GiB of memory. All packets are 64 byte large as this represents the worst case in packet processing [14]. As the RWQ and PCQ are inherently limited to one producer, one consumer constellations, the benchmark can not be run in the same fashion for them and is reduced to two threads. The results are therefore not directly comparable, but still included, as workarounds with one separate queue per NIC are thinkable.

Low latency—Depending on the implementation and features of a data structure, the minimum time it takes for a element to traverse the queue may differ significantly. A low value makes the structure eligible for use in real time (critical) environments, e.g., audio buffering or packet forwarding. In the context of traffic analysis latency only plays a minor role as small delays in the millisecond range can be tolerated and are offset by larger buffers. All tested structures except QQ are optimized for this.

Variable length objects and memory preallocation—As packets come in different lengths it must be possible to store them efficiently, while at the same time using only the preallocated memory for performance reasons(Section 4.3.2). *rte_ring* and *QQ* solve this problem by using special container data types, while the generic queues can not solve this: Either pointers to individual allocations are stored, or the queues use fixed size containers, which can be either too large or too small.

[1] 10 Gbit/s = 14.88 Mpps @ 64 byte packets

Growability—Growability describes the possibility to adjust the total number of slots of the queue after creation. This can be beneficial when the load profile changes over time and memory usage should be minimized. While not technical impossible, none of the queues that can store variable length objects and preallocate memory are resizeable. *rte_ring* is different to QQ in having a fixed number of slots, where in QQ the maximum number of packets is limited implicitly through the memory capacity.

Special functions—Special functions describe features going beyond the scope of a queuing data structure that could aid in the implementation of a traffic analyzer. Section 4.3.3 describes them and their use in greater detail.

Multi-producer/-consumer—Apart from the RWQ and PCQ all queues can be used from more than two threads at the same time. This is an important property when e.g. traffic from multiple NIC should be bundled or packets are distributed to more than one core. In general single producer queues can be faster than multi-producer ones in the 1:1 case, as they can ignore certain edge cases in their model.

Bulk operations—To reduce the overhead of function calls, synchronization and general housekeeping, bulking packets together into one single operation is a common technique in networking. All provide such a mechanism, except the RWQ and the PCQ . While QQ takes this principle even further by incorporating it deeply into its design, it has no explicit bulk call for multiple packets.

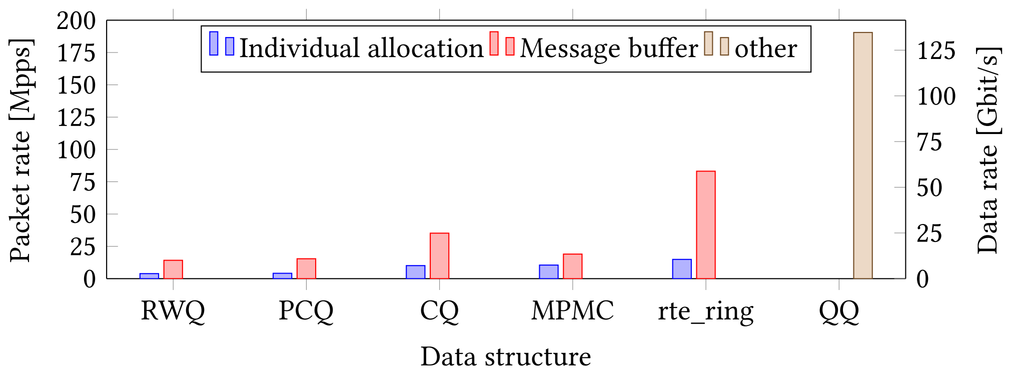

Figure 4.1: Throughput comparison with 64 byte packets

[2]Configurable, only the multi-producer/-consumer version was tested

Feature/Data structure	ReaderWriterQueue [8]	ConcurrentQueue [9]	ProducerConsumerQueue [10]	MPMCQueue [11]	rte_ring + rte_mbuf [12]	QQ
Variable length objects and memory preallocation	X	X	X	X	(✓)	✓
Growable	✓	✓	X	X	X	X
Special functions	X	X	X	X	X	✓
Multi-producer/-consumer support	X	✓	X	✓	✓^{2}	✓
Low latency	✓	✓	✓	✓	✓	X
Bulk operations	X	✓	X	X	✓	(✓)

Table 4.1: Feature matrix

4.3 Detected Problems

This section lists the detected problems with the tested data structures and how these problems limit their usage for packet processing. They are grouped by categories as some problems are often universal and not limited to one specific implementation.

4.3.1 Latency Overoptimization

All tested data structures optimize heavily for short operational latency. They achieve this by employing lock-free programming techniques with atomic operations or CAS-Sequences. While this keeps the delay of a single call very short, the small overhead of these operations accumulates over millions of packets per second, limiting the throughput severely. One should also note that one operation does not always result in one processed packet. Lock-free operations can fail when two cores access the same data at the exact same moment. Then only one of the calls succeeds while the other has to try again. It is obvious that this will occur more often with an increasing number of threads. In Figure 4.1 therefore only the rate of actually stored packets, i.e., successful calls, is listed. These are often lower than the benchmarks provided by the vendors, but as we are only interested in the packet processing rate, it is conclusive to only count real results.

The different design goals are reflected by the measured results. Only *rte_ring* can reach data rates higher than 40 Gbit/s and in direct comparison *QQ*, which does not optimize for latency, outperforms all other structures by a factor of 2.2 to 100.

4.3.2 Memory Management

When dealing with variable length objects like packets, we have seen two approaches of storing them in memory:

Message buffer—An extra class provides fixed size chunks of memory which encapsulate the actual packet data and store some meta information about it. Due to their homogeneity they can be allocated in bulk and beforehand, which both increase performance. Of the tested data structures only *rte_ring* in combination with *rte_mempools* uses preallocated message buffers. The generic queues do not provide such features. To still get comparable data we simulated message buffers by using fixed sized structs larger than the packet. Generally message buffers have an inherent space overhead, as short packets are padded and packets longer than 128 byte get segmented over multiple buffers.

Individual allocation—The naive way is allocating every single packet individually with the correct length. The performance of the methods depends nearly entirely on the underlying implementation of the `malloc` function. For the measurement we used the implementation of the GNU C Library 2.19, which results in the following problems:

- The `malloc` function is thread-safe [15], which is ensured by locks or other mutal exclusion mechanisms. As the function is called for every packet by every inserter, the contention for this critical code path becomes very high and creates a bottleneck. In Figure 4.1 the performance penalty of using malloc compared to memory pools becomes evident. For the same setup and ring size, only 20% of the throughput compared memory pools can be reached.
- Identical to the allocation problem, is the deallocation phase on the consumer side. Reclaiming memory from processed packets is done via the `free` function, which is synchronized in the same way as malloc and thus is limited in the same way. Reusing memory directly to prevent any diversion over libraries is nearly impossible as every chunk has been allocated with the specific length of the previously stored packet.
- Individual allocation creates small packets fragments distributed over the heap. In particular they are not guaranteed to be laid out continuously or ordered by arrival time, so that memory accesses cannot benefit from the principle of locality of reference. The CPU prefetcher cannot predict where the next packet is stored and consequently not preload it, leading to massive performance penalties [16].

