2013•2014
FACULTEIT WETENSCHAPPEN
master in de informatica

Masterproef
Network infrastructure optimization

Promotor :
Prof. dr. Wim LAMOTTE

Copromotor :
Prof. dr. Peter QUAX

Balazs Nemeth

Proefschrift ingediend tot het behalen van de graad van master in de informatica

2013•2014
FACULTEIT WETENSCHAPPEN
master in de informatica

Masterproef

Network infrastructure optimization

Promotor :
Prof. dr. Wim LAMOTTE

Copromotor :
Prof. dr. Peter QUAX

Balazs Nemeth
Proefschrift ingediend tot het behalen van de graad van master in de informatica

universiteit hasselt | Maastricht University

Acknowledgments

First and foremost, I would like to thank the promoter, Prof. Dr. Wim Lamotte, for all his guidance. Without his recommendation, I would have never even thought of considering an internship as a thesis. I would also like to thank both Prof. Dr. Wim Lamotte and Prof. Dr. Peter Quax (the copromoter) for taking the time to attend meetings in which the topics of this thesis were discussed and to provide detailed feedback on the thesis text. Furthermore, I would like to thank Jori Liesenborgs (the assessor) for providing feedback.

Second, I would like to express my gratitude to all the people in the team at Intel of which I was part of during my internship from July 2013 to June 2014. There are two people within this team who I would like to thank explicitly. First, I want to express that it has been an honor to work with Xavier Simonart. He put an enormous amount of time both in endless technical discussions and in providing feedback on my work. He was always willing to listen to my ideas. All his questions and ideas were *very* valuable during the work on this thesis. Second, my manager during the internship, Luc Provoost, helped improving the quality of my texts and with planning the next steps for my work. He set time aside for one-on-one meetings with useful discussions even in his busy schedule.

Third, I would like to thank all the people (fellow students, teaching assistants and professors) at the University of Hasselt who have made the last five years possible.

Finally, I want to take this opportunity to express how grateful I am to all the people outside the University for helping me during my education.

Contents

Glossary

10GbE	10-gigabit Ethernet
ABI	Application Binary Interface
API	Application Programming Interface
ARP	Address Resolution Protocol
ASIC	Application Specific Integrated Circuit
Bpp	Bytes per packet
Commercial Off-The-Shelf	Commercial off-the-shelf products are products that are available to the general public.
CPE	Customer Premises Equipment
CQM	Cache QoS Monitoring
CVLAN	Customer VLAN
DCU	Data Cache Unit
DMA	Direct Memory Access
DPI	Deep Packet Inspection
DTLB	Data Translation Lookaside Buffer
DW	Double Word
FIFO	First In, First Out
FPGA	Field Programmable Gate Array
Gbps	Gigabits per second
GRE	Generic Routing Encapsulation
GT/s	Gigatransfers per second
HT	Hyper-Thread
Intel® DDIO	Intel® Data Direct I/O Technology
Intel® DPDK	Intel® Data Plane Development Kit
Intel® PCM	Intel® Performance Counter Monitor
IP	Internet Protocol

IPG	Inter-Packet Gap
ITT	Instrumentation and Tracing Technology
LIFO	Last In, First Out
LRU	Least Recently Used
mbuf	The Message (or memory) buffer used to store packets.
MLC	Mid-Level Cache
MPC	Missed Packet Count
Mpps	Mega packet per second
MSR	Model-Specific Register
MTU	Maximum Transmission Unit
NAT	Network Address Translation
NFV	Network Functions Virtualization
NIC	Network Interface Controller
OEM	Open Event Machine
PCIe	Peripheral Component Interconnect express
PDV	Packet Delay Variation
QoS	Quality of Service
RA	Reference Architecture
RFO	Read For Ownership
RMID	Resource Monitoring ID
RX	Reception
SUT	System Under Test
SVLAN	Service VLAN
TC	Traffic Class
TLB	Translation Lookaside Buffer
TLP	Transaction Layer Packet
Total Cost of Ownership	All costs associated with an asset including costs in all parts of its life cycle. Comparing TCOs, buyers can make educated decisions.
TX	Transmission
vBRAS	virtual Broadband Remote Access Server
VM	Virtual Machine
VMM	Virtual Machine Manager

Chapter 1

Introduction

1.1 Purpose of this thesis

In today's network driven world, it is becoming increasingly difficult to keep up with new standards and the customer's demands. As demand for bandwidth increases (caused by increasing demand for online video through services like Netflix), service providers are searching for ways to decrease the cost of new equipment. They are considering software based solutions with the goal to increase the flexibility and agility of their infrastructure while reducing the Total Cost of Ownership [Int22]. Within this thesis, we will study different aspects of software based implementations of network functions on top of Commercial Off-The-Shelf available hardware. This thesis is a feasibility study. *The goal of the thesis is to present an analysis of software based implementations and their performance. The analysis is used to localize bottlenecks and to determine if it is feasible to move some of the existing nodes within the network of the service provider to a software based solution. The measurement results are studied to find techniques for alleviating bottlenecks.* We will focus on specific use cases to determine how hardware and software interactions influence performance. The use cases focus on real-world industry problems faced by the service providers. After a proof of concept implementation is created, performance is evaluated through measurements. The results are studied in order to reason about specific behaviors.

Application Specific Integrated Circuits (ASICs) are being used to implement networking appliances due to the performance requirements. As ASICs are built for one purpose, they can be highly optimized to meet the performance requirements. The biggest downside is that these ASICs are not flexible. The devices cannot be updated to support new technologies. They need to be replaced when the infrastructure is updated. A more flexible solution that still meets performance requirements is the use of programmable ASICs. A programmable ASIC is also known as a Field Programmable Gate Array (FPGA). The programmable ASICs are used both for prototyping and in the final product to provide functionality that depends on the workload. At first, using programmable ASICs in combination with traditional ASICs seems to be the perfect solution (performance combined with flexibilty), but the costs remain high. Therefore, an alternative could be to use general purpose processors running software based solutions. If the software based solutions can provide acceptable performance, they can replace existing appliances while cutting costs and improving flexibility.

1.2 Overview structure

In Chapter 2, we will be introducing the basic concepts that we will refer to in later chapters. We will focus on introducing the library that enables the creation of software based implementations of networking nodes within the network of the service provider. We will introduce metrics which

we will use to evaluate system performance. It is through these metrics that we can decide if it is feasible to use the proposed techniques (i.e. these metrics allow testing if theoretical limits are reached) for software based solutions. Note that we are not implementing complete solutions and we leave out details that are needed in a complete solution. Instead, we focus on aspects that define overall performance. As a consequence, we are not developing full commercial products. Existing equipment is not sufficient to gather all statistics necessary for the analyses. We had to develop a program to collect these at the required level of detail. The program will be presented after we have defined the metrics.

In Chapter 3, we will be focusing on one specific network function: Quality of Service (QoS). We will define the traffic used for testing and the system parameters. We show performance results and we will describe how we came to these results. The goal of providing this information is to allow others to replicate the setup and verify our performance results. The first part of the chapter focuses on general notions around QoS. The second part of the chapter will look at the QoS implementation. We will be adding QoS to an existing virtual Broadband Remote Access Server (vBRAS) implementation and we will look at how QoS influences the traffic. We will measure different aspects of the system in order to understand the internal behavior.

In Chapter 4, we will present a set of techniques that have been used to optimize the software implementations. We will give background and we will motivate when and why each optimization is needed. As these optimizations are not only applicable to our implementation, we will describe more general scenarios in which they can be employed. The optimizations focus on the fast path (i.e. the tasks that are executed most frequently) as this path determines the overall performance of the system.

In Chapter 5, we will focus on one aspect of the system: caching. This subject is important for packet processing applications that require significant amount of memory. We will start with an introduction to the subject. We will then show how caching influences performance from the perspective of software. We will look at caching in recent processors to motivate why we need ways to monitor these caches. In the final section of this chapter, we will discuss how a feature that may be present in future processors can be used to monitor caching. We will show how this feature can be used specifically in the context of packet processing applications.

Finally, we draw conclusions and give ideas for future work in Chapter 6.

Chapter 2

Data plane processing applications

2.1 Introduction

Traditionally, network service providers utilize dedicated hardware within their network to meet service demands. Each node within their network provides specific functionality. At the time these appliances were first created, it was reasonable to use dedicated hardware implemented mostly with ASICs instead of general purpose processors to meet the performance requirements. Network Functions Virtualization (NFV) tries to disrupt the market by showing that these appliances can now be replaced by virtualized implementations. Within this chapter, we will lay the grounds for the following chapters in which we focus on these implementations. We start by referring to the paper that marked the beginning of network functions virtualization (Section 2.2). We then move to Intel® Data Plane Development Kit (Intel® DPDK) (Section 2.3), one of the key enablers of NFV. The details of the library are important as the library is used by the software written to evaluate performance within this thesis. As we will be using multi-core processors and multi-threaded software, we will list the relevant core organization models in the context of packet processing and we will motivate our choice in Section 2.4. Next, we define the performance metrics in Section 2.5. After calculating the maximum theoretical throughput, we describe a bottleneck that prevents reaching this theoretical throughput in Section 2.6. We will refer back to this bottleneck when discussing performance results and optimizations in the following chapters. We point out an issue with traditional equipment used to gather statistics in Section 2.7. The section describes an implementation that can be used as a workaround to these problems. In the next two sections we focus on inherent latency (Sections 2.8 and 2.9) which plays a role in all measurements related to latency. As performance optimizations are central in the context of this thesis, we will close this chapter by pointing out issues with profiling packet processing applications and how we can workaround these issues in Section 2.10.

2.2 Network functions virtualization

In October 2012, the NFV introductory white paper was released [ETS12]. The paper conveys the idea of providing network functions (such as cache servers, load balancers, application accelerators, firewalls, virus scanners, ...) through the use of software implementations in high volume servers. The biggest benefit is from an economic perspective. As a service provider, the downside of using dedicated, specialized and proprietary hardware for each network function is the diminishing return on investment. As innovation accelerates, the lifetime of each network appliance shortens. Other benefits include scalability (scaling up/down based on demand), remote management, reduced power consumption, increased flexibility towards new standards, saving physical space through consolidation.... By employing commodity servers and switches running open software to replace existing hardware, the

skills and the scale of the software industry can be leveraged. The white paper lists Intel® DPDK as one of the enablers of NFV.

2.3 Intel® Data Plane Development Kit

Intel® DPDK [Int14d, WI] is a set of libraries that can be used to develop data plane applications. Defining the control plane and the data plane is a way to separate functionality. The control plane is responsible for the configuration of the data plane. The data plane has the task of forwarding traffic according to the configuration provided by the control plane. The performance requirements for the control plane are less than the performance requirements for the data plane. The reason for this is that, compared to the data plane, the control plane has to perform less work. The applications developed using Intel® DPDK are intended to be used in packet-switched networks where high performance is required.

Typical user space applications that perform network related tasks use functionality provided by the Linux kernel. The packet destined for the user space application is copied multiple times by the kernel. Generally, the kernel tries to balance resources between multiple running processes. On the other hand, packet processing applications involve running highly tuned and specific workloads. In this case, the functionality provided by the kernel is not used and features like scheduling (and the associated context switches) introduce additional overhead. Intel® DPDK allows user space applications using the provided libraries to access the Ethernet controller directly without needing to go through the kernel. To bring the performance gains into perspective, 10-11x improvements in throughput compared to the stock Linux kernel have been reported [Win13]. One argument would be to develop the whole application in kernel space in which case the application has complete access to the hardware. The downside with this approach is that development becomes more difficult. In comparison to development in kernel space, we can rely on more tools in user space. Furthermore, as more developers work in user space, more techniques are known. Throughout this thesis, we will be using Intel® DPDK as the underlying library. To give background, we will summarize the main aspects of the provided set of libraries that are of interest in later sections.

Even with these libraries in place, optimizations are required to meet performance requirements. As we will see throughout this thesis, different aspects of the system need to be taken into account but an understanding of Intel® DPDK is required.

2.3.1 Message buffers and packet representation

The most important notion within Intel® DPDK is how packets are stored. Programs that require network access commonly rely on the socket Application Programming Interfaces (APIs). A set of features is provided by the Linux kernel for the convenience of the programmer. The programmer can focus on implementing the application specific protocols. Within Intel® DPDK, a pointer to a message buffer (mbufs) is used to represent a packet. The buffer holds the *entire* packet (Ethernet header, Internet Protocol (IP) header, ...). The packet can be transferred from one core to another simply by transferring the pointer instead of copying the whole packet. With this concept, the packet can progress through the pipeline without a single copy. This is referred to as "zero copy" and it is one of the reasons why Intel® DPDK outperforms the standard Linux kernel. Beside the actual packet data, the mbuf holds meta data (packet length, memory pool to which the mbuf belongs, ...) and provides headroom. The headroom is memory allocated before the start of the packet data. It can be used during packet encapsulation without having to move the whole packet (including the payload) in memory to accommodate the extra space requirement. Some relatively small part of the packet might still need to move. For example, encapsulating an IP packet requires to move the Ethernet header. In this case, 14 bytes need to be moved.

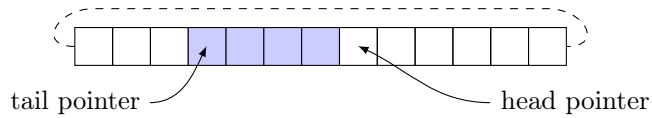


Figure 2.1: Structure of the Intel® DPDK rings. The number of elements within the queue is calculated by taking the difference between the head and tail pointers. All calculations are performed using modulo arithmetic (bit-masking the lower $n-1$ bits for a ring of capacity 2^n).

2.3.2 Memory pool

As noted in Section 2.3.1, a memory pool is used to hold the elements (for example mbuf). As an optimization, mbufs are allocated once during initialization. If they are needed, they are taken from the memory pool. When we call the free function on an element, the element is added back to the memory pool. It can then be reused in the future. During the setup of an Intel® DPDK application, a memory pool needs to be associated with an interface. When a packet arrives to the interface, an mbuf is taken from the memory pool to hold the packet. The memory pool library uses the memory allocation library implemented by Intel® DPDK (see Section 2.3.6).

The memory pool implementation from Intel® DPDK takes the computer architecture details into account to optimize performance. Modern computer systems usually support multiple memory channels[1] to increase memory bandwidth between the RAM and the memory controller. To maximize utilization of the *total* memory bandwidth available, memory accesses should be spread across the channels. In the worst case, if memory on only one channel is accessed, the available bandwidth is limited to the bandwidth of one channel. The memory pool allocation algorithm will pad entries (the details are not relevant to our discussion) to ensure that each element starts on a different channel. Note the relative memory location of elements within the memory pool needs to be controlled to ensure that memory access is spread across the channels. For this reason, all elements are allocated in a *contiguous block of memory*. The padding is added within this block. Each element will start on a different memory channel. In most workloads, the same offsets will be used for each packet as most packets are processed in similar ways. If the present memory alignment is not taken into account, access to each entry (or the same offset within each entry) within the memory pool could result in access to the same memory channel limiting the available bandwidth.

2.3.3 Rings and inter-core communication

Rings are circular buffers with a fixed size. In the simplest scenario, they are used to connect two CPU cores (a producer core and a consumer core). They are used to exchange messages and packets. A ring can be set up to support multiple producers and/or multiple consumers[2]. Enqueuing packets in a ring is implemented by copying pointers (see Section 2.3.1) and updating the head pointer to the new position. Dequeuing is implemented by moving the tail pointer forward. Figure 2.1 illustrates the structure of a ring.

The maximum number entries that the ring can hold is configurable but it is always rounded up to a power of 2. The rounding is needed so that the head and tail pointer operations can use modulo arithmetic through bit-masks. The ring library uses the memory allocation library described in Section 2.3.6. A ring is used within the memory pool library (see Section 2.3.2) to hold the free

[1] A memory channel is a bus on which memory is connected to the system. The total *theoretical* bandwidth is the sum of the bandwidth of each channel. The Intel® Server Board S2600IP provides four memory channels.

[2] As a side note, multiple producer/multiple consumer rings can be used to implement automatic load balancing by polling these rings from multiple cores. An example of this is the Open Event Machine (OEM) library [WWS13]. Due to performance constraints and the overhead introduced with inter-core communication needed to achieve automatic load balancing, this library has not been considered.

elements. If we take into account the alignment for rings, we know that memory will be wasted if the number of elements within the memory pool is not a power of 2.

2.3.4 Poll Mode Library

One of the most important features provided by Intel® DPDK is the poll mode library. The driver replaces the conventional Ethernet driver and it interfaces with other libraries from Intel® DPDK. Using this driver allows bypassing the kernel. Instead of using interrupts to signal packet arrival (which would introduce extra overhead as the current execution is suspended and resumed to handle the interrupt), the receive buffers are polled by the user application. The API provided to transmit and receive packets works with groups of packets instead of single packets. These groups are called bulks. The reasoning behind handling packets in bulks is to amortize overhead and improve locality. The Ethernet controller will use Direct Memory Access (DMA) to transfer packets from and to the configured memory locations. As head and tail pointers of the Reception (RX) and Transmission (TX) rings are updated, packet arrivals and transmissions are signaled to the user space application.