It should be noted that this is a known problem and several `malloc` implementations specifically for use in multi threaded applications exist, but comparing these is out of the scope of this thesis.

4.3.3 Lack of Special Features

Implementation efforts of traffic analyzers can benefit from a certain set of functions if provided by the underlying data structure.

Random access—While a queue can usually only be accessed at either the front or back, a traffic analyzer must have access to packets at any position. All tested queues forbid this as it requires additional synchronization and is beyond the scope of a normal queue. Because of its great use, the QQ implementation must permit this mode of access, but it may be limited to read only operations. As an analyzer should only monitor and not modify traffic, this is sufficient.

Roles—Similar to the access, the existing queues lack the distinction between different roles. Only the obvious roles of the producer *inserting* and the consumer *removing* elements are provided. As the analyzer needs to look at the flow while not discarding it, we require a third function named *peek*. It must behave equivalent to the tail by moving forward up to the head index, but it does not remove packets from the queue, so they are still available to a later dequeue call.

Additionally we want to control how an overflow situation where more packets are inserted than dequeued is handled. In the tested data structures we have seen either a total block on inserts until slots are available again or an overwrite of the oldest element. Both can be desired, for instance when an event has been detected and all stored packets should be written to disk. Then new packets must only be inserted if space is available or else valuable information could get overwritten.

Context coherent storing—As real traffic is not uniform in throughput, especially if combined from different sources, packets do not get distributed equally over the queue. It can help analysis if there is a coherence between the position in the queue and the arrival time of the packet. One can then calculate the estimated position of a packet by knowing its arrival time, without doing a linear search. Queries for events that happened 30 seconds before then become possible. Obviously none of the tested queues support this.

Persisted storage—Due to performance reasons the packets are only kept in RAM until deemed interesting by the analyzer. But then it is crucial to have the option to move packets into persistent storage for later offline analysis. While this is not implemented by any queue, it could be added via a separate library supporting this functionality, as it is done in our design.

Chapter 5

Proposed Solution

In Chapter 4.3 we analyzed the problems that were found with the tested queues. Based on the results we have worked out a set of requirements that a data structure must fulfill to be used in a traffic analyzer. With them the outline of our prototype named *Queue of Queues* (QQ) is set up in the last section.

5.1 Requirements

We have seen that data structures are often optimized for one specific property, i.e. latency, and tend to ignore other key features that are important for traffic analyzers. With the following requirements a frame for the later evaluation of QQ is set.

QQ must...

(R1) ...be able to handle the throughput of a single 10 Gbit/s NIC and a total of 40 Gbit/s combined.

(R2) ...be able to store several million packets or at least 16 GiB efficiently.

(R3) ...provide the special functions, enumerated in 4.3.3, needed to implement traffic analyzers.

(R4) ...scale on multi core architectures by at least not decreasing performance with increasing number of workers.

5.2 Structure

To accomplish the set goals, we developed the structure shown in Figure 5.1, partly incorporating features that have been proven functional in other designs.

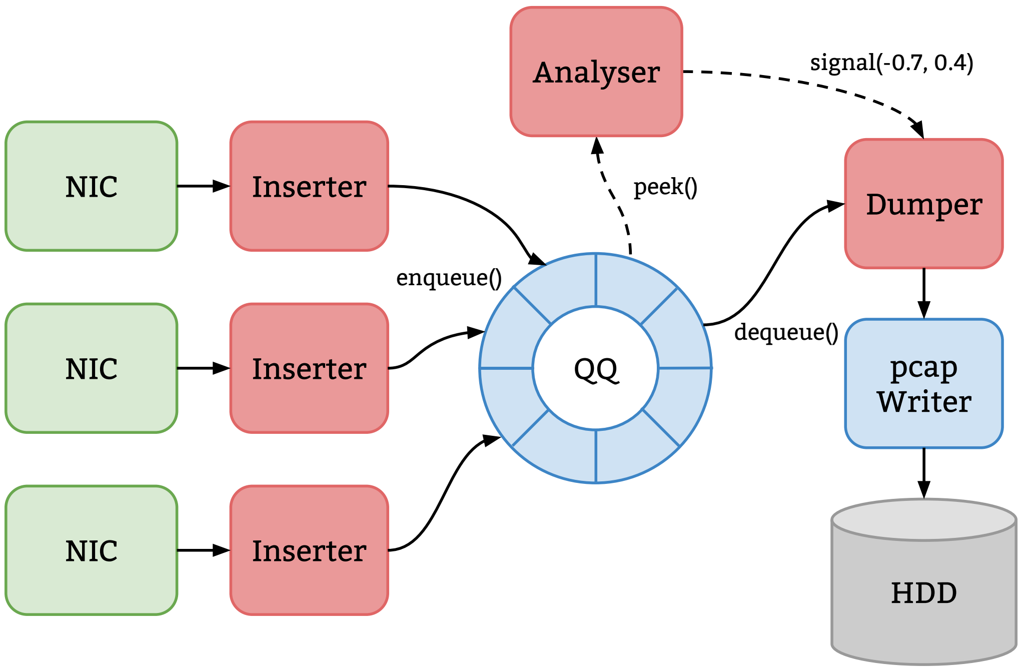

Figure 5.1: QQ design overview

The red boxes represent tasks and are run in concurrent threads. As *QQ* is not part of a specific packet processing framework, NICs are not aware of it and can not store packets in *QQ* directly. The inserter tasks therefore fetch the packets from the respective driver in use and enqueue them into *QQ*. Analyzers can then obtain read only access to them via the aforementioned peek function. Upon detection of an event they signal the dump tasks the needed filter criteria like a time range, IP addresses or port numbers. Theses tasks in turn get the packets by calling the traditional dequeue function and can then move them to persistent storage. This is done via the pcap writer, a memory mapped writer implementation of the pcap storage format.

To distribute concurrent accesses efficiently we utilize a two layer model:

Outer Queue—An outer queue handles incoming calls, manages memory and defers accesses to the inner queues. In particular it does not store packets itself. In Chapter 4 we have shown that memory management is a crucial component in an efficient design, therefore will the needed memory is be allocated at queue initialization time and further allocations are minimized. To prevent fragmentation and to benefit from locality of reference, the space for the packets is allocated as one continuous block. To prevent data races, all functions modifying shared state are secured by a queue wide mutex. Normally having one global lock would lead to a very high contention around it, resulting in very poor performance with an increasing number of threads. However, with the concept of inner queues the number of function calls actually requiring that mutex can be reduced drastically, so that this overhead becomes negligible.

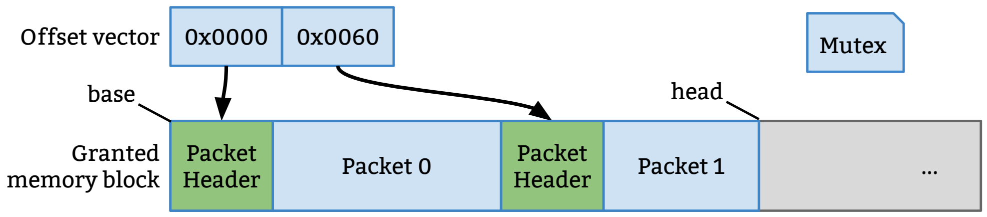

Figure 5.2: Inner queue structure

Inner Queue—Inner queues are set up at initialization time and behave like independent, equal sized queues. They do not allocate space themselves, but are assigned a range from the large block by the outer queue. Unlike in the implementation of the CQ, inner queues are not bound to a fixed task, but can be assigned to any role and to any member of each role. In particular it is not necessary to know the number of inserter tasks in advance. Further we exploit that the mutex guarding an inner queue only has to be locked once by a task, then it can operate independently and without further checks on that queue. In contrast to models that rely on atomic operations, not every insert or dequeue is synchronized with all others immediately. Thus the higher initial cost of locking the mutex is amortized by the large amount of following operations with no additional costs. Once the mutex is unlocked all changes become visible to other threads. The inner queue will serialize each packet into the assigned memory block. A prepended header keeps auxiliary meta data and marks the start of an packet. To enable constant time access, a reference to each header is stored in an offset vector as shown in Figure 5.2. New packets are copied at the head index until all space from range is exhausted and a new inner queue must be requested. The current queue is then unlocked and can be dequeued by an analyzer or dump task. Once the packets are no longer needed, the queue can be reset by setting the head index to the base index, making the clear a $\mathcal{O}(1)$ operation.

Chapter 6

Implementation

This chapter documents the implementation details of the newly developed QQ data structure, based on the analysis in Chapter 5. It provides an overview over all classes, how they relate to each other, what and why design decisions have been made.

The implementation language of choice is C++ as it provides powerful abstractions for container data types with the standard library. Writing these from scratch in pure C would take a considerable effort while providing little to no benefits compared to writing separate bindings for other languages. Where possible the use of static types was preferred over `void` pointers as they inhibit performance penalties and provide no strong type safety guarantees, which hinders analysis and is generally more error prone. Overall is QQ written as a generic, userspace data container with no dependency on external software libraries like boost and refrains from using lock-free programming techniques. While its main purpose is high speed packet processing and it is used in conjunction with the MoonGen and DPDK project in this thesis, it does not depend on these frameworks or even packet traffic at all.

6.1 `QQ::packet_header` Class

The `QQ::packet_header` class serves as a header prepended to the stored packets. Its general purpose is to hold additional meta information about an L2 frame that is not contained in the packet itself. To retain binary compatibility with plain C and LuaJIT the structural complexity is limited to the semantics of a *Plain Old Data* (POD) struct [17]. As the C++ standard library does not provide a data type for in-place variable length objects, we decided to use an incomplete array type from C99, called *Flexible Array Member* (FAM) [18], for the frame data. Listing 6.1 shows the struct definition with `len` storing the length of a packet and `data` holding the pseudo pointer to the frame. When this struct is allocated with enough space to hold `sizeof(packet_header) +`

`len * sizeof(uint8_t)` bytes, operations on the `data` member must behave as if the member were a normal array of type `uint8_t` [18]. The resulting binary layout in memory is illustrated in Figure 6.1 with the bitfields `timestamp` and `vlan` being packed into the first eight byte to reduce the total size of the header and therefore overhead per packet to 16 bytes. Note that the `data` field conveniently starts at the 10th byte. This aligns the start of an L3 PDU after the 14 byte long Ethernet header, marked in blue, to an 8 byte boundary ($(10 + 14) \mod 8 == 0$), which reduces the latency of read and write operations on the x86-64 architecture [1] and is required on others like ARM. The timestamp field holds the arrival time of the packet in microsecond resolution with a range of 8926 years. The value itself has to be provided by the user and can either be from a software clock or from the NIC if it supports hardware timestamps. With the extra `vlan` field it is possible to store the *VLAN identifier* (VID) of the optional 802.1Q header out of band. As this tag is often striped by networking hardware and only provided as meta information, it would be otherwise lost.

```
struct packet_header {
    uint64_t timestamp:8;
    uint64_t vlan:12;
    uint16_t len;
    uint8_t data[];
};
```

Listing 6.1: struct definition

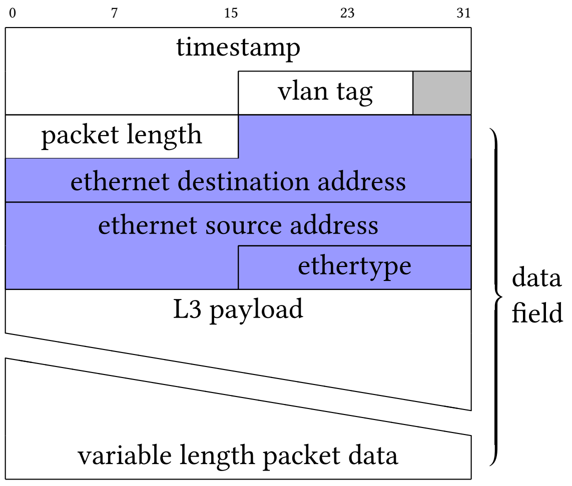

Figure 6.1: `QQ::packet_header` layout with an Ethernet frame (header marked blue) stored

6.2 `QQ::Storage` Class

The `QQ::Storage` class is the implementation of the inner queue, as laid out in our design structure.

To store the packets inside the memory block assigned by the outer queue, the *placement new* functionality is used. It allows the construction of objects at a specified location, while leaving lifetime management to the implementer. With an index the current fill level is check before every insert, so that no out of bounds writes occur. As constant time access to all packets is a benefit, a pointer to each packet header is pushed upon an offset vector. In addition to the amortized $\mathcal{O}(1)$ append of vectors, some space is reserved at initialization time, so that in most situations no expensive reallocation is needed. In general the `Storage` class behaves very similar to a `std::vector` and implements common functions like the `[]`-operator, pop_back(), clear() and iterators in the expected manner. To preserve the 8 byte alignment worked out in Section 6.1, an optional padding is inserted between two packets, depending on packet lengths. A `std::mutex` in combination with `std::unique_lock` is used for synchronization purposes.

6.2.1 Maximum Hold Time

To account for situations where one writer does have significantly less packets per second to handle compared to its co-threads, we implemented a maximum hold time check. As its write rate is lower than others, it takes more time to fill up one `Storage`, resulting in older packets being stored between much newer ones. There is also the possibility of an lock occurring if this slow writer still holds a `Storage` when the rest of the writers have moved forward and the readers are waiting for this thread. With this check the packets are distributed more evenly on the outer queue, even at lower rates.

Each `Storage` has a field to store the last time it was given out to a task by the outer queue. On each store call this value is checked against the current time. If more time has passed than the timeout allows, the store operation is denied which signals the task that this Storage element should be released and returned to the outer queue again. The implementation uses the `__rdtsc()` [1] compiler intrinsic which returns the current timestamp counter of the CPU. Modern CPUs have a synchronized counter across all cores and increment it at a fixed rate called *reference cycles per second*, independent from the actual, variable CPU frequency [1]. This makes the counter suitable for our check where only the delta between two timepoints and not a wall time is needed.