2.3.5 Environment Abstraction Layer

At initialization time, the environment abstraction layer will set up thread affinities. By affinitizing one thread per core, the overhead of the kernel moving a thread to a different core is avoided. Another benefit of this design is that the programmer knows on which core each thread will run and highly tuned workloads can be created. From this point on, we will refer to cores when we are talking about threads. The two terms are exchangeable due to the one-to-one mapping that exists between threads and cores in this context.

2.3.6 Memory allocation and huge pages

As with all processes, virtual addresses need to be translated to physical addresses. The translations are performed at the level of pages (i.e. the higher order bits of the virtual address). To speed up translations, they are stored in a cache called the Translation Lookaside Buffer (TLB). A TLB miss occurs when a translation is not cached. When this occurs, other memory accesses are needed during the search of the page table containing the mapping between virtual and physical addresses (this search is referred to as a page walk). Once the translation has finished, it is cached in the TLB. The next memory translation could benefit from the cached entry.

At any given time, the entries present in the table can be used to access a limited set of memory addresses. This is called the TLB reach. As an entry is used per page, the TLB reach also depends on the page size. For example, with traditional page sizes (4KB) and a TLB that can hold up to 16 entries[3], we can access 64 KB of memory in the best case. Larger pages are supported by the CPU and they can be used in conjunction to the default 4KB pages. These page are referred to as huge pages [Gor10]. To bring this into perspective, recent processors support pages of 1 GB. The goal of using huge pages is to increase the amount of memory accessible without a TLB miss. A reduced number of TLB misses results in less cycles spent on address translations which in turn improves overall performance.

Support for automatic use of huge pages by the kernel is available, but the performance gains are not as big compared to manual use [Cor11]. Different existing interfaces allow explicit use of this feature. Intel® DPDK uses mount points through `hugepagetlbfs` in combination with `mmap`. The actual huge page initialization is similar to the one shown at the end of the relevant Linux kernel documentation [ker]. After these huge pages have been set up, the memory allocation library provided

[3]We use 16 as an example to illustrate the TLB reach. The number of entries that can be stored within a TLB depends on the processor.

by Intel® DPDK will allocate memory within these pages. All other Intel® DPDK libraries that allocate memory depend on this library.

2.3.7 Programming without mutexes

A final concept that we touch on is not specific to Intel® DPDK. The general approach to solve contention related issues is to use mutexes. The downside is that mutexes introduce overhead. A lock needs to be acquired before work in the critical section can be performed. After the work is completed, the lock has to be released. Atomic operations can be provided by the underlying hardware. These operations are guaranteed to execute in a deterministic manner. By using these operations, some algorithms can be implemented in a "lockless" manner. The ring library (see Section 2.3.3) within Intel® DPDK uses a lockless implementation. As all inter-core communication can be accomplished using only rings, we can write complex applications (for example, 20 inter-connected cores) without having to use a single mutex.

2.4 Core organization models

Only models where a packet is handled once by a core and one core at a time are considered. Other models where packets are handled by multiple cores at the same time or where packets are passed back and forth are not used in packet processing applications due to the overhead they would introduce. Two prominent models for packet processing applications are described in the literature [Int08]. The first is the conventional pipeline model where each core in the pipeline provides a portion of the functionality. One of the advantages of this model is that access to the data structures needed to handle a packet can potentially be limited to one core and this does not require locks. Although balancing the work manually allows optimizing at a fine grained level, the disadvantage is that balancing can never be done perfectly. There will always be some cores partially idle in the pipeline. Another disadvantage is that the throughput is limited to the slowest core in the pipeline. The second is the cluster model. In this model, it is not known upfront which core will process the packet. Even though all cores are kept busy causing an overall higher utilization of the system, synchronization becomes an issue. A hybrid model that combines the cluster model and the pipeline model is also possible.

Choosing one model over the other depends on the use case and workload. Within the context of this thesis, we only focus on the pipeline model. Even though it is more difficult to balance work within the pipeline, it is not impossible. In the considered use cases, significant amount of data is accessed and updated. Using the cluster model with our use cases would require inter-core communication and locks for each shared data structure. For the same reasons, we have not considered the hybrid model.

2.5 Network performance metrics

In order to evaluate system performance, we need certain metrics. These metrics are used to make reliable comparisons between systems implementing similar functionality. They also allow to objectively study optimization techniques.

2.5.1 Throughput

Instead of measuring the amount of data (i.e. in bytes per second) that a system can handle, it is more interesting to look at the number of packets a system can process per second (i.e. packets per second) [Spi12]. The motivation is that the system is only required to look at the header. From a packet processing point of view, the payload is not important. Deep Packet Inspection (DPI) is an

Packet size	Line rate (pps)	Line rate (Mpps)
64	14880952	14.88
78	12755102	12.75
128	8445945	8.45
256	4528985	4.53
512	2349624	2.35
1024	1197318	1.20
1528	807494	0.81

Table 2.1: Line rate for different packet sizes.

example workload where the situation is different. For these workloads, the complete packet could be of interest. In this thesis, we do not consider workloads where access to the payload is required.

We will express throughput as Mega packet per second (Mpps). The maximum number of packets on a link is limited by the bandwidth of that link. We will calculate this maximum on a 10-gigabit Ethernet (10GbE) link. Although the same calculation can be made for links with a different capacities, we explicitly use 10GbE in this thesis. The smallest packet size is 64 bytes. We need to consider the Ethernet frame overhead[4] (20 Bytes per packet (Bpp)) in our calculation. It consists of the preamble (7 bytes), start of frame delimiter (1 byte) and Inter-Packet Gap (IPG) (12 bytes).

$$\text{line rate} = \frac{1.25 \times 10^9 Bps}{64Bpp + \underbrace{20Bpp}_{\text{Ethernet frame overhead}}} = 14880952pps \approx 14.88Mpps$$

When characterizing performance, we use 64 byte packets to maximize load on the system. We will also test performance using longer packets. If the system is unable to handle line rate for 64 byte packets, we still classify it as being able to handle, for example, 128 byte packets at line rate. Table 2.1 lists line rate for different packet sizes. These numbers are referenced when performance numbers are shown in later chapters. In the worst case (64 byte packets), we need to handle a packet in approximately 200 cycles with a processor running at 3 GHz[5]. Within the scope of this thesis we do not consider power saving features. Additionally, we explicitly disable turbo frequency[6]. As a consequence, all cores are running at the same clock speed.

Consider a workload that involves packet encapsulation. For these workloads, line rate is determined by the size of the encapsulated packets. For example, if the size of packets increases from 64 bytes to 78 bytes due to the encapsulation performed by the System Under Test (SUT), then line rate at 64 bytes is 12.75 Mpps. Loading the system with 14.88 Mpps will result in 14.28% of the packets being dropped as only 12.75 Mpps can be forwarded on a 10GbE link.

2.5.2 Packet delay and delay variation

The type-p-one-way-delay metric is defined in RFC 2679 [AKZ99]. This metric represents the difference in time between the clock of the source and destination host. The methodology for measuring delay is

[4] See Appendix A for the packet structure.

[5] Dividing the clock rate by the packet rate estimates the number of cycles that can be spent processing a packet if line rate needs to be maintained.

[6] Turbo frequency can improve performance but it also adds noise to the measurements.

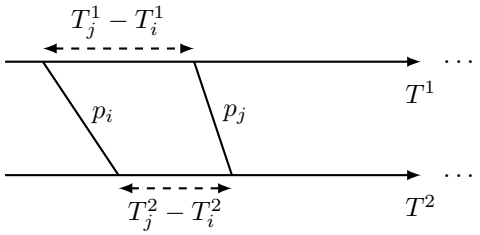


Figure 2.2: Time stamps used to calculate PDV.

to have the source host put a time stamp in the packet from which the destination hosts subtracts its clock after reception of the last byte. If the clocks are not synchronized, the Packet Delay Variation (PDV) metric [DC02] should be used instead. Intuitively, this metric can be used to measure the minimum buffer sizes of the systems between the source and destination expressed in time. This is a lower limit on the latency that the SUT introduces. Packets travel from system S^1 to system S^2. Each system has its own clock (T^1 and T^2). For every packet pair (p_i, p_j) where both p_i and p_j arrive at S^2, the PDV is given by $|(T_j^1 - T_i^1) - (T_j^2 - T_i^2)|$. Note that PDV does not give an absolute measurement for the buffer length. For example, PDV will be 0 when the delay is constant. Figure 2.2 illustrates how PDV is calculated. Besides providing a lower limit for the latency, the PDV can be used to *compare* two systems.

2.6 PCIe bandwidth

In later chapters, We will refer to limitations caused by Peripheral Component Interconnect express (PCIe). We are considering PCIe specifics because the Network Interface Controllers (NICs) we use in the systems throughout the thesis are connected through PCIe. The implementation details relating to PCIe of the NICs are not publicly available. As a consequence, we can only make educated assumptions. The goal is to show that the bandwidth requirements are close to the total available PCIe bandwidth. We start giving background on PCIe [Bud07]. Next, we look at what these PCIe specifics imply for packet processing applications.

PCIe is used to interface with a broad range of devices. The PCIe architecture uses a packet based protocol to transport information. Each PCIe device is connected through a switch and uses an address in the same manner as a host in conventional packet based networks. There are three layers defined in the specification. The three layers are: the physical layer, data link layer and the transaction layer. The payload of the transaction layer contains the actual data used by the device. The general structure of a packet is shown by Figure 2.3. After data has been sent, an acknowledgment packet is sent by the receiving side[7]. The same packet format that is used for acknowledgments is also used for flow control, power management and for vendor defined packet. The size of these packets is always 8 bytes. The structure of the packet is not important for our discussion.

The total available bandwidth with PCIe depends both on the PCIe version and the number of lanes used. We are only considering PCIe version 2.0 as the NIC used in this thesis uses this version. The bandwidth of PCIe is reported in Gigatransfers per second (GT/s). The actual available bandwidth is less due to the used encoding (8 bytes are encoded as 10 bytes during transfers[8]).

[7]Transmitted packets are buffered in a replay buffer. They are removed after a ACK packet is received. The buffer is used for retransmission in case a "not-acknowledged" (NACK) packet is received.

[8]Long streams of ones or zeros are more difficult to decode than streams that have shorter sequences. The encoding prevents these long streams.

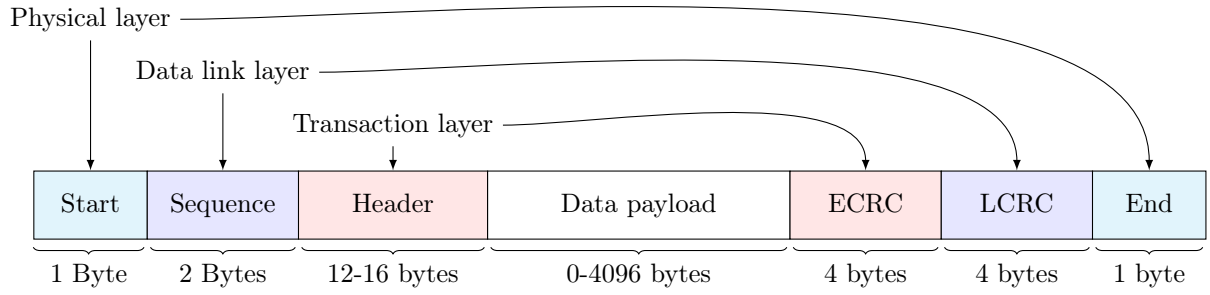


Figure 2.3: PCIe packet used to carry application data.

To convert GT/s to Gigabits per second (Gbps), we take 80% of the rate in GT/s. For PCIe, the bandwidth is specified as 5 GT/s (4 Gbps) per lane. This is the bandwidth in each direction.

In this thesis, we exclusively use the Intel® 82599 10 Gigabit Ethernet Controller [Int14h]. The NIC has two 10GbE ports. The NIC has the most bandwidth if it is connected on a x8 PCIe socket[9]. With 8 lanes, the total bandwidth (in each direction) is 32 Gbps. Comparing this bandwidth to two 10GbE links, it would seem that there is no bandwidth issue. The remainder of this section is devoted to showing that there are scenarios in which at least all the available bandwidth is required[10].

As an example scenario, we will look at receiving line rate with 64 byte packets through two ports. Receiving a packet involves the following steps as described by the datasheet [Int14h]. Although the poll mode library in Intel® DPDK (see Section 2.3.4) takes these steps into account, the steps are specific for the Intel® 82599 10 Gigabit Ethernet Controller. First, the NIC needs to fetch the next receive descriptor to use for the packet. For this, it will send a read request. The descriptor data structure contains the location that needs to be used to store the actual packet data. After the descriptor has been delivered, the packet data can be transferred to the host. Finally, the NIC will write back the receive descriptor along with extra information relating to what features have been offloaded. We assume that all these steps require transmission of a Transaction Layer Packet (TLP) packet. In the worst case, these packets add 28 byte of overhead (see Figure 2.3). We are sending 3 packets and the second packet will contain the actual Ethernet packet. To summarize, we need to transfer *at least* $3 \times 28 + 64$ bytes (or 1184 bits) per packet from the NIC to the host. The total packet rate in this example scenario is two times line rate. The required bandwidth is thus 35.24 Gbps (29.76 Mpps $\times$ 1184 bps) while the available bandwidth is 32 Gbps. The situation is worsened if packets are traveling in both directions. From the perspective of the receiving side, packets being transmitted need to be acknowledged adding to the bandwidth requirements.

2.7 Pkt-stat: A flow matching framework

During performance evaluation, the SUT needs to be loaded with traffic. If a software traffic generator is used, clearly a separate system is needed. A few details needs to be taken into account with software traffic generators. First, since hardware packet generators are standardized, the results they produce are more reliable. Second, the same time constraints ($\approx$ 200 cycles per packet) apply. Third, whenever the operating system interrupts the traffic generation software, the generated stream *could*

[9]We always use these connections when connecting the NIC, although the NIC supports x4 connections.

[10]The complete protocol would need to be reverse engineered in order to calculate the *exact* bandwidth requirements. For example, the size of the replay buffer would need to be known to predict the minimum number of acknowledgment packets sent.

be interrupted as well. For the SUT, this means that there could be a time interval when there are no packets to be processed. Finally, the bulk API that was described in Section 2.3.4 reduces the accuracy of software generators implemented with Intel® DPDK[11]. Although a single packet can be sent or received, the performance penalty might be too high. Taking all these factors into account, the measurement results will show the behavior of both the SUT and the software traffic generator. If the behavior of the software traffic generator is not completely predictable, analysing the SUT through the measurements becomes more difficult.

In practice, software generators are sufficiently accurate to test correctness and to load the system at a basic level. When final results are gathered, traffic generators implemented in hardware are used instead. These hardware generators have precise control over the timings and can maintain line rate at all times. As opposed to software solutions, hardware generators are provided as a closed system. Throughput is reported in Mpps by these generators. Licenses might be required to unlock advanced testing and statistics features and extensibility is limited. In some cases, we need statistics at a higher level of detail that cannot easily be gathered with hardware generators. As a workaround to the issues of the hardware traffic generators (extensibility) and those of the software generators (accuracy), we have developed `pkt-stat`. With this program, hardware traffic generators can be used during the test to generate traffic and to create packet captures. Statistics are collected from packet captures after the tests are completed.

`pkt-stat` reads two packet captures using libpcap [lib13][12]. The first capture contains packets sent to the SUT and the second capture contains packets received from the SUT. Each packet capture can be filtered to prevent inaccurate statistics (i.e. packets routed to interfaces on which packets are not captured can be removed). It reports the following statistics: latency (packet delay[13]), packet loss, PDV and the number of packets received between consecutive packets belonging to a stream of interest. With a stream, we are referring to a set of packets that have the same characteristics. The definition of a stream and how it is identified depends on the use case.

The design of `pkt-stat` is based on the following observations. As a black box, the SUT transforms incoming packets to outgoing packets through processing steps. Headers can change during specific workloads like encapsulation, decapsulation, Network Address Translation (NAT), ... Packets are potentially reordered or dropped. The traffic from one incoming stream could be split among a set of destination interfaces. How packets are processed depends completely on the use case. For extensibility, `pkt-stat` needs to abstract all these aspects. Finally, collecting statistics has to be done efficiently. The motivation comes from the packet rates at 10GbE and the fact that the time frame of the considered packet captures is around 500 ms (packet captures created in 500 ms are around 454 MB in size).

Processing is split up into three phases. First, during the preprocessing phase, the two packet captures are filtered to remove packets that are irrelevant for the statistics. To illustrate the use of filters, consider a SUT that routes IP packets to one of the destination interfaces depending on the configured routing tables and the destination IP address within the packets. Additionally, the SUT can handle Address Resolution Protocol (ARP) packets. These packets are used only to update internal tables and no ARP reply packets are generated. In this scenario, we can use a simple filter to remove all ARP packets. A more advanced filter is used to remove packets destined for interfaces on

[11] Although the same accuracy limitation applies to other packet processing applications, the achievable accuracy is sufficient for most workloads.