6.3 `QQ::Pointer` Class

As described in the Section 6.2, `Storages` require some additional steps to function correctly. To simplify and ensure correct usage for programmers, the `QQ::Pointer` class takes care of most of these tasks by wrapping around a `Storage`. In particular it sets the acquisition time correctly and ensures that the lock of the inner queue as actually hold before operations on it take place. Should a `Storage` be unlocked by either going out of scope or by explicitly calling release(), it guarantees that the mutex is unlocked and prevents later dead locks. All other function calls are simply forwarded to the underlying `Storage`, masquerading this wrapper to tasks.

6.4 QQ::QQ Class

The QQ class serves as the outer queue and resource manager of the inner queues in our design. It also handles the configuration parameters set at compilation and run time. Once a QQ object is instantiated it allocates the total memory needed in one contiguous block and backed by huge physical pages. These are larger than the standard memory pages of 4 KiB and therefore need less entries in the Translation lookaside buffer (TLB) of the memory management unit (MMU). In addition to the traditionally used (file-)system for hugepages, the Linux Kernel provides the concept transparent hugepages (THP) [19] for userspace programs, albeit limited to 2 MiB sized pages. Because of the simplicity in usage and silent fallback in case of lacking hardware support, we decided for THP. Listing 6.2 shows an excerpt from the code of the constructor. The first system call mmap returns a pointer the allocated block of memory in the virtual address space. It is marked readable and writeable as we want to store the packets directly within it. Line 5 sets the mapping as private which is an requirement of the kernel for the usage THP and prevents an additional mapping into the address space of another process. Subsequently, all parallelism efforts are based on threads as they share the same address space. With the second system call in Line 9 the pages are marked eligible for THP via the MADV_HUGEPAGE flag. Depending on the system's configuration the merging of the pages will either be performed directly, deferred to later or skipped silently. Section 7 summarizes the relevant kernel parameter. Communication between different tasks happens via condition variables.

```
1  const size_t qq_capacity = pages_per_bucket * num_buckets * huge_page_size;
2  if ((backend_ = (uint8_t*) mmap(NULL,
3                                  qq_capticity,
4                                  PROT_READ | PROT_WRITE,
5                                  MAP_ANONYMOUS | MAP_PRIVATE, -1, 0)
6      ) == MAP_FAILED) {
7      handle_error("mmap");
8  }
9  if (madvise(backend_, qq_capticity, MADV_HUGEPAGE)) {
10     handle_error("madvise");
11 }
12 for (unsigned int i = 0; i < num_buckets; ++i) {
13     storage_in_use.emplace_back(new Storage<pages_per_bucket*huge_page_size>(
14         backend_ + i * pages_per_bucket * huge_page_size));
15 }
```

Listing 6.2: Memory allocation in the constructor

The loop in Line 12-15 then creates the Storage elements and stores references of them into a vector. Each element is granted an even piece of the memory block determined by an offset and length.

The analysis in Chapter 5 showed that having multiple different roles is beneficial for certain packet analysis tasks. We wanted the possibility to look at the incoming traffic while not discarding it doing so. This is implemented in the additional `peek` position index on the ringbuffer. It can be moved independently from the other ringbuffer pointers but shares some semantics with the tail pointer. Under normal operation it is positioned behind the head where the new packets are inserted and before the tail where they get dequeued again. Unlike its POSIX counterpart it does not return the same element repeatedly until new ones arrive, but moves forward up to the head, giving a preview to each element exactly once.

6.4.1 Locking Strategy

Handling multiple locks requires great care to prevent deadlocks or bugs. Listing 6.3 shows the `dequeue` function with the locking sequence developed. First the global mutex of the outer queue is acquired in Line 2 to grant exclusive access to the shared variables. Should the mutex be held by other threads at the time, the caller will block until the other tasks finish and the mutex is released. Line 3 and 4 check several preconditions with the use of condition variables, such as the current priority value and if the queue even contains enough elements to be dequeued at the moment. Should these conditions not be fulfilled, the thread will wait until it is notified again, releasing the global mutex while doing so. Upon notification from, e.g., an inserter thread, it will reacquire the lock atomically so that the checked conditions are guaranteed to remain true thereafter. In Line 6 the `Storage` at the current tail index is fetched and its mutex is locked. This has to happen before advancing the index in Line 7, as an index of a ringbuffer always points to the next, not the current, element. Manually unlocking the global mutex in Line 10 before notifying waiting threads is common praxis to prevent one unnecessary wakeup and the immediately following stall: A waiting writer thread would be notified that space is now available and would try to acquire the global mutex. But at this point it is still held by the reader thread sending the notification, thus the writer would be put on hold again until the reader leaves the dequeue function.

```
1  Ptr<pages_per_bucket> dequeue(const uint8_t call_priority = 1) {
2      std::unique_lock<std::mutex> lk(mutex_);
3      cv_prio.wait(lk, [&]{ return check_priority_no_lock(call_priority); });
4      non_empty.wait(lk, [&]{ return distance(tail, head) > guard_interval; });
5      Ptr<pages_per_bucket> p{*tail};
6      tail = wrap(tail + 1);
7      ++dequeue_call_counter;
8      lk.unlock();
9      not_full.notify_one();
10     return p;
11 }
```

Listing 6.3: Dequeue function

Enqueue succeeds	Task priority compared to QQ ≥	=
Queue not full	✓	✓
Queue full	✓	X

Table 6.1: Priority check matrix for enqueue calls

6.4.2 Priority System

Generally QQ handles overload situations where more packets are enqueued than dequeued by discarding the oldest packets. In certain situations it can be required to deviate from this behavior, e.g., when a flow has been detected and it is more important to dump all stored packets than to not drop new ones. Therefore, *QQ* has a global priority level ranging from 0 (highest) to 255 (lowest) that is checked before unhandled packets are overwritten by an enqueue call. Conversely tasks have an equivalent priority value set as seen fit by the user, which they append to their calls to QQ. Table 6.1 lists the possible constellations in a priority matrix. Should the call priority be high enough for the current level or the queue is currently not full, then the enqueue will continue immediately, as the call would either not overwrite elements or is specifically advised to do so. Only if the queue is both full and the priority is too low, the thread will block until either condition changes. With this it becomes possible to assign inputs different importance. It should be noted that this scheme is not fair and resource starvation of low priority tasks is possible.

6.4.3 Guard Interval

To reduce unnecessary contention for the outer queue mutex, a guard interval is introduced. It prevents the tail index from being pushed closer to the head index of the ringbuffer than a specified distance. This is necessary since inserter threads do not notify the queue once they release a specific `Storage` element. It is therefore impossible to decide if an element in the window $[head - n, head]$ can be locked immediately or if the `Ptr` creation in Line 9 of the dequeue function in Listing 6.3 would block because a writer is still inserting packets. As this block would occur while still holding the global queue mutex, all other threads would also be prevented from getting new containers, thus resulting in an overall performance degradation. With the guard interval in place, reader threads are prevented from trying to lock such potentially still in use elements.

In Figure 6.2 this concept is illustrated with four writers and an equivalent guard distance. Despite being already filled and unlocked by a writer, `Storage` n-3 can not yet be dequeued to a reader thread as this fact is neither known by the queue, nor guaranteed. In particular element n-2 is not yet ready and would block all operation until this container is filled and unlocked. This check is implemented in Line 5 of Listing 6.3

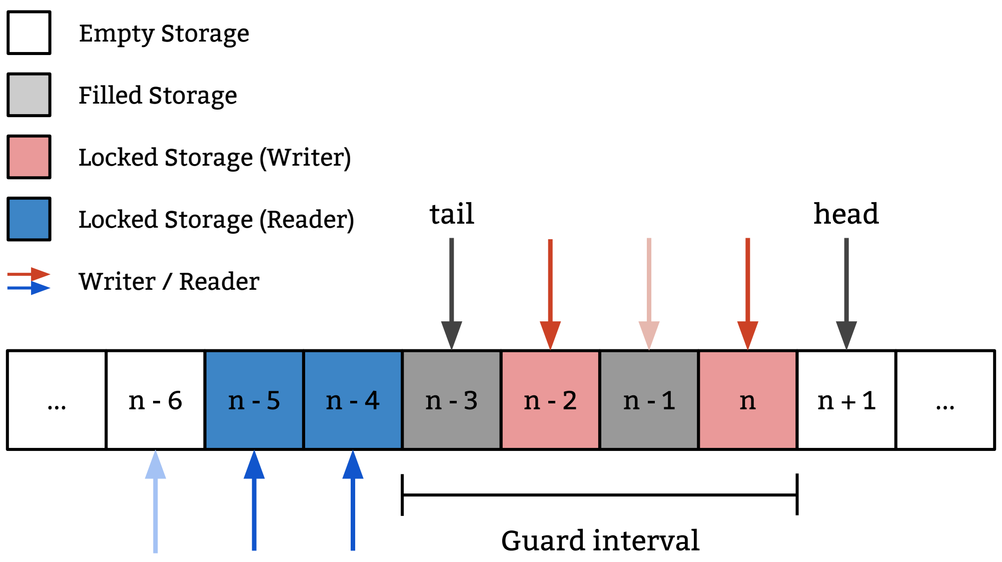

Figure 6.2: Guard interval preventing the tail from being moved further ahead

where the current distance of tail to head is checked against the guard interval. Should the test fail the queue mutex is unlocked and the thread suspended until notification from, e.g., the enqueue function. Contrary to situation before all other threads may still operate on the outer queue as the mutex is using condition variables.

Chapter 7.5 evaluates the impact of varying guard interval lengths on factors like throughput and latency and which trade offs can be made.

6.5 `QQ::pcap_writer` Class

Although *QQ* has a large capacity, it can be of interest to save certain packets for later offline analysis. The developed `QQ::pcap_writer` class provides an interface to store packets to persistent storage in the widely used pcap[1] file format. To achieve high peak performance and to not block queue operations, the implementation relies on the efficiency of file-backed memory mappings into the virtual address space of processes. Unlike with file handles the packets are not written via `write` calls, but copied to the appropriate memory areas. The kernel deferrers the actual, significantly slower, write-out to disk to a later time point, asynchronous to the packet analysis. Until then the packets are buffered in free physical memory. This makes it possible to handle short bursts of packets with much higher rates than the actual disk bandwidth.

As memory mappings are static in size and the final length of a capture is not known

[1] `https://wiki.wireshark.org/Development/LibpcapFileFormat`

at creation, the file has to grow dynamically. To keep track of the current fill level and to not write beyond the page boundary, a variable counts the total written bytes. Once the current capacity N_{old} has been reached, the file is grown and remapped according to the following formula:

$$N_{new} = \lceil K * N_{old} \rceil^{4\,\text{KiB}} \tag{6.1}$$

The calculated new size is rounded up to the next full 4 KiB as memory maps always have a size multiple of whole pages. While using less would still work, it would waste some space as the last partial page would be zero filled [20]. With this exponential growth the amount of resizes and resulting remaps is minimized, which can be expensive because they may cause data to be copied to a new, large enough, memory area. We chose a growth factor of $K = 2$ over the other commonly used value $K = 1.5$ as it further decreases the number of resizes by growing faster, while still using an conservative amount of space[2]. Together with the on-demand load of pages on access, this reduces the costs of stores to an amortized constant factor. As it is expected to store several hundred MiB of data, starting with a size of 1 Byte would only lead to countless unnecessary resizes at the beginning. The pcap_writer therefore defaults to a initial size of 256 MiB, but any hint can be specified in the constructor. When a capture file is closed, it is truncated to the actual size as reported by the written bytes counter. Opening, reading or appending to existing capture files is not supported.

6.6 LuaJit/MoonGen bindings

To simplify the testing procedure, *QQ* has been made available to MoonGen usersripts via the *Foreign Function Interface* (FFI) of the *Lua Just in Time* (LuaJit) compiler. In its functionality and usage it is similar to interface of the DPDK driver and therefore can be used in-place and besides of a NIC. Additionally, the `packet_header` is compatible with MoonGen's existing packet library, so that all programs building upon that will also function with packets from *QQ*. A receive loop has been implemented so a a NIC can be coupled to *QQ* with a single call.

[2]gcc uses $K = 2$ for its `std::vector` implementation

Chapter 7

Evaluation of QQ

This chapter evaluates if the developed data structure *QQ* fulfills the requirements from Chapter 5.1 and measures various performance metrics.

7.1 Setup

All tests were performed on the same host equipped with multiple, interconnected NICs to simulate more complex network setups. A detailed list of the used hardware is given in Table 7.1.

Hardware

Omanyte	
Mainboard	Supermicro, X10SRL-F
CPU	Intel Xeon CPU E5-2620 v3 @ 2.40GHz
Memory	DDR4, 32 GiB, 1866 MHz
NIC	2 x Intel LX710 40 GbE QSPF+
	2 x Intel X710 10G
	2 x Intel X540-AT2 10G
Storage	2 x Samsung SSD 840 EVO 250GB (RAID-0)

Table 7.1: Test host hardware

Software

Various tools and frameworks were used in this thesis. Table 7.2 lists them with their respective versions.

Name	Version
MoonGen	3b1a047
DPDK	16.04
pmu-tools	r105
gcc	6.1.1
Ubuntu	14.04.4 LTS
Linux Kernel	3.13.0-86-generic
STREAM benchmark [21]	5.10

Table 7.2: Used software

Unless otherwise noted, *QQ* was configured to use 26 GiB of the available 32 GiB RAM, with one `Storage` element being 4 MiB large.

Kernel Flags

The kernel provides several flags to influence the behavior of its modules. In particular the support for THP has to be enabled [19].