[12] Even though it is theoretically impossible to receive two packets with time stamps that differ less than the time it takes to transmit the first packet (See Section 2.8), the difference reported might still be below this theoretical limit. The culprit is the libpcap format which is limited to microsecond resolution. These errors are introduced when the time stamps are rounded. We could compensate for these errors by extrapolating time stamps from known theoretical limits, but the errors introduced by time stamp rounding in the final statistics are negligible (errors at the nanosecond scale are hidden by system behavior at the microsecond scale).

[13] The reported value is valid only if the clocks used in the two captures are synchronized.

which no capture is performed. Such a filter needs to read the same routing tables that are used by SUT during the tests.

Next, the main processing phase takes the filtered packet captures and searches for packets that are present in both streams. Packets can be compared directly when the SUT has forwarded the packets without any alteration. In more complicated workloads, packet matching requires performing the same actions as the SUT performed on the packets (encapsulation, decapsulation, translations, ...). Once an order relation on the packets has been defined[14], the packet captures are sorted to reduce the time complexity from $O(n^2)$ to $O(nlog\,(n))$. By sorting the captures, a single scan through both captures is enough to find all matches. If it is known upfront that the SUT never reorders packets, sorting is not required. The output of this second phase is a set of packet pairs. Each packet pair contains the same packet twice (once for each of the two packet captures). Additional meta-data like the time stamps from the packet captures is stored with the packet pair. As opposed to traditional methods, `pkt-stat` does not require time stamps or other data stored in the packet for its calculations. However, a packet needs to be unique within the captured set of packets. For example, if the traffic consists of UDP packets and the SUT does not use the UDP header, *up to* 2^{32} packets could be tracked using the source and destination ports. Clearly, if the capture contains more packets, the bits from the source and destination ports need to be used in conjunction with bits from other header fields.

Figure 2.4 illustrates the first two phases for a few packets. The actions of the SUT are not shown. Packets are numbered within the file. In reality, the packet captures (`send.cap` and `recv.cap`) would both contain around 6×10^6 packets (approximately 500 ms worth of packets). The goal is to find packets in `send.cap` that are also present in `recv.cap`. Depending on the workload performed by the SUT, some packets in `send.cap` may not be present in `recv.cap` and some packets in `recv.cap` may not be present in `send.cap`. Routing is an example of a workload where this is the case. By definition, routing involves sending packets to multiple interfaces while `recv.cap`[15] contains packets captured on one interface.

In this specific example, the SUT routed packet 2, 5 and 6 to an interface on which no packets were captured. A filter removes these packets from the `send.cap` as we do not have any further information on these packets. The packets in `send.cap` are captured on *one* interface. Depending on the workload, packets could enter the SUT through other interfaces. In the example shown, packet $1'$ and packets $4'$ to $7'$ are packets that are not present in `send.cap`. These were sent through other interfaces on which no packets were captured. Another filter is used to removes these packets. After filtering, all remaining packets from `send.cap` can be found in the remaining set of packets from `recv.cap` (unless the packets have been dropped or the packet captures were not created at the same time). In this specific example, the SUT reordered packet 3 and 7 and the SUT dropped packets 4 and 8.

The final phase takes the set of packet pairs and extracts the statistics. The statistics are reported *per stream*. For each of the statistics, the 0^{th} (minimum), 25^{th}, 50^{th} (median), 75^{th} and 100^{th} (maximum) percentile are reported to give an overview of the underlying distributions. Clearly, the calculated metrics need to be sorted again before percentiles can be reported.

`pkt-stat` can be extended by inheriting from the `matcher` base class. As long as the derived class processes packets in the same way as the SUT, the correct matches will be found. Depending on the traffic used for a specific use case, new filters may need to be implemented. In the same way, these are derived from the `filter` base class. They can then be applied on either of the packet captures.

Creating two packet captures might require extra equipment due to limitations in the functionality provided by the traffic generators. The traffic generators used throughout the thesis are not capable of capturing what is being generated. Only packets *arriving* at the traffic generator can be captured. We describe two setups that allow to capture `send.cap` and `recv.cap`. The port mirror feature present

[14] As a example, an order relation R between two packets p_i and p_j may be defined as $R(p_i, p_j) \Leftrightarrow p_i[42:44] \leq p_j[66:68] \wedge p_i[20:21] \leq p_j[20:21]$. With the $p_i[x:y]$ notation, we are referring to the value presented by the bytes from packet p_i in the range $[x, y]$.

[15] Currently, only two packet captures can be read.

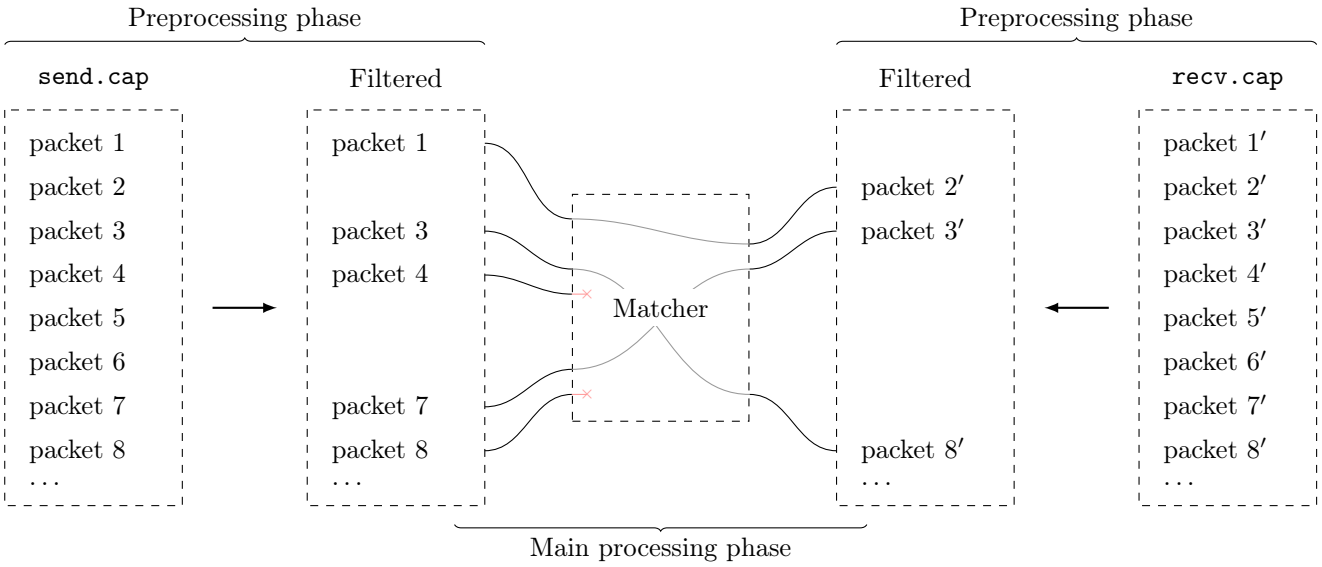


Figure 2.4: The first two phases of the execution of `pkt-stat`.

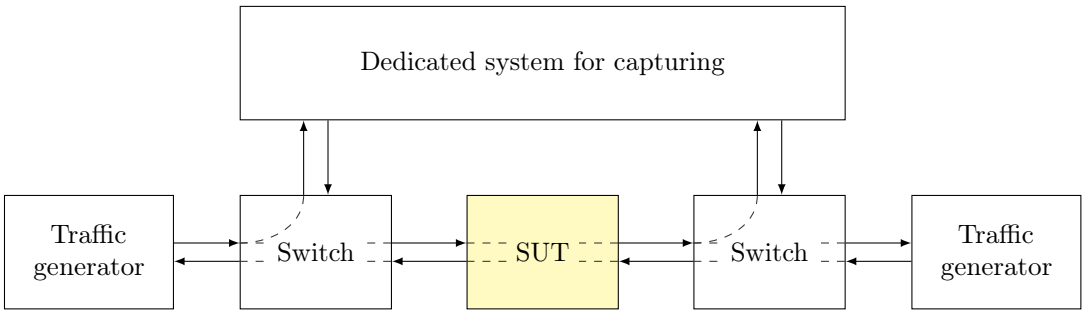


Figure 2.5: Setup to capture `send.cap` and `recv.cap` using port mirroring and a dedicated capturing system. Note that two switches are shown in the diagram while a single switch that can handle line rate with at least six ports can be used instead.

in network switches can be used to monitor traffic entering and leaving the SUT. The disadvantage of this setup is that two extra interfaces dedicated for capturing are required. The setup is illustrated in Figure 2.5. To capture traffic in the opposite direction, the switch can be reconfigured remotely through software and no physical access to the setup is required.

An alternative is to use an OpenFlow enabled switch. The switch is configured to loop back traffic from the traffic generator while still forwarding it to the SUT. Traffic leaving the SUT is dropped by the switch (this traffic merely serves to load the SUT and is not relevant for the statistics). The setup is illustrated in Figure 2.6. We can measure discrepancies in the time stamps introduced by the switch by removing the SUT from the setup. While the biggest advantage with this setup is that no extra systems are required, the downside is that it has to change physically when statistics for traffic in the opposite direction are considered. When statistics with the level of detail provided by `pkt-stat` were required in this thesis, the setup from Figure 2.6 was used.

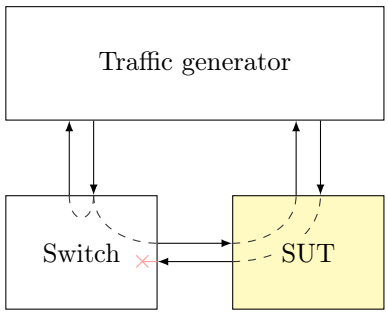


Figure 2.6: Setup with no extra hardware requirements beside an OpenFlow enabled switch to capture `send.cap` and `recv.cap`.

2.8 Inherent latency

In this section, we will look at the inherent latency for packets. The values are useful for the purpose of debugging and to partially explain measurement results. Note that we are considering the *whole* packet (we include the Ethernet overhead) in preparation for Section 2.9. The time it takes to transmit a 64 byte packet is given below. Similarly, we calculate 78 byte packets to be transmitted in 78.4 ns, 128 byte packets in 118.4 ns, ...

$$t = \frac{64B + 20B}{10Gbps} = 67.2ns \tag{2.1}$$

In packet-switched networks, the forwarding mode defines *how* the packets are forwarded [Int10]. In turn, the mode has implications not only on error detection but also on the minimum possible latency. The two main modes are cut-through switching mode and store-and-forward switching mode.

Cut-through switching mode allows for the absolute minimum latency. In this mode, packet transmission starts as soon as possible (i.e after the first 14 bytes of a packet (preamble, start of frame delimiter and destination MAC address) are received). The minimum latency with this mode is 11.2 ns. In store-and-forward switching mode, the whole packet has to arrive before transmission starts (minimum latency for 64 byte packets in this mode is 67.2 ns). The minimum latency is not only higher than with cut-through switching, but it is also proportional to the packet size. The added benefit is that the packet can be checked for errors. Intel® DPDK falls into the store-and-forward switching category. As a detail, we have always used First In, First Out (FIFO) exclusively when measuring latency. FIFO latency is calculated by taking the difference between the time at which the first byte of the packet is transmitted and at the time which the first byte of the packet is received. If the latency results need to be compared with Last In, First Out (LIFO) results, they need to be adjusted by subtracting the time it takes to transmit the packets used in the tests [Cis12].

2.9 Latency introduced by handling packets in bulk

As described in Section 2.3.4, packets are handled in bulks. Even though handling packets in bigger bulks reduces the cost per packet, more latency is introduced. Packets inside a bulk have to wait for other packets before being handled both during reception and transmission. The latency introduced by bulk handling partially explains the total latency in the system.

To use a general model, we distinguish between receiving packets and handling packets. Handling packets also includes the time required to transmit the packet. Let $t_{\text{in}}(p_i)$ be the time between packet arrivals and $t_{\text{out}}(p_i)$ be the time it takes to handle a packet. We can calculate the latency introduced

Packet size	Bulk size (#)			
	8	16	32	64
64	0.5376	1.0752	2.1504	4.3008
78	0.6272	1.2544	2.5088	5.0176
128	0.9472	1.8944	3.7888	7.5776
256	1.7664	3.5328	7.0656	14.1312
512	3.4048	6.8096	13.6192	27.2384
1024	6.6816	13.3632	26.7264	53.4528
1528	9.9072	19.8144	39.6288	79.2576

Table 2.2: Inherent latencies (μs) at line rate for different bulk sizes.

for packet $l(p_j)$ in the bulk using Equation (2.2). Equation (2.2) without the second summation gives the minimum time that needs to elapse since the moment that the packet arrived at the system before processing on packet p_j can start.

$$l(p_j) = \sum_{i=j+1}^{n} t_{\text{in}}(p_i) + \sum_{i=1}^{j} t_{\text{out}}(p_i) \tag{2.2}$$

We calculate the latency for one of the common test cases (64 byte packets and a bulk size of 64 packets at line rate). Note that we assume copying pointers to the packets between the ingress and egress happens instantaneously. For all i, $t_{\text{in}}(p_i) = 67.2$ ns and $t_{\text{out}}(p_i) = t_{\text{in}}(p_i)$. The minimal latency for each packet will always be 4.3008 μs for systems using these parameters to forward packets. Table 2.2 lists inherent latencies for a range of packet and bulk sizes. As suggested by Equation (2.2), if the input packet rate falls below line rate and while the bulk size is kept constant, latency will increase. In Section 3.9.4 we will discuss this behavior in more detail.

2.10 Profiling packet processing applications

Profiling can help with localizing bottlenecks. Intel® VTune™ Amplifier XE was the profiler used within this thesis. Although profiling by itself is generally well known, profiling packet processing applications can be tricky. One way that packet processing differs from other applications is that packet processing could generally run forever. There is no predefined amount of work to be completed after which the program would exit. Another important aspect is that we are not interested in processing time spent during initialization. We even try to move as much of the work as possible to this phase so that when the system is up and running, less work has to be performed. During initialization, internal data structures are set up and configuration files are read and parsed. Depending on the use case, we might require to run the program to load tables that depend on packets (i.e. ARP table). This portion of the execution time is less relevant as it happens only once. We are mainly looking at bottlenecks occurring during the main execution of the program. The general stages of packet processing applications is shown by Figure 2.7.

By attaching the profiler to a running process after the initialization phase, we prevent the initialization steps from polluting the profiler results. Using this technique we can already find bottlenecks that would otherwise be hidden in short profiling sessions. Typically, the profiling

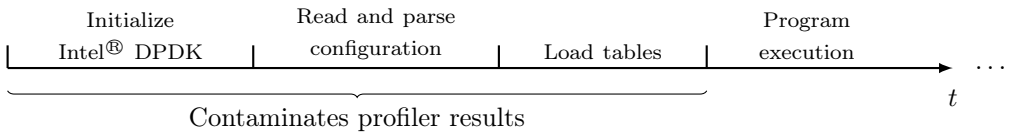


Figure 2.7: Stages of execution in packet processing applications.

workflow consists of localizing a bottleneck, applying changes and finally running the profiler again. A comparison between the results gathered during the two runs verifies if the changes had any impact.

For the comparisons to make sense, we need to instruct the profiler from within our application. Using the Instrumentation and Tracing Technology (ITT) API provided by libittnotify [Int], we can send pause and resume commands to the profiler. We start the profiler after detecting that the program is fully initialized. At the same time, we start counting the number of packets that have been processed. When we reach a predefined number of processed packets, we instruct the profiler to stop collecting information. Code changes that improve throughput will reduce the execution time and this will be reflected in comparisons of the profiler results.

Chapter 3

Quality of Service Network Function

3.1 Introduction

Software architectures and configuration options can have a dramatic impact on system performance including the throughput and latency. The primary focus of this analysis is QoS in combination with the vBRAS[1] from Reference Architecture (RA) 3.0 [Int14b]. However, it is applicable for Network Services that use QoS without any vBRAS functionality and use traffic different from QinQ.

We provide details for writing and configuring host based software applications. The complete code is publicly available [Int14e]. The platform tested uses Intel® Xeon® Processor E5-2697v2 (codenamed Ivy Bridge) and Intel® 82599 10 Gigabit Ethernet Controller (codenamed Niantic).

This chapter[2] shows that the vBRAS prototype application running on Intel® architecture can be extended with QoS functionality. We show that choosing the correct parameters and configurations throughput and latency can be maintained with acceptable service quality. We discuss how the system behaves under "normal" load and when the system is overloaded. During overload, with the correct configuration in place, the service degrades gracefully and does not come to a complete stop. Note that we mainly test how the system behaves in difficult cases (small packets, difficult to handle from a cache perspective). In reality, the traffic that the system has to process will put less stress on the system.

We evaluate Intel® Architecture based platforms for NFV workloads by better understanding the implications of software design decisions. The compute platform used during the tests is shown in Table 3.1 and an overview of the most important software components is given by Table 3.2. This chapter first describes testing of a system that uses QoS only followed by testing of the full prototype vBRAS application from RA 3.0 [Int14b] with added QoS functionality.