Flag	Value
`/sys/kernel/mm/transparent_hugepage/enabled`	madvise
`/sys/kernel/mm/transparent_hugepage/defrag`	always
`/sys/kernel/mm/transparent_hugepage/use_zero_page`	1

Table 7.3: Kernel flags

7.2 Benchmarks

In Chapter 4 we showed that most data structures struggle to keep up with high data rates. The first benchmark in Figure 7.1 therefore measures the total achievable throughput of *QQ* at different packet sizes and varying number of threads. It was performed within MoonGen via the developed *QQ* bindings for Lua. To exclude external influences, e.g., from interaction with NIC drivers or link speed limitations, all packets were generated in software via concurrent MoonGen tasks. Each inserter continuously acquires `Storages` from *QQ* and fills them with packets as fast as possible. To make the test more realistic, one additional task acts as a sink, dequeuing and counting the packets. Each run consists of a warm up phase of 15 seconds to fill the caches and ensure the allocated memory is at least touched once. Then the rate samples are taken over a period of 30 seconds at the sink. While we did not set a priority level to prevent packets from being overwritten by enqueue calls, we could not observe an overflow, even at the maximum recorded rate of 159 Gbit/s. This shows that the overhead of just dequeuing packets

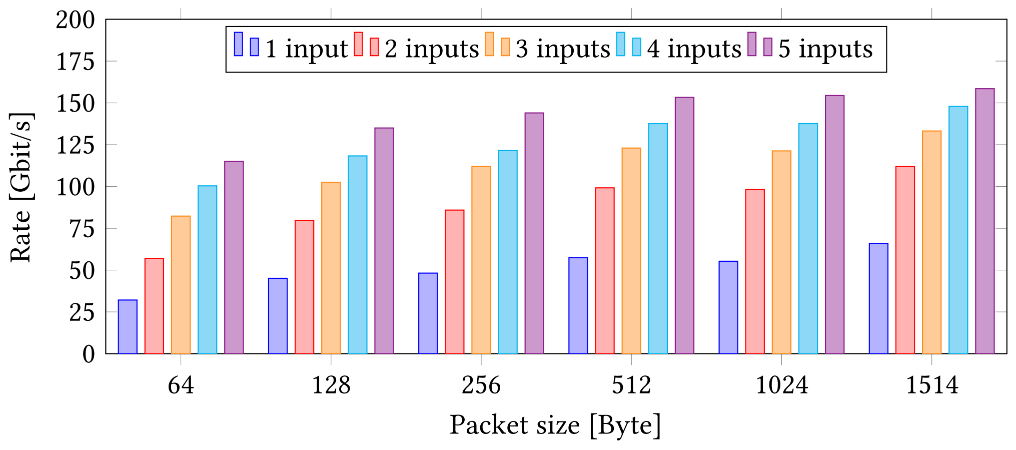

Figure 7.1: Theoretical *QQ* throughput within MoonGen

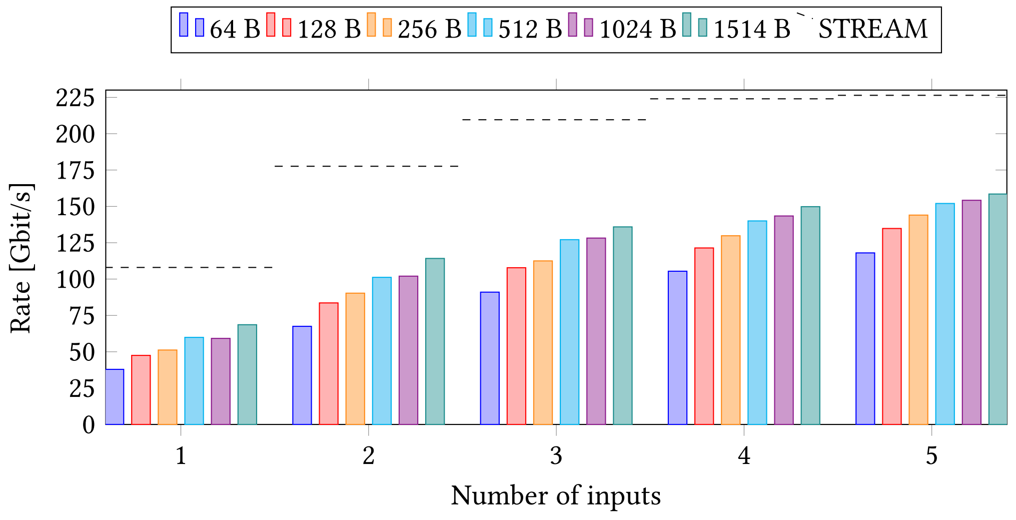

Figure 7.2: Theoretical *QQ* throughput compared to STREAM

from *QQ* is extremely small, as a single sink thread could handle the entire flow at $\sim 3\%$ CPU utilization.

The graph in Figure 7.1 shows the recorded rates in Gbit/s grouped by packet sizes. We started with the smallest sensible size of 64 bytes and doubled it until the MTU of a typical Ethernet link was reached at 1514 Byte[1]. The different bars in each group show the results for the number of input threads. As expected the rate does increases with the number of threads.

Figure 7.2 shows the same data ordered by the number of inputs. To set the results into a context, we included the copy measurement of the STREAM benchmark [21] as the

[1] 14 byte Ethernet header + 1500 byte L3 SDU, excluding the 4 byte FCS

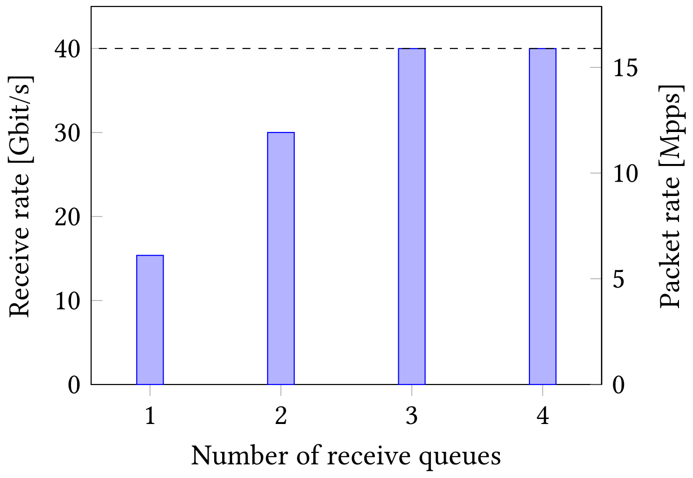

Figure 7.3: Multi-core (1.2 GHz) RSS with an LX710 40 GbE NIC and 256 byte packets

dotted line. It is often used to determine the total memory bandwidth of a system. I can be seen that different packet sizes have little impact on performance, while the overall maximum memory throughput is not reached at any packet size or number of threads. Runtime analysis of the code showed that 28% of the time in the store function is spend for the timeout check. A bulk store option could skip this check and potentially improve performance.