A deep understanding of QoS functionality, behavior and implications on the traffic flow is required to understand the results. Therefore, the chapter starts with a detailed description of how the QoS framework from Intel® DPDK works and how it can be used. We then turn our attention to QoS parameters and core allocation schemes. We will show that line rate can be achieved in the full vBRAS with QoS configuration starting from 128 byte packets in the upload direction and 152 byte packets in the download direction.

[1] A Broadband Remote Access Server is equipment that is used at the boarder of the network of the Service Provider. Customers connect to the internet through this equipment. In addition to providing connectivity and routing functionality, the same equipment can provide QoS functionality.

[2] The contents of this chapter is based on RA 3.3 [Int14a] which was written as part of the internship.

Item	Description	Notes
Platform	Intel® Server Board S2600IP Family	R2308IP4LHPC
Form factor	2U Rack Mountable	
Processor(s)	1x Intel® Xeon® CPU E5-2697, only one of the available two sockets was used.	30720KB L3 Cache (2560KB per core)
Cores	12 physical cores per CPU	24 logical cores (due to Hyper-threading)
Memory	32 GB RAM (8x 4GB)	Quad channel 1333 DDR3
NICs	2× Intel® 82599 10 Gigabit Ethernet Controller	• 2 ports per NIC • PCIe v2.0 (5.0GT/s)
BIOS	SE5C600.86B.02.01.0002 version 2.14.1219	• Hyper-threading enabled • Hardware prefetching disabled

Table 3.1: Compute platform.

Component	Function	Version/Configuration
Ubuntu 13.10	Host OS	• Kernel 3.11 from Ubuntu packages • All cores expect first core isolated[a] • System services - irqbalance service disabled • Turbo frequency disabled
Intel® DPDK	IP stack acceleration	Version 1.5.2
vBRAS prototype application		v0.10 vBRAS prototype application (developed by Intel) [Int14e]

Table 3.2: Software components.

[a]Through the isolation feature provided by the Linux kernel, we can minimize the number of processes scheduled on the isolated cores. This maximizes the CPU time that the packet processing application receives.

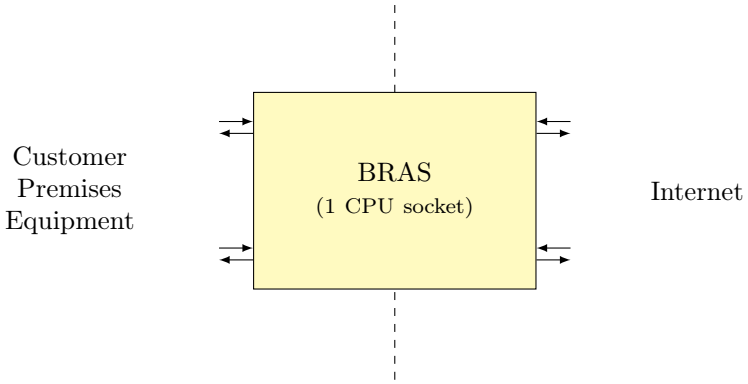


Figure 3.1: Setup used in RA 3.0.

3.2 Test traffic

This chapter describes details related to the performance impact of adding QoS to the vBRAS prototype application introduced in RA 3.0 [Int14b]. For this reason, a similar test configuration is used. The system uses two Intel® 82599 10 Gigabit Ethernet Controllers. Two ports are facing the customers and use QinQ[3]. The two others are facing the core network and use Generic Routing Encapsulation (GRE) tunneling. Only one CPU socket is being used. The system routes and translates traffic between the two sides. The setup is shown in Figure 3.1.

3.2.1 QoS with vBRAS traffic

Continuing on the work in RA 3.0, the same test traffic was used. The traffic configuration is repeated here with remarks relevant to QoS. The traffic consists of 64 byte Customer Premises Equipment (CPE) packets on one side and 78 byte core network packets on the other side. CPE traffic is applied to two out of four interfaces (these interfaces are facing the customers) and the core traffic is applied to the remaining two interfaces (core network facing).

The CPE packets are double-tagged VLAN packets containing either UDP/IP (500 out of 501 packets) or ARP (1 out of 501 packets). With QinQ, 4096^2 users could be identified. For our use case, we configure 65536 users. Varying 7 bits in the first service VLAN tag and 8 bits in the customer VLAN tag defines 32768 unique VLAN combinations. Two such sets are created by using an extra bit in the service VLAN. The value of this bit depends on which of the two customer facing interfaces traffic is generated to. Each unique VLAN combination defines a user. In total, the VLAN combinations create two sets of 32768 users, one for each of the two customer facing interfaces.

The core network traffic consist of GRE tunneled packets. IP was used both for the payload and the delivery protocol. As there is a one-to-one mapping between a QinQ tag and a GRE ID, 65536 different GRE IDs are used. Traffic destined for both sets of users is generated on each of the two interfaces.

The traffic is either configured in equal share mode or in single user oversubscribed mode. In both cases, ARP is treated separately. In equal share mode each user is transmitting at 31.87 KBps

[3]QinQ is also known as "stacked VLANs". It is a standard employed by service providers when VLANs are used to identify users. With a single VLAN tag, only 4096 users can be identified. Stacking two VLANs increases the limit to 4096^2. An alternative use of stacked VLANs is tagging packets that have been tagged by the customer. In this case, the service provider adds its own Service VLAN (SVLAN) tag before the existing Customer VLAN (CVLAN) tag. The added tag is used to prevent collisions of VLAN ID's between customers and packets can be handled without changing the original VLANs [Cis05].

(10 Gbps $\times 84/98 \times 500/501/32768$ users) and in oversubscribed mode, 32767 users are transmitting at 31.86 KBps and one user is transmitting at 250 KBps (or 2 Mbps). The actual transmission rate for the payload may differ from the user's perspective. This configuration allows testing of QoS rate-limiting for a single user while the system is under load.

3.2.2 QoS Intel® DPDK example traffic

Intel® DPDK includes a QoS example application. This application expects similar traffic as the vBRAS prototype application. The traffic consists of double-tagged VLANs where the service VLAN ID is set to 0, the customer VLAN ID has 10 random bits in the tag, the 3^{rd} and 4^{th} byte of the destination address of the IP header are also random. In contrast to the vBRAS, only a single type of packet stream is needed for the QoS Intel® DPDK example.

3.2.3 Use case requirements

The configuration for the vBRAS with QoS is based on a use case. We list the requirements in this section and refer to them throughout the text when we configure the system and when we make design choices. A total of four interfaces need to be supported. The customers (CPE) reside behind two of the four interfaces. Each of these interfaces needs to support up to 32768 users. The core network is accessible through the remaining two interfaces. Each user is rate limited at 1 Mbps both in the upload direction and in the download direction. The double QinQ tagged packets define the user's traffic rate. Up to 5 ms of traffic needs to be buffered per user.

3.3 Benefits of QoS

In a system without QoS, no guarantees about flows and their rates can be made. A single flow could, in theory, congest other flows as all traffic is handled in a best-effort fashion. If congestion occurs in the system, packets are simply dropped instead of being buffered and sent during idle periods. There is no selection mechanism to decide which packets need to be dropped based on priority or any other attribute of the packet.

QoS enables traffic policing where flows can only influence each other based on predetermined parameters. Depending on the configuration and system parameters, oversubscription and jitter in the incoming traffic can be reduced in the outgoing stream. In essence, the way that this is achieved is through the use of queues and precisely managing when packets are transmitted or dropped.

3.4 QoS building blocks

We start by describing the general building blocks for implementing QoS in a pipeline. There is some overlap with the information available in the Intel® DPDK programmer's manual [Int14i] but the focus here is to describe how the system works from an architectural perspective. Figure 3.2 shows these building blocks in the pipeline. This pipeline is proposed by the Intel® DPDK programmer's manual. We will explain when and why we have deviated from this design. Any system using QoS will, at least partially, use these building blocks.

- Pkt I/O RX: Packet reception from the physical interface.
- Pkt Parse: Determine destination and path inside the system.
- Classif: Classify packets to user/flow (mandatory for QoS).

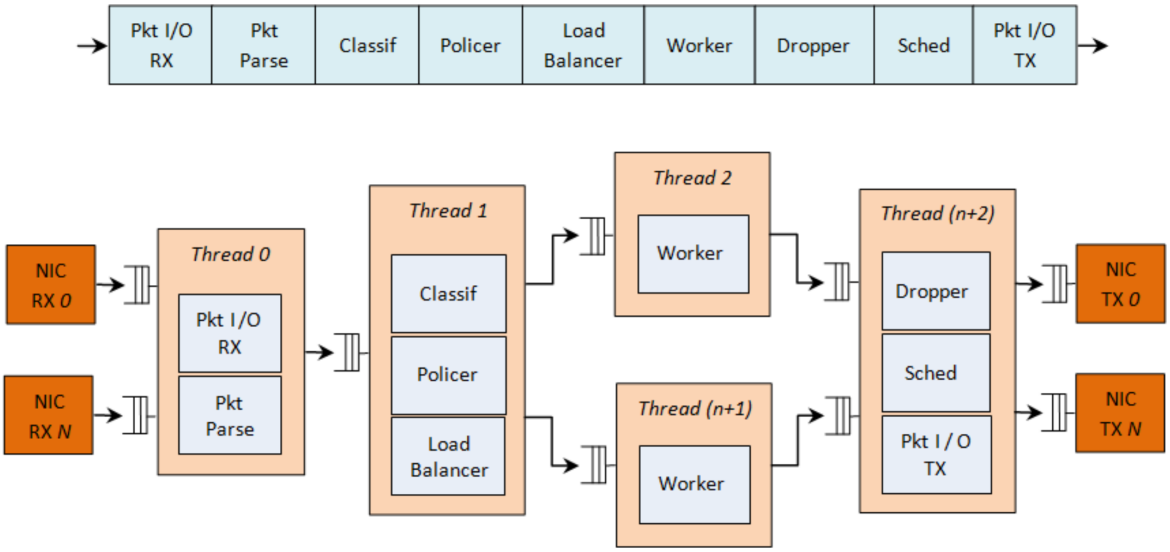


Figure 3.2: QoS building blocks [Int14i].

- Policer: Traffic metering (traffic coloring) based on RFC 2697 [HG99a] and 2698 [HG99b][4] (Optional for QoS).
- Load Balancer: Distribute packets between worker threads.
- Worker: Perform actual workload (for example, vBRAS functionality).
- Dropper: Congestion mechanism based on (weighted) random early detection (Optional for QoS)
- Scheduler: Implements actual QoS scheduler which uses information from the classification stage. (mandatory for QoS)
- Pkt I/O TX: Packet transmission to the physical interface.

A minimal QoS implementation thus requires a classifier and a scheduler. Even in this case, QoS parameters, classification implementation (mapping from packet to a tuple) and core layout (i.e. logical cores[5] vs. physical cores, combining QoS and TX on a single core ...) have implications on performance and must be chosen carefully as we will show later.

3.5 Intel® DPDK QoS implementation specifics

3.5.1 Enqueue and dequeue operations

At the highest level, QoS is a buffer that can hold thousands or millions of packets. In reality, this buffer is a collection of queues organized in a hierarchy of five levels. The port (i.e. the physical

[4]These RFCs respectively describe a Single Rate Three Color Marker and a Two Rate Three Color Marker. Traffic coloring is supported by the Intel® DPDK QoS implementation. Coloring is mentioned for completeness, but it has not been considered in this thesis.

[5]A logical core or a Hyper-Thread (HT) refers to a hardware thread which is provided by Intel® Hyper-Threading Technology.

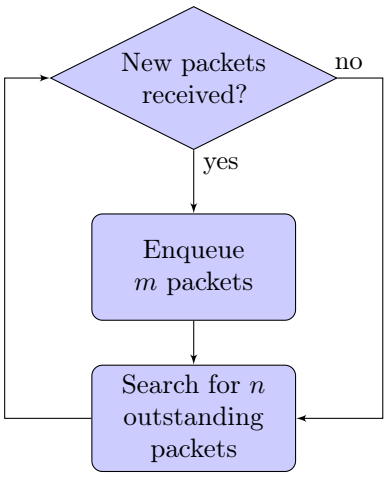


Figure 3.3: Flow chart describing QoS API usage.

interface) is at the root of the hierarchy followed by the subport (a set of users), the pipes (specific users), the Traffic Classes (TCs) (each with a strict priority) and at the leaves, the queues (2^n pointers to packets per queue).

The QoS can be divided in two distinct steps of which the first one is the enqueue operation and the second one is the dequeue operation. Note that both steps are executed on the same (logical) core. The reason for this design choice, as noted by the Intel® DPDK programmer's guide [Int14i], is to avoid having to use locks and the MESI cache coherency protocol[6].

The enqueue operation uses the information from the classification step to decide in which queue to store the packet. This is the only point where the QoS can decide to drop the packet. The decision is made only if the specific queue is full. Note that this does not necessarily mean that the user, to whom the queue belongs, was oversubscribing. Instead, it means that not enough packets were taken out from that queue to allow more packets to be buffered. The cause for this is either that the system is taking out packets slower on purpose (rate limit configuration) or it is not able to keep up due to performance limitation. This more general description of dropping packets allows to explain and to reason about some of the behavior encountered during runtime.

The dequeue operation starts where the last dequeue operation finished. It consists of searching for outstanding packets. If such packets are found, they are selected for dequeuing and the relevant counters are updated. The dequeue operation runs until the requested number of packets is found or until all the active queues have been searched for.

The flowchart in Figure 3.3 illustrates how the QoS task should be implemented on a core. Both the enqueue and dequeue operations work with a bulk of packets. Typically, the bulk size used during enqueuing is bigger than during dequeuing (i.e. 64 vs. 32[7]). The reasons for this are discussed next.

If M packets would be used in both steps, the same packets would be dequeued immediately after they have been enqueued. This would, in most cases, result in all queues being empty and at most M packets being enqueued at any time. In turn, this would cause the dequeue operation to become very expensive. The dequeue operation has to search for the requested number of packets. If a few packets are buffered, it is more expensive to find the requested number of packets. As packets are checked before they are dequeued, checking a packet only to find that it cannot be dequeued is a waste of

[6]For now, it is only important to note that the protocol can have impact on performance. We will describe how the protocol works in Section 5.5.

[7]These bulk sizes were determined empirically.

cycles. Hence the QoS core becomes too slow and cannot handle all incoming packets. Buffering more packets increases the probability that a packet, that is allowed to be transmitted, will be checked next. For the same reason, it does not help increasing the number of packets to request during the dequeue operation.

The alternative is to have the number of packets dequeued at each iteration be lower than the number of packets which can be enqueued in each operation. This will initially cause the number of buffered packets to increase. After some time, the situation will stabilize. The cost of the dequeue operation will be the same as the cost of the enqueue operation. The dequeue operation will speed up as there are more packets buffered. Hence there will be more enqueue/dequeue loops per second and the QoS loop will not enqueue packets at each loop. For the 64 vs. 32 case, this means that if the input rate is maintained, there are on average 2 dequeue operations for each enqueue operation.

In practice, we have seen that QoS converges to some number of buffered packets for the input flows we have tested. The number of buffered packets is either limited by the maximum number of mbufs we configure or it is determined by the given configuration parameters, clock rate, input flows, memory hierarchy, etc ...

Figure 3.4 shows the enqueue and dequeue operations over time. The duration of each operation is shown by the length of the lines. The input rate, output rate and number of packets buffered in the QoS are also shown. The vertical axis is combination of rates, number of packets and operation types. The cost of the dequeue operation changes depending on the number of packets being queued in the QoS. In Figure 3.4, the lines that represent the dequeue operation are drawn progressively shorter. The initial output rate depends on the ratio between the number of packets used in the enqueue operation and dequeue operation.

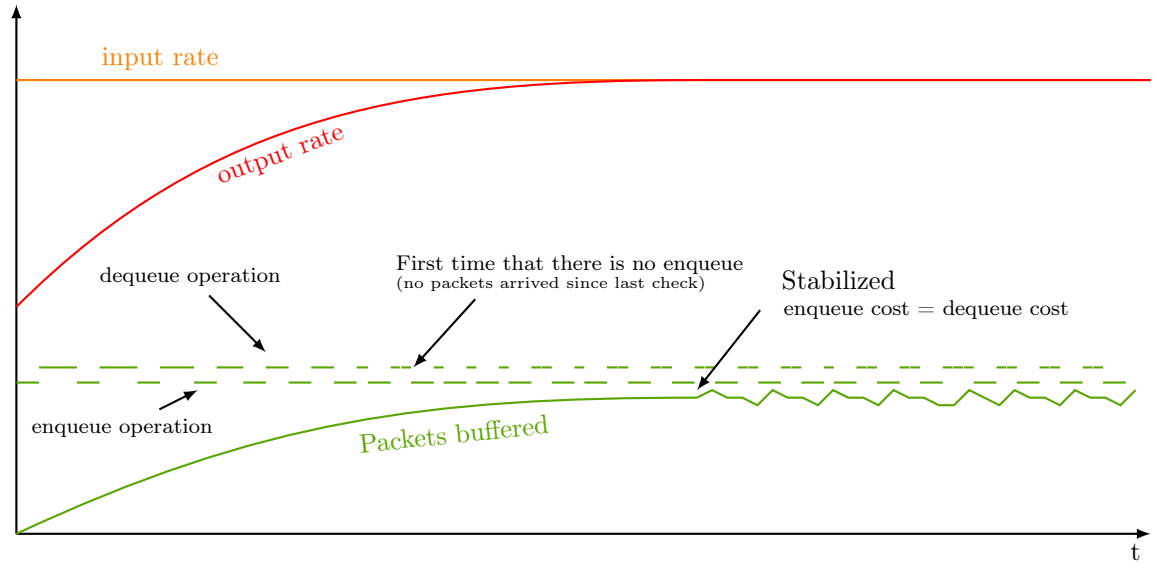


Figure 3.4: Duration of enqueue/dequeue operations, number of packets buffered and input/output rate (conceptual).

3.5.2 Token bucket for rate limiting

The Token bucket (or equivalently the leaky bucket) is a well known algorithm which implements rate limiting and manages bursts. It can be used in other applications beside packet switched networks.

We briefly discuss how the token bucket works as the Intel® DPDK QoS implementation uses token buckets at its core.

The token bucket algorithm consists of a counter (representing the tokens) which is incremented at a predetermined rate (called the bucket rate). The value of the counter represents credits that are consumed when packets are sent. Packets can only be sent if enough credits reside in the bucket. An extra parameter, beside the rate, defines the maximum size (value) of the token bucket. For a given input stream and an initial number of tokens, we can calculate the time during which the input stream is allowed to pass through without limiting the rate as: (initial tokens)/(input stream rate - bucket rate). If the input rate is below the bucket rate, the stream is allowed to pass. The effect of using a queue with the token bucket algorithm will be discussed later.

3.5.3 QoS data structures and parameters

In this section we go over the different data structures in the hierarchy and how the QoS can be configured. We discuss how our use case translates to configuration parameters. The data structures are relevant in the dequeue step. Recall that the enqueue step simply queues packets without being concerned about the data structure states. The enqueue step only checks if there is space available in the relevant queue.

The first parameter is the subport (i.e. the number of sets of users). QoS enforcement will be limited within each set of users. Each set will receive a preconfigured portion of the total rate of the port. In our configuration, we always configure one subport as we allow all flows to compete.

The second parameter is the number of pipes (i.e. the number of users). Clearly, the more users are configured, the more management data has to be stored. As we will see later, this parameter impacts performance in several ways. The Intel® DPDK QoS example configures 4096 users by default, while we study a use case requiring 32768 users per interface and per QoS instance.

The third and fourth parameter is the token bucket size (`tb_size`) and rate (`tb_rate`). The `tb_size` parameter is the previously discussed maximum token bucket size and the `tb_rate` defines the rate in unit of bytes/second. When the rate parameter is configured, Intel® DPDK calculates the token bucket period p, and the credit per period c, for the token bucket where the token bucket period is as small as possible within an error of 10^{-7} so that:

$$\texttt{tb\_rate} \approx \frac{c}{p}$$

Time is expressed in unit of bytes instead of seconds which in turn is calculated from line rate (i.e. 1.25 GB is equal to one second and 10 bytes is equal to 8 nanoseconds). The reason for this is that all calculations happen in unit of bytes and converting time to bytes is thus convenient.

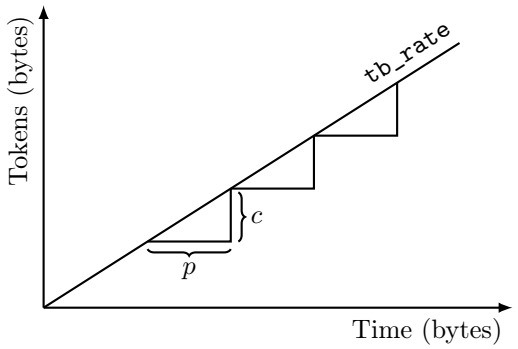


Figure 3.5: Illustration of token bucket period and credits per period.

The minimum time that has to elapse before the token is rewarded c credits is p. If more than p time elapses between updates to the token bucket, then more credits will be rewarded. In general, if the time between updates is $x\%$ of p, then $(1-x)\%$ of the updates will be zero and $x\%$ of the updates will add c credits. This means that if we configure the system for N users with equal share and if c is calculated to be 1 byte, we know that p should approximately equal to N bytes. Configuring for 32768 users, results in c being set to 1 byte and the period being set to 32680 bytes ($\approx$ 32768 bytes). For perfect accuracy (and to schedule as many packets between two packets from the same user as possible), we have to visit the same token bucket at least every 30.52 μs (1/32768 s). In the case that the time between updates takes longer, the rate limiting will be correct at the least common multiple of 30.52 μs and the actual period between visits to the token bucket. The calculation is for the absolute worst case and is far from realistic. For example, if only an accuracy of 2 bytes instead of 1 byte is required, the minimum update interval becomes 61.04 μs. In general, for N users sharing the total bandwidth equally and for perfect accuracy, we can spend a maximum of N^{-2} seconds processing each user (if all users have packets buffered, see the optimization discussed next). The reason for this is that each of the N users has to be processed N times per second.

As an optimization, if a queue is empty, it is not marked in the active bitmap and it is never visited during the dequeue step. Tokens are only updated for the active queues (i.e. there is no "update all token buckets" step executed periodically). This relaxes the requirement of N^{-2} seconds processing time for each user to processing time for each active user.

A token bucket is maintained for each pipe. It is used for the lowest level traffic policing algorithm. A token bucket is also maintained at the level of a subport. In all our test cases, we have used a single subport, thus the token bucket at this level limits the total outgoing rate of packets from the QoS at line rate (even if the outgoing rate is not limited at the pipe level) as long as the number of tokens inside this bucket is low.

The fifth parameter is the TC rate (`tc_rate`) and TC period (`tc_period`). The unit for TC period is ms, but because a byte is the common factor in all parts of the QoS, Intel® DPDK converts this to bytes at initialization time. The unit for TC rate is bytes per second, which is converted to bytes per TC period. The user is responsible for choosing the correct period in this case. Choosing it too low, may require updating the TC period multiple times with the cost of processing while choosing it too high might result in inaccuracies and more packets being queued.

The sixth and final parameter is the queue size. The length of the queue determines how many packets (not bytes) can be buffered per user. The queue size should not be too big (memory requirements) nor should it be too small (buffering requirements).

Depending on the achieved accuracy (as discussed above), a minimum queue length is required. The ideal situation would be that perfect accuracy is achieved. If this is the case, and the user is transmitting packets at exactly the allowed rate, a queue length of 1 would suffice. The last packet would be taken out before the time the new packet arrives. Note that handling packets in bulk pushes the required queue length from 1 to bulk size. In the worst case, the bulk could consist of packets belonging to the same user.

Another constraint can come from the use case. Our use case was to be able to buffer up to 5 ms of data. We have provisioned the buffers for a slightly different use case where users are limited at 10 Mbps (10 times more than our original use case). The buffer requirements that we calculate are also sufficient for 1 Mbps rate limiting. We calculate that to sustain 10 Mbps rate for 5 ms, we need to buffer 75 packets in the worst case (64 byte packets). Due to the optimizations inside the QoS implementation, we are required to round up to the next power of 2. With this in mind, we will be able to buffer at least 8.6016 ms instead of 5 ms.

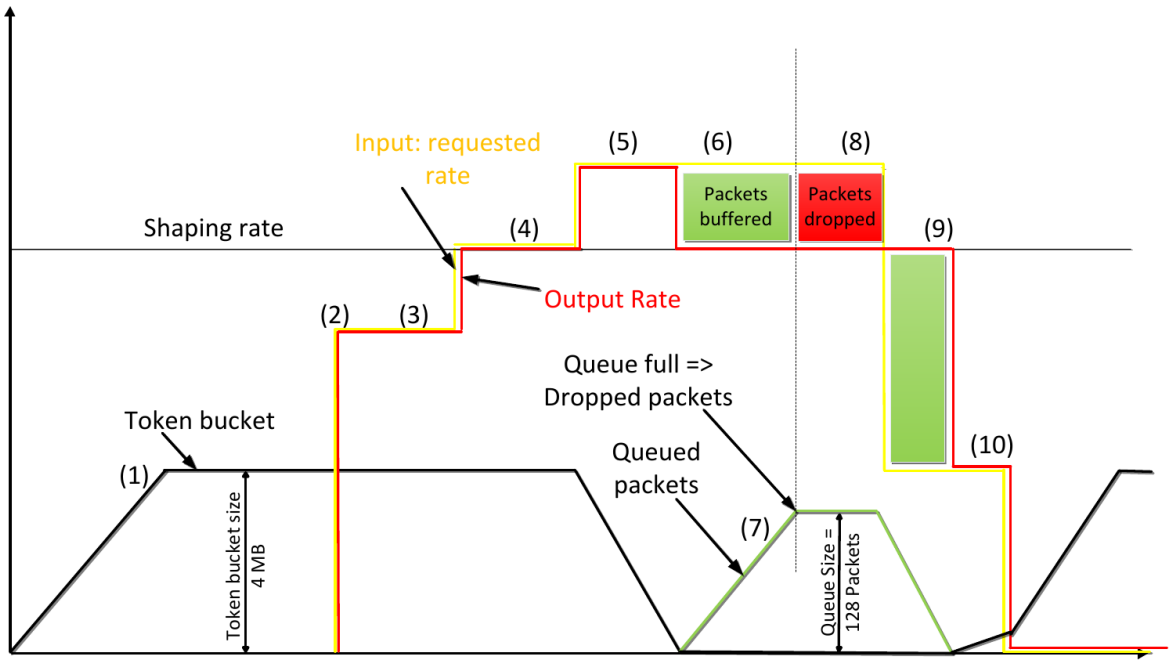


Figure 3.6: Token bucket rate limiting for a single user (conceptual).

3.5.4 QoS data structures during execution

We now look at how the data structures evolve during execution and how the input traffic changes the internal states. Assuming perfect accuracy, Figure 3.6 illustrates, for a single user, how the token bucket limits the rate and when packets are being enqueued and dequeued. Number of tokens in the token bucket, number of packets queued, input/output rate and the shaping rate are shown.

Initially, if the input rate is zero (no packets), tokens in the token bucket can accumulate up to a moment when the token bucket is full (1). When first packet enters the system, the output rate and the input rate are the same. There are enough tokens in the token bucket to allow packets to pass through (2). If the input rate is lower than (3) or equal to (4) the shaping rate, then the number of tokens in the token bucket does not decrease. Consumed tokens are replaced immediately. When the input rate becomes higher that the shaping rate (5), the number of tokens in the token bucket starts to decrease. If the input rate remains high for some time, after a while all tokens will be consumed, and the output rate decreases up to the shaping rate (6). At the same time, packets are buffered (7). If rate remains high, the packet queue will also become full, and we need to start dropping packets (8). Output rate remains the same as the shaping rate (8). When the input rate finally decreases below the shaping rate, (9), the output rate can remain at the shaping rate, higher than the input rate, as long as there are packets queued. When there are no more queued packet, and if input rate is still lower than shaping rate, the number of tokens in the token bucket can again increase (10).

The behavior is shown at a higher level of detail in Figure 3.7). Note that in this case, we don't assume that we have perfect accuracy (the interval between updates can be bigger than `tb_period`). As a result, a bulk of packets is dequeued every time when a set of tokens become available. If perfect accuracy would be achieved, a single packet would be dequeued and transmitted at line rate. The time between consecutive packets of the same user ensures that the rate is conforming to the configured shaping rate.

Comparing the input traffic to the output traffic as a whole allows explaining some of the

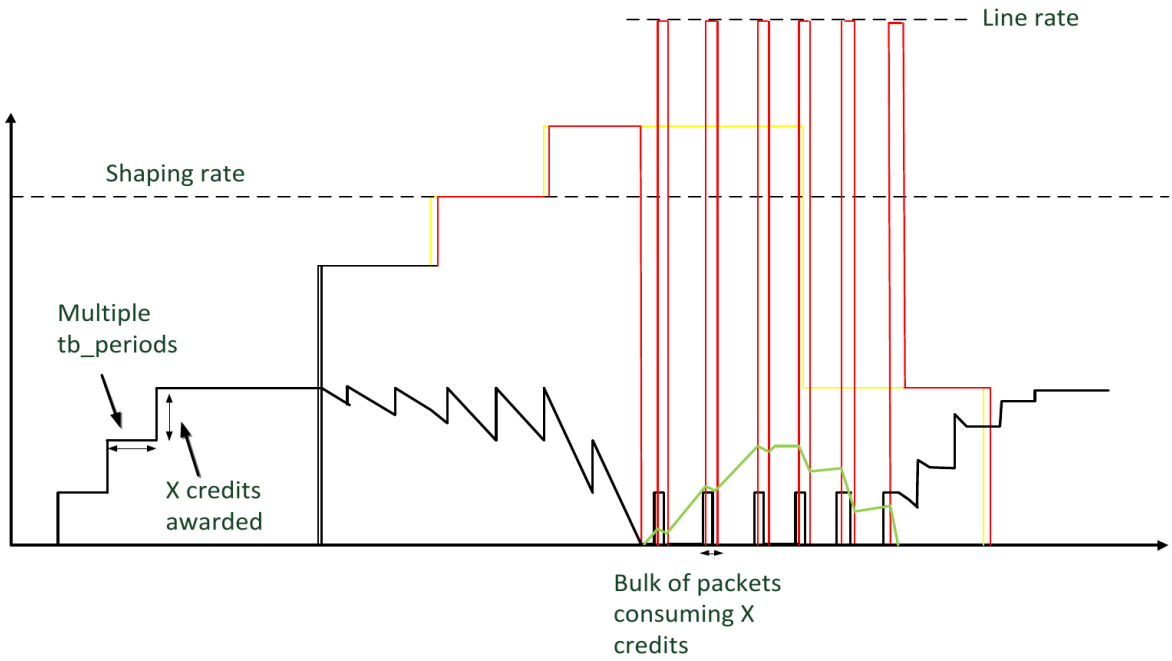


Figure 3.7: Detailed data structure states and input/output rate for a single user (conceptual).

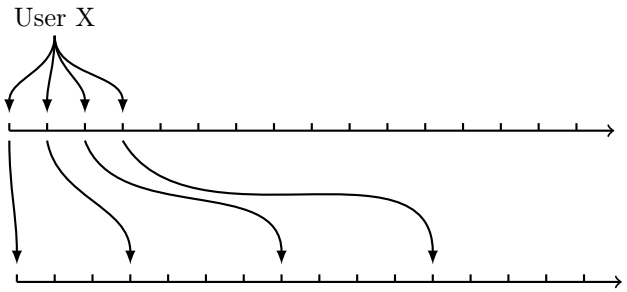


Figure 3.8: A burst of packets is spread over time.

performance results shown later. For this reason, we discuss the effect of QoS on the input stream. The task of the QoS can be seen as choosing a packet from the set of packets that are buffered with the goal to reduce jitter and to make sure packet rates are within limits. This brings with it that packets are reordering at global scope. Inherently, this adds latency. The total latency added depends on the number of packets buffered, the accuracy, and the input traffic.

Consider the case where a user is transmitting at line rate during a time interval. In this time interval a few[8] packets will be transmitted for this user. All packets will be queued. The queue is drained at the configured rate and packets for other users (or pauses) are inserted in between packet of the same user. The latency is proportional to the burstiness of the traffic and depends on the number of packets scheduled between consecutive packets (which depend on the number of users). An example situation is shown in Figure 3.8.

[8]It is possible that no packets are transmitted if the time interval is sufficiently short.

3.6 Receive message buffer requirement with QoS

The use case we have studied puts a requirement on the number of packets (and thus the number of mbufs) to be able to buffer. The requirement to be able to buffer 5 ms at 10Mbps translates to the ability to buffer 75 packets. There are a total of 1000 such flows that can occupy the total available bandwidth (10 Gbps) per port. Thus, we calculate the minimum number of mbufs to be 75000 for the QoS alone. Extra mbufs are needed as packets can reside in rings and descriptors throughout the system. For example, the vBRAS prototype application requires a minimum of 16384 mbufs.

An extra factor in determining the number of mbufs in the system is the cost/packet ratio during the dequeue operation. Allowing more packets to be buffered, reduces the cost/packet during the dequeue operation. One way to see this is that the probability of a visit to a queue that will result in no packets being dequeued is lowered.

This means that the cycles/packet during the dequeue operation changes depending on the total number of packets in the buffers. A high cost compared to the incoming packet rate (see Figure 3.3) will cause packets to arrive each of the iterations causing the number of packets buffered to grow indefinitely. In this case the total number of mbufs limits this growth. We have chosen to configure 131072 mbufs per interface. Note that using 131072 mbufs limits the maximum latency for 64 byte packets to 12.17 ms if throughput can be maintained at line rate and packets are not reordered (commonly known as Little's Law [Bar07]). The reason that the latency can be higher in the QoS is that packets can be reordered.