7.3 Multi-Queue Scaling for 40+ Gbit/s NICs

In Figure 7.1 and 7.2 it can be seen that the data rate for single inputs is severely limited compared to what is possible with multiple inputs. Speeding processes up by running them in parallel is a common approach also pursuit by networking hardware vendors by placing multiple queues with separate buffers on one NIC. These can be bound to different CPU cores exploiting modern multi-core architectures and solving the computational resource bottleneck. Techniques like Receive Side Scaling (RSS) distribute incoming packets over different queues, and consequently cores, by calculating a symmetric two-way hash function over the packet header. The properties of the hash functions are chosen so that the intra flow packet order of the stream is preserved. While packets of one flow are all assigned to the same queue, no assertions can be made about the order and timing of packets from distinct flows. The concept of splitting the stream up seems contrary to our goal of providing a single choke point for analysis, but RSS can help to get the packets from the NIC into *QQ* within the time critical loop. Every receive (RX) queue of the NIC is matched up with an inserter thread that stores the packets into `Storages` of the same *QQ*. Combined with our design of independent inner queues the resulting stream available to the analyzers is not further reordered.

Applications that can handle RSS traffic will not find problems with the packet order in *QQ* traffic.

Figure 7.3 demonstrates the performance improvements with one 40 Gbit/s NIC when receiving 128 UDP traffic flows from varying source ports generated with another LX710 NIC. The packets are 256 bytes large and the link capacity of 40 Gbit/s is fully saturated on the sending side. As the receive rate is largely dependent on the available processing power, we reduced the clock frequency on the receiving cores to 1.2 GHz to simulate higher link speeds and to show the scaling potential of having multiple receive queues. In the case of one and two RX queues the remaining processing time left after interaction with the DPDK driver is not enough to insert the packets into a `Storage` element of the *QQ*. The NIC therefore has to drop packets from its internal buffer as the inserter thread can not keep up with the traffic flow. With each additional RX queue and paired inserter thread the amount of dropped packets decreases until the full line rate is handled with three queues. Adding more queues beyond that does not influence performance, but would allow further scaling to 100 Gbit/s NICs as their queues are independent from each other and the speedup from adding queues is nearly linear. *QQ* itself provides the capacity and does not become a limiting factor until around 125 Gbit/s as discussed in Section 7.2.

7.4 Influence of Storage Sizes on Throughput

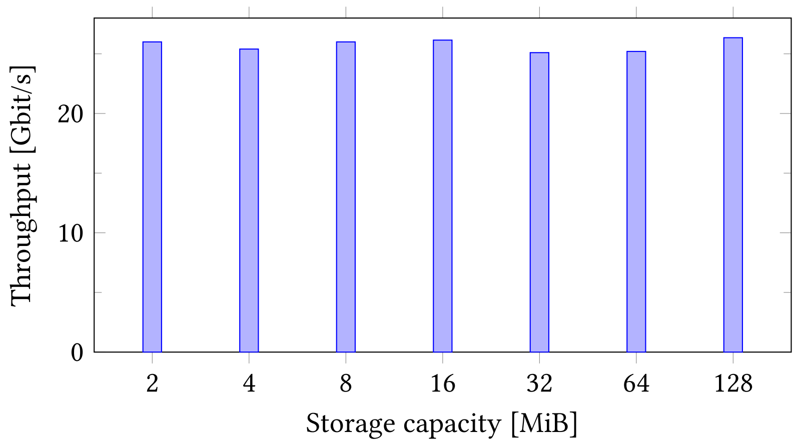

Figure 7.4: Single `Storage` throughput with 64 byte packets

In this section the influence of different `QQ::Storage` container size configurations on performance are discussed. Figure 7.4 shows the throughput of a single container, depending on the chosen size. The experiment is run with four inserters and one sink.

Overall the impact of different container size configurations is minimal, with only 0.95 Gbit/s difference between the fastest (128 MiB) and the slowest (4 MiB) configuration

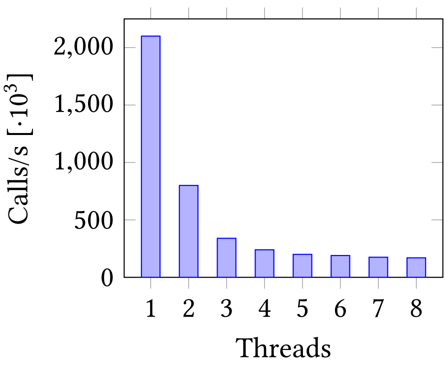

Figure 7.5: Outer queue call performance

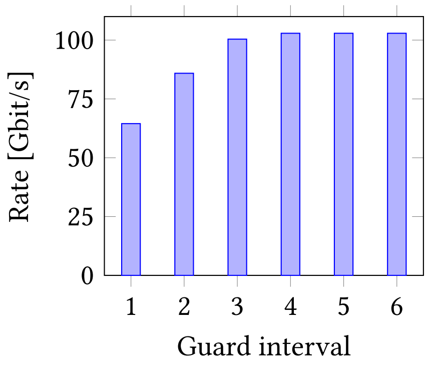

Figure 7.6: *QQ* throughput at different guard distances

and otherwise stable rates. We expected to see the smaller sizes to perform better due to caching effects, but measurements of the L2 and L3 cache hit rates showed no distinct trend.

In Section 5.2 we decided to use one queue wide mutex under the assumption that the concept of inner queues keeps the number of calls hitting the outer queue low. None the less we measure the call performance of the outer queue to ensure that it does not become a limiting factor. Figure 7.5 shows the total number of handled enqueue and dequeue calls per second. As common with mutexes, the rate decreases rapidly once the contention becomes higher with more threads. Despite the seemingly low result of 170000 calls/s at 8 threads, we can verify that this is enough for our purpose:

$$\frac{40\text{ Gbit/s}}{2\text{ MiB}} \cdot (1+1) = 4768\text{ calls/s}$$

Even with the smallest `Storage` size of 2 MiB and a rate goal of 40 Gbit/s, only 4768 call/s are required combined[2].

This concludes that our `Storage` data structure is suitable to be used as the inner queue.

7.5 Throughput - Latency Considerations

One of the decisions that allows us to achieve high speeds was the possibility to tolerate small latencies. In Chapter 6 we introduced a guard interval to prevent too heavy contention for `Storage` locks at the expense of an additional packet delay. This section expands on the usage of this interval and how it influences overall performance keys.

In Figure 7.6 the throughput of *QQ* at varying guard interval lengths is displayed. For this experiment 8 (4+4) inserter and analyzer threads are working on 64 byte packets.

[2] 2384 enqueue and dequeue calls each

It can be seen that the performance drops significantly if the guard distance becomes smaller than the number of inserters, down to the worst case of only one guard slot. As expected the peak rate is reached with a guard distance exactly equal to the number of inserters. We could observe that the inserter threads reach 100% CPU utilization at this point, which means that they are no longer blocked by the analyzers. Adding further guard slots does not lead to a higher rate and only increases the latency. So the optimal guard value is

$$g \geq \#\text{Inserters} \tag{7.1}$$

With the interval known, the minimum packet delay can be calculated:

$$d_{min} = \min\left(\frac{s}{r_{max}}, t\right) \cdot g \tag{7.2}$$

With s being the `Storage` size, r_{max} the maximum expected rate, t the set maximum hold time for a container and g the number of guard elements. Note that the hold timeout should not be set lower that $\frac{s}{r_{max}}$, as then a container can not be filled before the timeout occurs.