3.7 Managed packet drop

Dropping packets does not come for free. To decide if a packet will be dropped, the QoS has to index in the hierarchy to find the correct queue using the information provided by the classification stage. Only after this, can the decision be made to enqueue the packet (free space is available in the queue) or to drop the packet. Ideally, each packet should be indexed in the hierarchy by the QoS task.

Given that QoS can handle only a finite number of packets per second, the performance can decrease drastically if all processing power is spent on dropping packets. An example scenario is the case where two incoming 10GbE streams are routed to the same QoS core. For this reason, we suggest to use a separate token bucket on the enqueue side to limit the incoming packet rate. The extra token bucket introduces rate-limiting inaccuracies on some input streams (all flows are considered as one and rate-limited as such), but it can be used to increase throughput.

3.8 QoS in the pipeline

3.8.1 Correctness in the pipeline

If possible, the QoS task should be incorporated as the last task in a packet handling pipeline. The reason for this is that the QoS task manages how packets are scheduled (timings, rates, bulkiness, packet order ...). The position and timing of each packet in the output stream is controlled by the QoS. In some cases, more tasks can follow QoS in the pipeline. Figure 3.9 shows the required behavior to ensure that QoS is policing correctly.

Depending on the internal state (tokens, priority, time ...) the QoS keeps track of the stream by updating its internal states. Packet transmission from the QoS is reflected in these states. Thus, packet drops caused by core congestion should be avoided in tasks residing in the pipeline after the QoS. In practice, this is implemented by blocking (i.e. retrying on fails) on transmit queues in the pipeline. In contrast, packets can be dropped before the QoS. Before the moment that the QoS has touched a packet, no guarantees have been made about the stream.

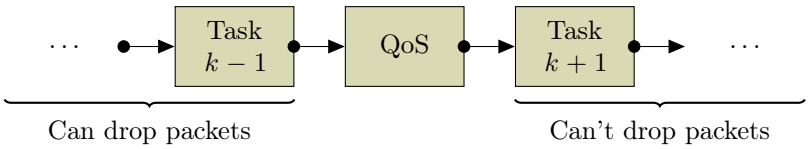


Figure 3.9: QoS in the pipeline and packet drops.

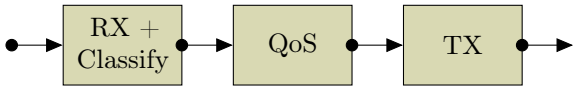


Figure 3.10: Each core with separate task.

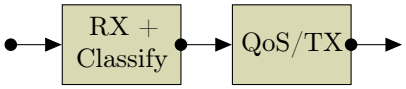


Figure 3.11: Combined Qos and TX task.

Blocking on transmit queues has an added advantage. No work (and PCIe bandwidth) is wasted on packets that would be dropped later in the pipeline. Only packets that don't conform to the allowed rate will be dropped by the QoS. The disadvantage is that latency is increased using this scheme. Instead of handling "fresh" packets, these packets are dropped before they enter the pipeline and "old" packets are pushed through the pipeline instead.

3.8.2 Impact of core layout on performance

We know from Section 3.4 that we need at least the following four tasks: RX, classify, QoS and TX. In this section, we discuss and show performance data for assigning these tasks to cores in different ways.

The first configuration is shown in Figure 3.10. The packets are received in the first core and are immediately classified. This means that the contents of each packet is read to determine the flow that the packet belongs to. The information is stored in the `rx_mbuf`. The QoS core will use this information in the enqueue step. After the packet has been enqueued and selected as possible candidate to be dequeued, the packet length is accessed to determine if it can be dequeued. Note that this requires the cache line containing the packet length (`rx_mbuf` metadata) to be present in cache. In most cases, more packets are queued in the QoS than cache lines fit in cache. After the packet has been dequeued, it is passed to the TX core which queues the packets to the physical interface without consulting the packet's content.

The second configuration is shown in Figure 3.11. It can be compared to the configuration in Figure 3.10 except that in this case, the second core not only dequeues packets but also has the task of transmitting them to the interface (extra step after dequeue operation in Figure 3.3. Recall that the dequeue operation should be as fast as possible, but this second configuration adds to the cost/packet from the QoS task's perspective slowing down the dequeue operation. Also note that transmitting packets to an interface requires more processing resources (compared to rings, more data structures need to be updated). We can predict that, for these reasons, the second configuration will perform worse both when testing for latency (more packets will be queued in the QoS) and for throughput (cost per packet is too high).

Using the Intel® DPDK provided QoS example, we can configure both scenarios. By default, the example allocates 2097152 mbufs, each having a fixed size of 1720 bytes. We add code to sample the number of packets in the QoS by keeping track of the number of packets enqueued and dequeued.

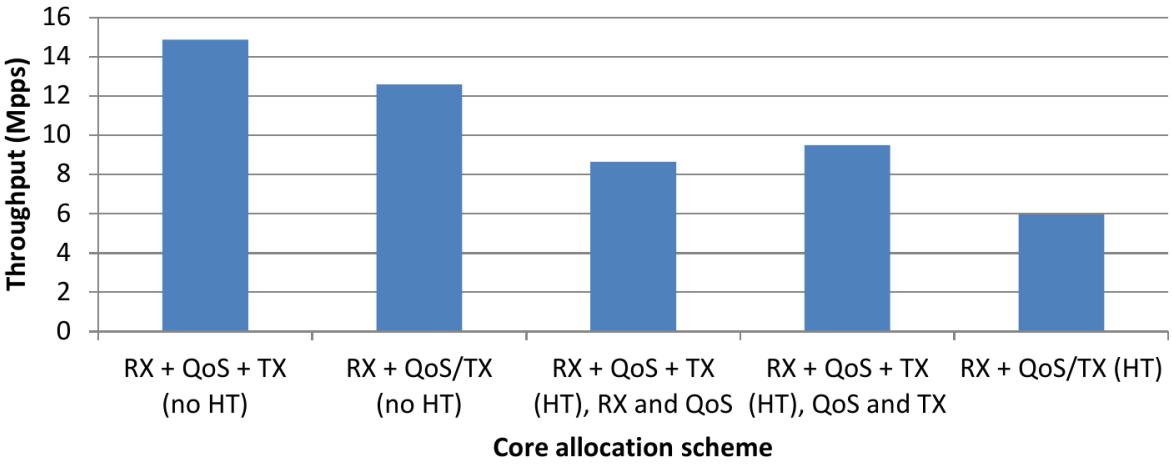


Figure 3.12: Intel® DPDK QoS example with a single QoS instance: Influence of core allocation scheme on performance.

Core allocation scheme	**Hyper-threading**	**Approximate number of buffered packets in QOS (sampled every second)**	**Number of physical cores**
RX + QoS + TX	No	110K	3
RX + QoS/TX	No	2096K	2
RX + QoS + TX	RX + QoS	40K	2
RX + QoS + TX	QoS + TX	2096K	2
RX + QoS/TX	RX + QoS/TX	2096K	1

Table 3.3: Number of packets buffered in QoS.

The core allocation scheme where the RX and TX cores are hyper-threaded would not fit in the final vBRAS architecture. For this reason it has not been considered. The results are shown by Figure 3.12[9] and by Table 3.3. The reported estimate for the packets buffered is the average of samples taken every second. It serves only as an indication for the actual number of packets buffered in the QoS data structure.

Line rate is achieved only in the first case where we configure the core allocation scheme from Figure 3.10. The total number of packets buffered in the QoS is 110K. This number is what the QoS converged to and will depend on the platform.

Combining QoS and TX in a single core shows that in both the hyper-threaded and non-hyper-threaded setup, approximately all (2 million) mbufs are residing in the QoS hierarchy. Only new packets can be received if packets are dequeued and transmitted or dropped by the QoS. We can conclude that, in this case, the dequeue operation is the bottleneck.

[9]Note that, using the platform described by Tables 3.1 and 3.2, the results for throughput and latency presented in this thesis can be repeated. The measured throughput will be within 10 Kpps of the reported throughput. Latency results are accurate up to the microsecond.

Size (B)	64	128	193	449	557	1217	1528
Aligned size (B)	320	448	576	832	960	1600	1856
Total (MB)	640	896	1152	1664	1920	3200	3712

Table 3.4: Memory requirements for 2^{21} mbufs for a range of sizes.

The configuration where hyper-threading is used, results in lower overall performance compared to the non-hyper-threading case. This behavior is caused by the memory hierarchy (see Section 4.2.1). We would see improved performance if memory access was slowing down execution but the Intel® DPDK QoS framework has been optimized to deal with cache misses[10] (referred to as the prefetch pipeline by the Intel® DPDK programmer's manual [Int14i]). The motivation is that the QoS will have to access its data structures from memory instead of cache because the sheer amount of packets being buffered would have evicted the data structures.

Finally, the third configuration where RX and QoS are allocated on the logical core of the same physical core shows different behavior than the other three cases. In this case, not all mbufs (40K out of the 2M) are stored in the QoS hierarchy. In this case, the bottleneck is the RX core which limits the throughput. The value of the Missed Packet Count (MPC) register of the Intel® 82599 10 Gigabit Ethernet Controller is increasing while the `no_mbuf` (which shows the total number of allocation failures) remains zero. Comparing this configuration to the fourth configuration where QoS and TX reside on the same physical core instead of RX and QoS, we can see that by supplying more packets to the QoS, throughput can be increased from 8.65 Mpps to 9.50 Mpps. Performance is still limited by the QoS dequeue operation which now has to block on packet transmission to the TX core. The TX core performance is impacted by being allocated on the same physical core as the QoS.

3.8.3 Impact of memory usage on performance

In this section, we explore the effect of memory usage on performance. We study the effect by configuring the system in unrealistic ways. By knowing why and how memory usage affects performance, we can isolate the behavior. This allows us to understand implications of other architectural choices. The configurations used in this section are only useful for our discussion and should not be used in real-world situations.

Keeping the number of mbufs constant (2^{21}), we configure different mbuf sizes and we take every memory alignment employed by Intel® DPDK into account. The memory alignment is added by the memory pool library (Section 2.3.2). This memory alignment is accomplished by padding mbuf sizes in such a way that memory accesses are spread across different memory channels equally. This means that, for example, mbufs that can hold up to 128 byte packets will consume the same amount of memory as mbufs that can hold packets up to 192 bytes. The actual mbuf size is 448 bytes after alignment in both cases. First, we reserve 128 bytes of headroom for each packet. Second, we need to reserve 64 bytes to store the `rte_mbuf` structure itself. Third, each mbuf needs to points back to its pool (8 bytes). Fourth, the total size of each entry in the pool is padded to next cache line. Finally, padding is added to ensure the total number of cache lines per entry minimizes alignment with number of memory channels and ranks. Table 3.4 shows the amount of padding and the total memory required for a range of mbuf sizes.

[10]The implementation details are not important for our discussion. At the *highest* level, the optimization can be compared with the prefetching technique shown in Section 5.4.

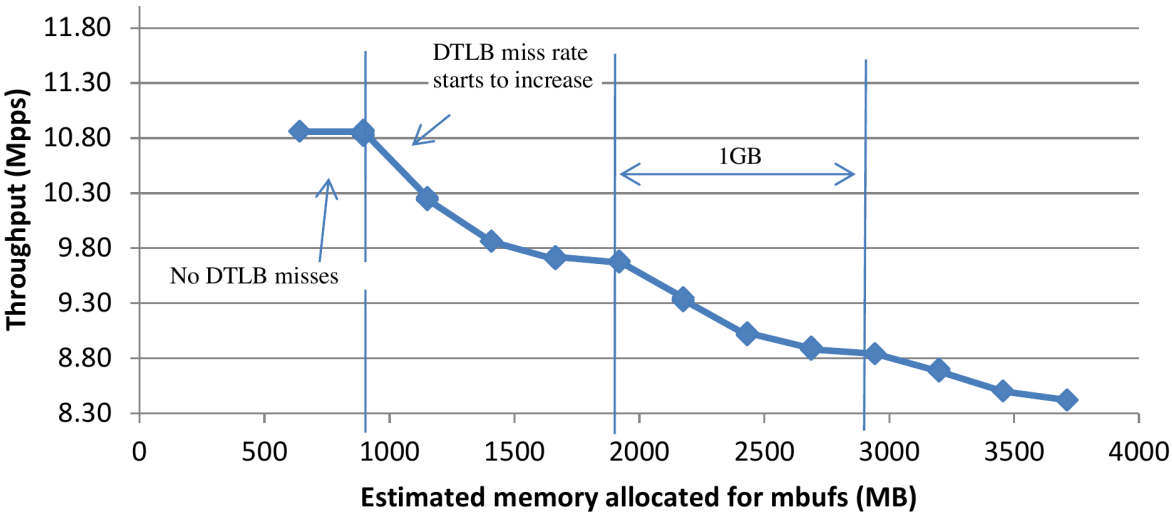


Figure 3.13: Memory usage vs throughput: RX and QoS on the same physical core, TX separate physical core.

For this test, the transparent huge pages feature [Hat] is disabled[11]. The transparent huge pages feature allows the operating system to allocate memory in huge pages without any required intervention from the programmer. With Intel® DPDK, memory is already (explicitly) allocated in huge pages (see Section 2.3.6). If we would allow the operating system to also use transparent huge pages, the results in this test would depend on how and what the operating system is allocating in huge pages without our control. This would in turn make the results harder to reproduce and would add noise to the measurements.

To speed up convergence in throughput, we add code to shuffle the packets at initialization time (done by requesting all mbufs from the memory pool and returning them in a random order). For the QoS, this is not an unrealistic scenario. The QoS essentially reorders packets (and with it, the mbufs) before they are transmitted (and returned to the memory pool) which has the same effect as shuffling has on the packets.

From the Intel 64 and IA-32 optimization guide [Int14g], we know that there are 4 Data Translation Lookaside Buffer (DTLB) entries for 1 GB page sizes. This allows addressing 4 GB of memory through 1 GB huge pages at the same time without a single DTLB miss[12]. If our working set addresses fall in more than 4 different pages, we start trashing the DTLB entries for huge pages.

Impact with logical cores

The first configuration we consider is the one where the RX task and QoS task is running on the same physical core. The results are shown by Figure 3.13. We can see that, up to mbuf sizes of 192 bytes, 10.90 Mpps throughput can be maintained (See Table 3.4 for the memory requirement for a range of mbuf sizes). Comparing with a configuration of 193 bytes (which is aligned to 576 bytes), we see the performance drop to 10.26 Mpps. Using Intel® VTune™ Amplifier XE 2013 [Int14f] with this configuration, we can confirm that no DTLB misses occur in the first configuration (`DTLB_LOAD-`

[11] This is the automatic huge page support that we referred to in Section 2.3.6.

[12] Note that the DTLB is 4-way set associative (meaning it is fully associative for the 1 GB entries). We will go into more detail on set associativity in Section 5.2.

_MISSES.MISS_CAUSES_A_WALK will be zero). However, in the second configuration, we start seeing DTLB misses. In fact, significant portion of the execution time is lost due to DTLB misses.

When we configured for 192 bytes, we were using a single huge page. The memory allocation algorithm in Intel® DPDK found a contiguous region and could furthermore fit all the auxiliary data structures (data structures used beside the mbufs) in the same page.

Comparing to the configuration with 193 bytes, we see significant difference in performance. Due to the alignment, we are asking 1152 MB[13] of contiguous memory for the mbufs alone. The kernel has allocated the first huge page starting from `0x40000000` to `0x7fffffff` and the other huge pages above `0x100000000`. The Intel® DPDK memory allocation algorithm has put the auxiliary data structures in the first huge page and mbufs in the second and third huge page (`0x100000000` to `0x180000000`). We are running the workload on two logical cores on the same physical core which are sharing the same DTLB. The Software Developer's Manual notes that an entry is associated with a logical core which means that a separate entry will be used for each logical core referencing the same huge page doubling the required entries to allow execution without DTLB misses. We would need 6 entries in the DTLB to prevent trashing for this configuration.

The vertical lines shown in Figure 3.13 are spaced 1GB apart. Note that we are estimating the memory allocated *only* for mbufs and we are not considering memory allocated for other data structures. As a consequence, the vertical lines in Figure 3.13 are not aligned to gigabyte boundaries. Each section fits into a single 1GB DTLB entry. Increasing the mbuf size, results in the working set being mapped to more DTLB entries. This, in turn, increases the probability that the required page entry is not in the DTLB (hence the impact on performance).

The cost associated with using more mbufs depends on the probability that the corresponding entry is in the DTLB. For this reason, the performance impact is the highest when we have a few mbufs inside a page (the page is mostly unused) and lowest when nearly all the memory pointed to by the entry is used.

Impact with multiple physical cores

The second configuration we consider is using separate physical cores for each of the tasks. The results are shown by Figure 3.14. We are artificially setting the mbuf size above the Maximum Transmission Unit (MTU) (i.e. we are using mbufs which could hold packets bigger than 1528) for this test. Measuring throughput for progressively larger mbuf sizes, we don't immediately see the impact of DTLB misses. A more sensitive metric is the number of packets buffered inside the QoS data structures. If the dequeue operation slows down, more packets will be buffered when the steady state is reached.

The graph confirms that, as long as we are not trashing the DTLB, performance is not impacted. Starting from mbuf sizes of 1217 bytes, we see the number of packets buffered in the QoS increases. At this point, we are allocating 3.2 GB of contiguous memory. As in the previous case, this block has to be allocated above 4 GB. Now, an additional four pages are required besides the first page containing the auxillary data resulting in a total of 5 huge pages used.

The dequeue operation slows down due to the time spent on handling the misses. The number of packets buffered in the steady state increases to the point where the cost of the DTLB miss is hidden by reduced cost for finding the next outstanding packet. At mbuf sizes of 2048 bytes, the dequeue operation is impacted to an extent where we are seeing packet drops during enqueue operation. This is an example where the time between visits is in some cases higher than the number of packets that arrived and could be enqueued for a specific user.

To close his section, we repeat the performance data from the previous section and compare with the mbuf size set to 64 bytes (without the overhead). The data is shown by Figure 3.15 and Table 3.5.

[13]The memory requirement for a memory pool can be *estimated* by multiplying the number of elements (mbufs) by the size per element.

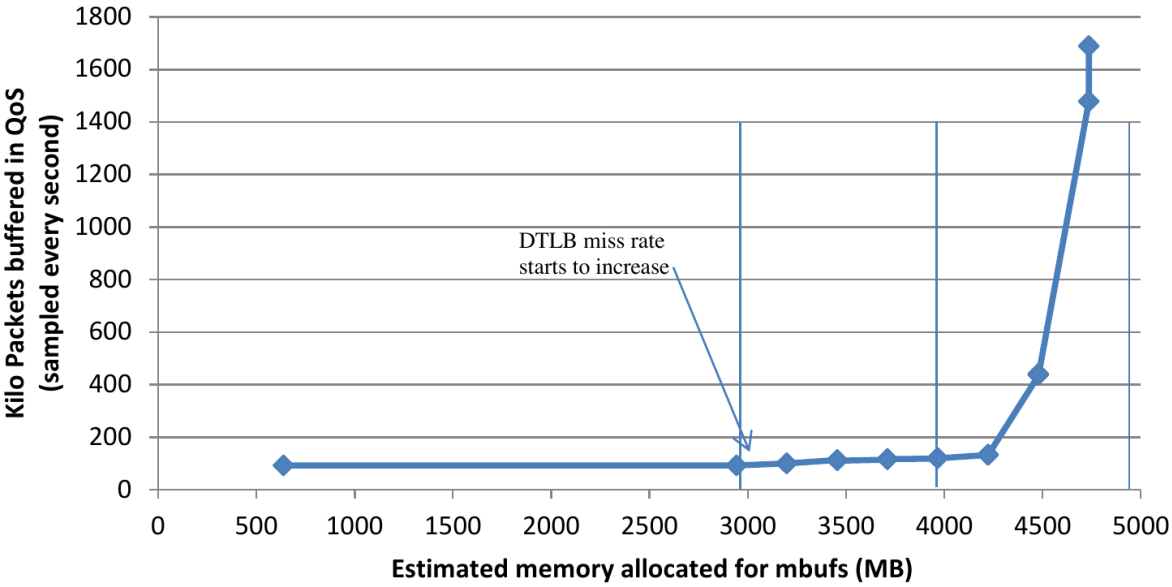


Figure 3.14: Memory usage vs buffered packets in QoS: RX, QoS, TX on separate physical cores.

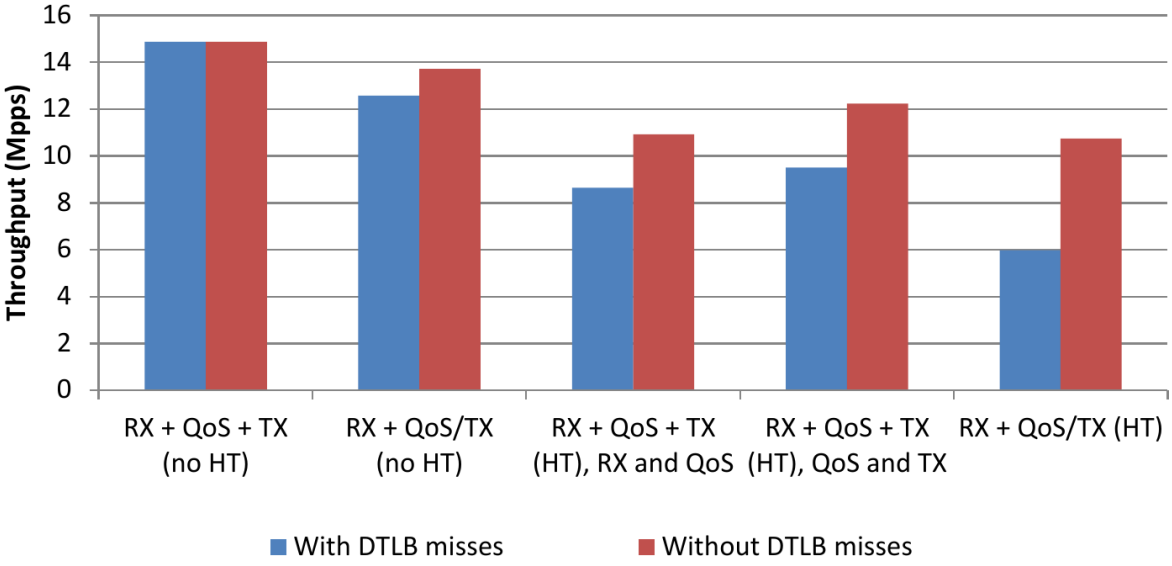


Figure 3.15: Intel® DPDK QoS example with a single QoS instance: Influence of core allocation scheme on performance (with and without DTLB misses).

3.8.4 Performance impact of multiple QoS instances

In this section, we study the effect of running multiple QoS instances simultaneously on the same system. The discussion is relevant for systems where a QoS task is needed for each port or for each direction. Even when packets between flows are not exchanged (i.e. each QoS instance is completely independent of other instances), performance is impacted when running more QoS cores. We don't know the exact reason for the impact on performance but we suspect that L3 cache is playing a

Core allocation scheme	Hyper-threading	Approximate number of buffered packets in QoS (sampled every second)
RX + QoS + TX	No	90K
RX + QoS/TX	No	2096K
RX + QoS + TX	RX + QoS	40K
RX + QoS + TX	QoS + TX	2096K
RX + QoS/TX	RX + QoS/TX	955K

Table 3.5: Number of packets buffered in QoS (without DTLB misses).

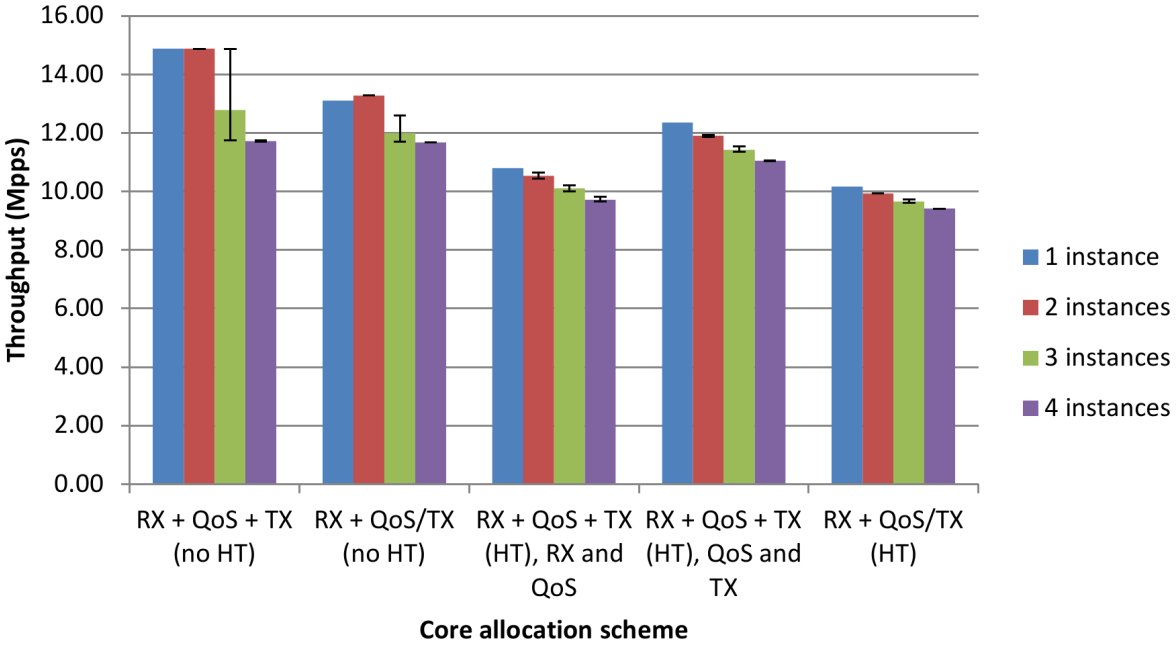


Figure 3.16: Influence of using multiple QoS instances on throughput (averaged over the number of instances, whiskers show the best performing instance and the worst performing instance).

significant role. The cache monitoring feature could be used at least partially to verify this. We will discuss the feature in more detail in Section 5.7.

We make the comparison by listing the throughput and the number of packets buffered for each case. In this particular test, we are using two NICs with two ports. The first two tests (one and two QoS instances) don't run into PCIe bandwidth limitation. We keep the small mbuf size configuration from the previous section and change the maximum number of mbufs from 2 million to 1 million to assure that total memory usage will fit in 2 DTLB entries in all the tests preventing any DTLB misses during these measurements. The effect of using 1 million mbufs instead of 2 million mbufs can be seen by comparing the single instance configuration from Figure 3.16 with Figure 3.15.

Figure 3.16 shows the performance impact for each core allocation scheme. Figure 3.17 shows the number of packets buffered for two cases (RX + QoS + TX without hyper-threading and RX + QoS

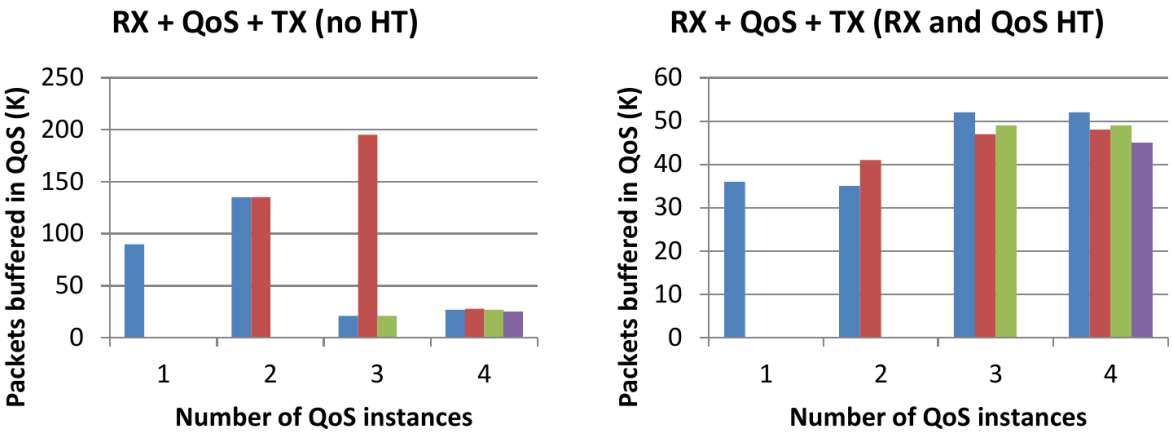


Figure 3.17: Influence of using multiple QoS instances on number of buffered packets.

+ TX with RX and QoS hyper-threaded). With two instances, we avoid PCIe limitations by using two NICs. Starting from three instances, PCIe adds an additional limitation. Two instances suffer from PCIe limitations. The whiskers shown for RX + QoS + TX without hyper-threading in Figure 3.16 confirm that this is the case. This is the reason why the number of packets being buffered does not increase for all instances at the same time when comparing two to three instances in the left graph of Figure 3.17. Not only is the output rate limited by PCIe, but also the input rate and thus the rate that QoS has to handle. The consequence of this is that fewer packets are being buffered by these two instances. Figure 3.17 shows that, without hyper-threading, the number of packets buffered increases at the same time. The cost of the dequeue operation needs to be further reduced by buffering more packets due to the performance impact of using more instances. We have seen the bottleneck in the configuration where RX and QoS are running on the same physical core. The bottleneck causes the number of packets buffered by the QoS to be less compared to other configurations. Figure 3.17 confirms that, with more instances, the number of packets buffered in the QoS remains small as the bottleneck is the RX core. Note the different scales used in Figure 3.17. As expected, less packets need to be buffered if the RX core is supplying packets at a lower rate.

3.8.5 Influence of using a single Queue per pipe

Using the QoS framework provided by Intel® DPDK, 16 queues are allocated for each pipe (4 TCs with 4 queues each). Currently, this cannot be changed through configuration. Changing the code to allow for less TCs and/or queues would require significant effort from the programmer. Internally, the QoS depends on these fixed numbers for optimizations. To fully utilize all 16 queues, the classifier has to be configured to map packets to all TCs and all queues. In this section, we look at the performance implications of using a single queue per pipe. This involves mapping all packets to the first TC and first queue during the classification stage.

Using a single queue has the following advantages. First, the cost for finding outstanding packets is reduced. Second, less memory is touched and less stress is put on the cache (inactive queues are skipped). Third, the same queue is revisited quicker. Forth, latency is reduced. Using only 1 out of 16 queues has similar advantages as configuring for fewer users. The disadvantages are that fewer packets can be buffered before packets are dropped and that the memory access pattern is influenced.

Using Intel® Performance Counter Monitor tools, we can compare configurations of using 16 queues with using 1 queue. When all queues are used, the memory access pattern results in 591 MB, 562 MB, 587 MB and 562 MB per second of memory read bandwidth on the four memory channels

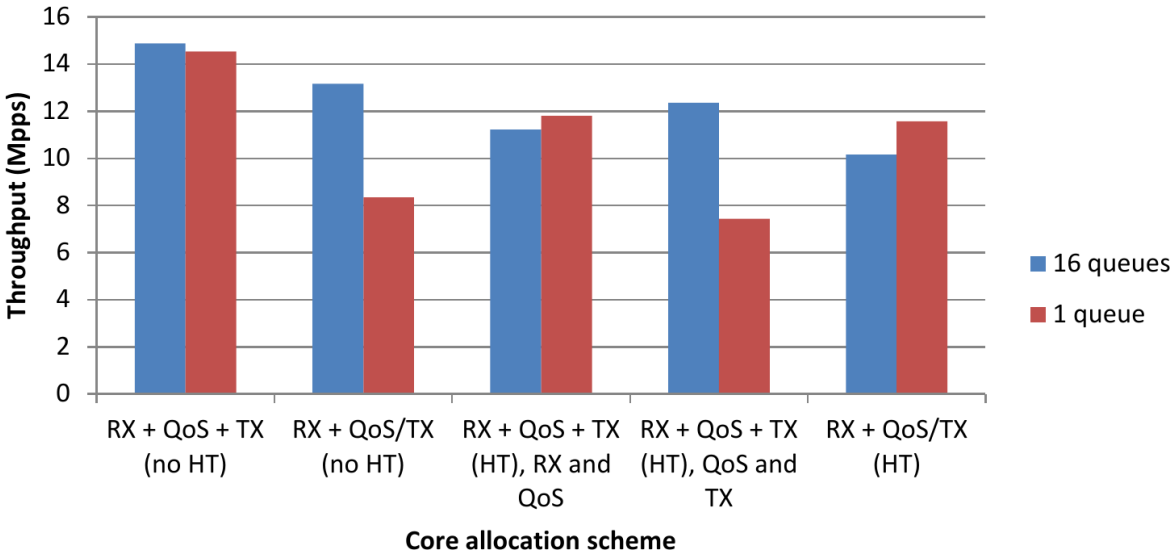


Figure 3.18: Comparison of throughput between using 1 queue vs. 16 queues.