Given a fixed total rate goal one can adjust several parameters to reach it, while still maintaining a low latency. As shown in Section 7.4 the `Storage` size can be chosen nearly freely without losing much single input throughput. At constant rates a smaller `Storage` will be filled faster, queued earlier and analyzed with less delay. Therefore, the minimum container size of 2 MiB is preferred and yields the smallest latency, as shown in Table 7.4.

`Storage` size	2 MiB	4 MiB	32 MiB
10 Gbit/s & 2 inserters	3.335 ms	6.711 ms	53.69 ms
40 Gbit/s & 4 inserters	1.67 ms	3.335 ms	26.84 ms

Table 7.4: Minimum latencies at various configurations

An option unexplored in this thesis is inserter multiplexing: As seen in Section 7.3 one inserter can easily handle tens of Gbit/s. Multiple NICs with low link speeds or generally little traffic could be served by one inserter, leading to a lower delay according to Equation 7.2.

7.6 Exemplary Usage

This section shows how the use case constructed in Chapter 2.2 can be solved with *QQ*. As building a complete subnet with hundreds of clients is rather elaborate, we simulated the incoming traffic on eth1 with a LX710 NIC generating 128 UDP flows. A second LX710 serves as the interface of the router itself. All packets are inserted into a QQ, where an analyzer task looks for packets not matching the destination port 80 of the web server. With the dump task shown in Listing 7.1 we could capture up to 3.5 Gbit/s of matching traffic to disk.

```
function dumpTask(qq, path)
  local pcap_writer = pcapLib.create_pcap_writer(path)
  while mg.running() do
    -- Signaling with analyzer omitted, variable ip_match holds the source IP
    local storage = qq:dequeue() -- QQ API call
    for i=0, tonumber(storage:size())-1 do  -- loop over all packets
      local pkt = storage:getPacket(i)
      local udpPkt = pktLib.getUdpPacket(pkt) -- MoonGen's packet library
      if udpPkt.ip4:getSrc() == ip_match then
        pcap_writer:store(pkt:getTimestamp(), pkt:getLength(), pkt.data)
      end
    end
    storage:release()  -- explicit release at end of loop
  end
end
```

Listing 7.1: Dump task

Chapter 8

Conclusion

Comparison

We have shown that existing data structures are not entirely suitable for use in traffic analyzers, because of different design goals that result in an impaired performance. With QQ we set the focus on throughput instead of latency and introduced features that traffic analyzers could benefit from. That improved the throughput performance by a factor of 2.3-100, while also increasing the inherent latency from 1 us to 1-10 ms. With real NICs QQ could handle at least 80 Gbit/s synthetic traffic on our test system.

Possible Future Work

Although the prototype is completely implemented and rudimentary tested, certain areas remain unexplored.

JIT Filter Expressions—As shown in Chapter 7.6 the filtering process is solved by an analyzer function which discards uninteresting packets. In its current state the user is required to provide this filter function and therefore has to have knowledge about both Lua and the MoonGen API to write one. To simplify the usage of this core functionality of QQ and to satisfy the requirement of MoonGen to "be as flexible as possible" [7] it could be of interest to integrate a Berkeley Packet Filter (BPF) module. The BPF syntax consists of a relatively small instruction set which allows to write filter programs. E.g., the Linux kernel uses an extended form (eBPF) to allow user-supplied code to run in kernel space for, but not limited to, efficient network traffic filtering purposes. This interface is also used by *libpcap*, the library for the traffic capture tools *Tcpdump* and *Wireshark*. It should be investigated if it is possible to emit Lua code from such already used filter expressions. This code then could be JIT-compiled and used instead of the handwritten analyzer function, similar to how MoonGen customizes packets with LuaJIT.

Bibliography

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

[2] A. Brown, "I3: Maximizing packet capture performance," Wireshark Developer and User Conference, 2014.

[3] L. Deri, "nCap: Wire-speed packet capture and transmission," in *Workshop on End-to-End Monitoring Techniques and Services, 2005.* IEEE, 2005, pp. 47–55.

[4] ntop. n2disk. `http://www.ntop.org/products/traffic-recording-replay/n2disk/`. Last visited 2016-07-01.

[5] 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), 2015 ACM/IEEE Symposium on.* IEEE, 2015, pp. 29–38.

[6] P. Emmerich. Moongen. `https://github.com/emmericp/MoonGen`. Last visited 2016-07-12.

[7] P. Emmerich, S. Gallenmüller, D. Raumer, F. Wohlfart, and G. Carle, "MoonGen: A Scriptable High-Speed Packet Generator," in *Internet Measurement Conference 2015 (IMC'15)*, Tokyo, Japan, Oct. 2015.

[8] Cameron. (2013) A fast lock-free queue for C++. `http://moodycamel.com/blog/2013/a-fast-lock-free-queue-for-c++`. Last visited 2016-07-01.

[9] ——. (2014) A fast general purpose lock-free queue for C++. `http://moodycamel.com/blog/2014/a-fast-general-purpose-lock-free-queue-for-c++`. Last visited 2016-07-01.

[10] Facebook. (2016) A one producer and one consumer queue without locks. `https://github.com/facebook/folly/blob/master/folly/ProducerConsumerQueue.h`. Last visited 2016-07-01.

[11] ——. (2016) A high-performance bounded concurrent queue that supports multiple producers, multiple consumers. `https://github.com/facebook/folly/blob/master/folly/MPMCQueue.h`. Last visited 2016-07-01.

[12] Intel. Data plane development kit. `http://dpdk.org`. Last visited 2016-07-10.

[13] ——. (2016) Data plane development kit - Source. `http://dpdk.org/browse/dpdk/tree/lib/librte_ring/rte_ring.h#n190`. Last visited 2016-07-01.

[14] F. Fusco and L. Deri, "High speed network traffic analysis with commodity multi-core systems," in *Proceedings of the 10th ACM SIGCOMM conference on Internet measurement.* ACM, 2010, pp. 218–224.

[15] *The GNU C Library*, Free Software Foundation, 2016.

[16] B. Stroustrup, "Software development for infrastructure," *IEEE Computer*, vol. 45, no. 1, pp. 47–58, 2012.

[17] ISO, "Technical report on C++ performance," ISO/IEC, Tech. Rep. ISO/IEC 18015:2004, 2005.

[18] ——, "The ANSI C standard (C99)," ISO/IEC, Tech. Rep. ISO/IEC 9899:1999, 1999.

[19] Transparent hugepage support. `https://www.kernel.org/doc/Documentation/vm/transhuge.txt`. Last visited 2016-07-06.

[20] *The Open Group Base Specifications Issue 7*, The Open Group, 2013.

[21] J. D. McCalpin, "Memory bandwidth and machine balance in current high performance computers," *IEEE Computer Society Technical Committee on Computer Architecture (TCCA) Newsletter*, pp. 19–25, Dec. 1995.