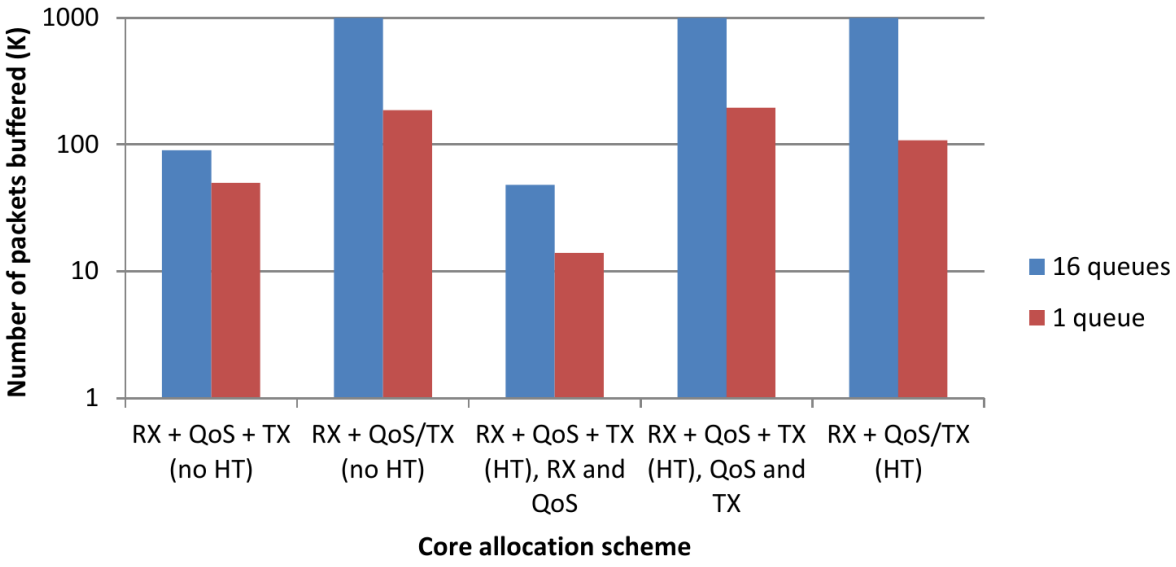


Figure 3.19: Comparison of number of packets being buffered between using 1 queue vs. 16 queues.

respectively. Using a single queue changes the memory read bandwidth to 842 MB, 453 MB, 478 MB and 454 MB respectively. Figures 3.18 and 3.19 show that, depending on the core allocation scheme, using a single queue can be beneficial.

3.8.6 Impact of number of users (pipes) on QoS performance

Now we look at how the number of pipes influences performance. We only consider configurations where the number of pipes is a power of two. This is a requirement by the QoS implementation and use cases have to round up to the next power of two. We consider the core allocation scheme where

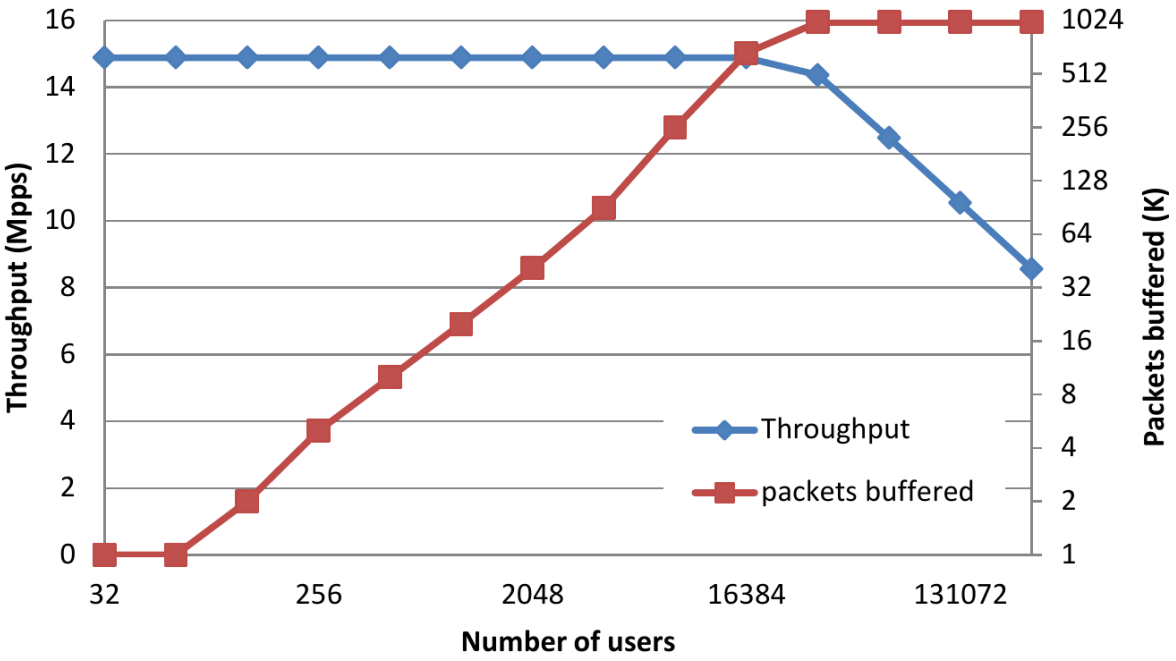


Figure 3.20: Influence of the number of users on throughput (blue) and packets buffered (red).

each task is allocated on a separate physical core. Finally, we change the classifier to mask out and map more than 2^{12} pipes (12 bits in the inner VLAN tag) by also using bits from the outer VLAN tag.

As we expect, the number of packets that are buffered increases when more users are configured (see Figure 3.20). Revisiting the same user takes more time and thus more packets will be buffered. When we hit the maximum number of mbufs (1 million), throughput starts to decrease. We would need to buffer more packets in the QoS to lower the cost/packet sufficiently to maintain throughput, but new packets cannot enter the system because all the mbufs are residing in the QoS.

3.8.7 Classifying using a table

It is here that we move away from the Intel® DPDK QoS example code and continue the tests with the vBRAS prototype application. The main difference relating to the QoS functionality is that a table lookup is used during the classification step allowing the system to map arbitrary packet contents to any user. Another notable difference is that the classifier maps to a single queue per pipe. The configuration is made more flexible to allow integration with the vBRAS. Note that the QoS can be used without any vBRAS functionality and can be altered to work with input traffic that is different from what is used for the vBRAS by changing the classification implementation.

We revisit the different core allocation schemes and look at the performance when we use a table. Both configurations extract 12 bits from the CVLAN but one passes the value through a table that is configured to map the value onto itself. The results are shown by Figure 3.21. We conclude that there is no signification difference in performance between using a table and mapping packets directly to users. However, this does not prove that using a table does not consume resources. It only shows that for this test case, there was no impact (i.e. no bottleneck is introduced).

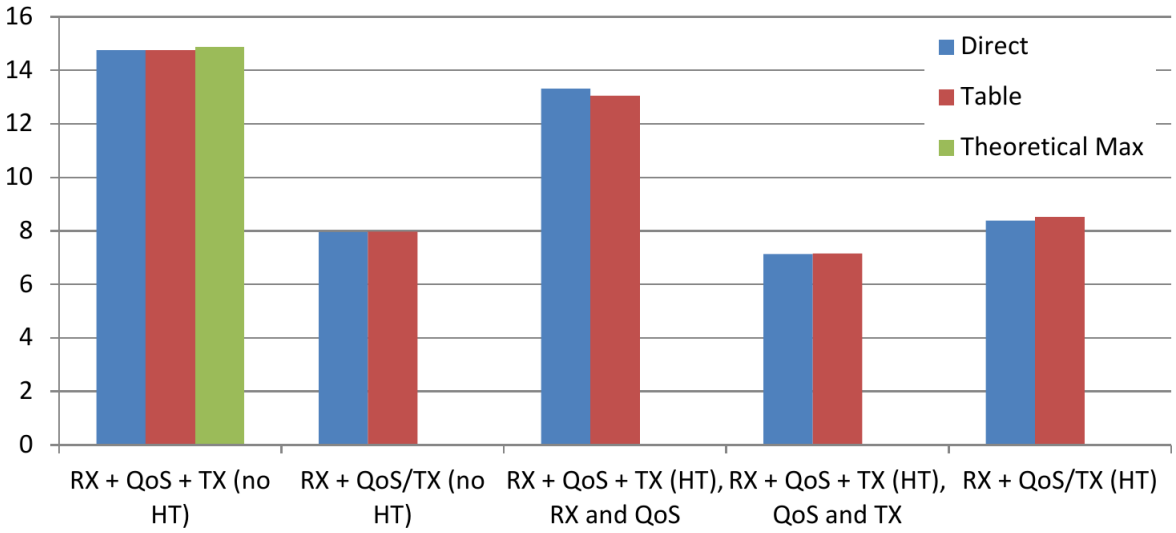


Figure 3.21: Mapping packets with and without a table.

3.9 vBRAS prototype application with QoS

In this section we focus on the vBRAS prototype application in combination with QoS. First, we show performance numbers for QoS handling the vBRAS traffic without any vBRAS functionality in the pipeline. Second, we show performance results of vBRAS without QoS. Third, we add QoS to the vBRAS configuration in either the download or the upload direction. Finally, we configure QoS both in the upload and download direction.

We can calculate the memory requirement r, for a given number of users u, and a given queue size q. The real memory requirement will be higher due to padding and alignment. Each subport and each pipe both require 64 bytes. Each queue table entry requires 4 bytes. Each queue entry requires 8 bytes. A single bit is stored per queue in the active queues bitmap. Additionally, we need 1024 bytes for the data structures used to control the prefetching. For convenience, Equation (3.1) can be used to calculate the memory requirement. For a more detailed list of all data structures and their sizes used internally in the QoS consult Table 4 of the Intel® DPDK programmer's guide [Int14i].

$$r = u(128q + 130) + 1152 \tag{3.1}$$

Recall that the use case requires 32768 users and that the queue size is 128. Using Equation (3.1), we calculate that, at least, 516 MB per QoS instance will be allocated even though only one queue will be used. To make sure that the QoS data structure does not cross huge page boundaries (which causes two page entries being used in the DTLB), either we would need to change the memory allocation implementation or we would need to allocate dummy data between consecutive calls to QoS initialization.

3.9.1 QoS without vBRAS functionality

In the final configuration, we will use two logical cores per QoS instance with a total of four instances (two per direction). As we will see in the following sections, in our use case we use the third allocation scheme for the upload direction and the fourth allocation scheme for the download direction. The classifier will use the QinQ tags to identify users by passing the tag through a lookup table, but recall

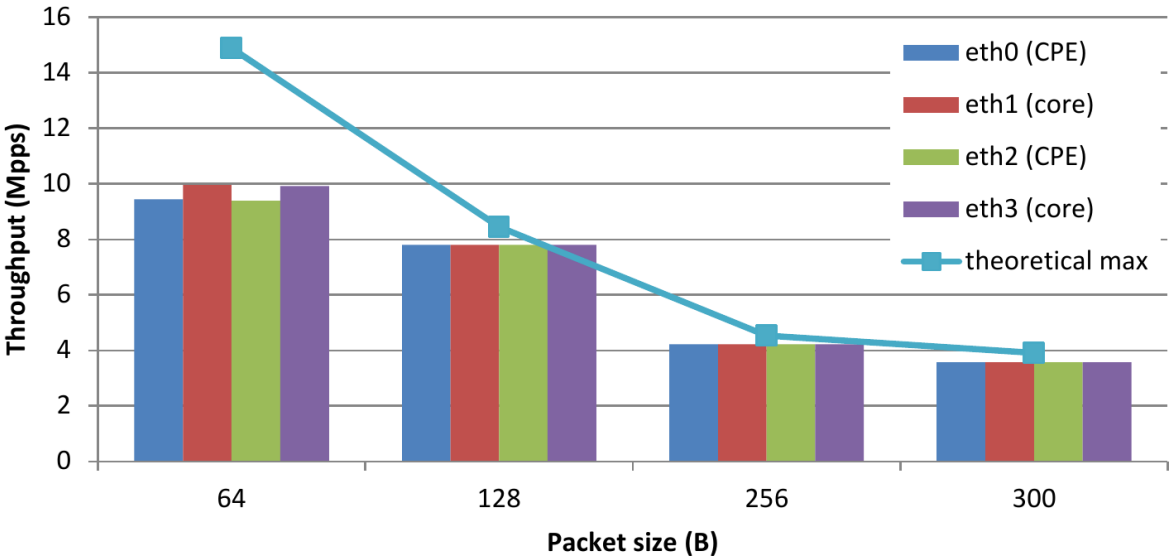


Figure 3.22: Throughput for progressively larger packets per interface for QoS without vBRAS (note that the packet sizes stay constant in this test as there is no vBRAS functionality).

that only the CPE packets contain QinQ tags. This test does not contain any vBRAS functionality. Hence we use CPE traffic for both the upload and the download direction.

Figure 3.22 shows the results. Each bar represents the throughput measured at a given interface, i.e. the packet rate sent out on that interface by the SUT. With the use case requirements (32768 users per QoS instance) in combination with the core allocation schemes, line rate cannot be reached with 64 byte packets as shown by the measurements results. The discrepancy between the throughputs measured when handling 64 byte packets is explained by the core allocation scheme we use (see previous sections).

3.9.2 vBRAS without QoS functionality

Comparing to RA 3.0, a few alterations have been made to the vBRAS application. The most notable change is that the routing has been incorporated into the worker thread. Another important change is to connect both the CPE and core network on the same NIC instead of connecting the CPE on one NIC and the core network on the other NIC. The packet rate on the CPE side is below line rate because of the encapsulation performed by the vBRAS (24 bytes are added to each packet). In the original configuration, the maximum bandwidth utilization was theoretically 100% (core network size) and 85.7% (CPE side) from the perspective of the NICs. By assigning the interfaces as described, the total bandwidth utilized is higher due to the improved balance and reduced impact of PCIe limitations. Other optimizations related to improving throughput with large number of mbufs will not be detailed here. For these optimizations, refer to Chapter 4[14]. The revised architecture is shown in Figure 3.23 and performance data is shown in Figure 3.24. Note that the theoretical maximum is determined by the size of encapsulated packets (see Section 2.5.1). While the original vBRAS architecture was already capable of handling 128 byte packets at line rate, the throughput at 64 byte has improved by 1.5 Mpps per interface[15]. With the "x-y" notation for packet sizes we are referring to tests where the

[14]Without these optimizations, using a vBRAS configuration with large number of mbufs required for QoS reduced the throughput to 4-5 Mpps per interface.

[15]Performance improvement for a vBRAS configuration with a large number of mbufs is 3-4 Mpps per interface.

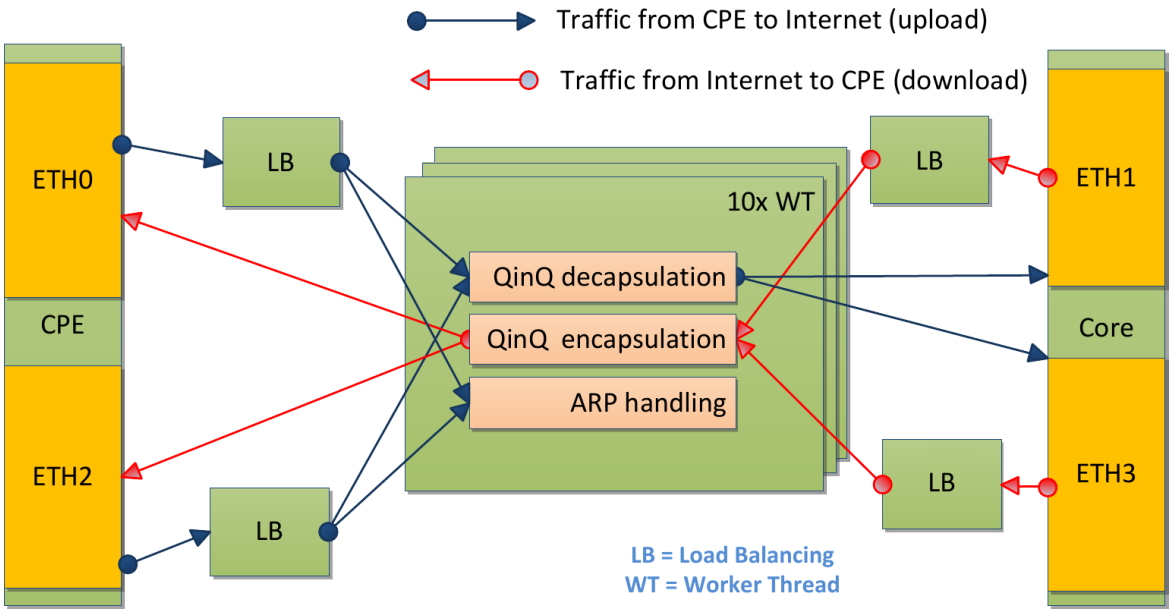


Figure 3.23: Revised vBRAS architecture

CPE traffic consists of x byte packets and the core network traffic consists of y byte packets. To put the system under maximum load (see Section 2.5.1), we use the 64-78 byte configuration[16]. Note that we are also considering the 64-88 byte configuration beside configurations for bigger packets. The CPE packets are effectively 64 bytes without any padding in this configuration. When we look at latency, we will first measure the overall latency of the system as opposed to the latency for one user. The test equipment used to measure overall latency requires inserting time stamps and tracking information in the packets. When we use `pkt-stat` (see Section 2.7) to measure latency for each user separately, we will focus on the smallest packets (64-78 byte configuration). The performance numbers are for a configuration with 10 worker threads (5 physical cores) and 4 load balancers (2 physical cores) for a total of 14 logical cores. The remaining 8 logical cores are left idle in this configuration. These will be used for QoS.

3.9.3 vBRAS with QoS functionality and throughput

In this section, we look at the performance of a configuration that combines both QoS and vBRAS functionality. Figure 3.25 shows the architecture for the vBRAS with QoS added in both the upload and the download direction. We separately give results for configuring QoS only in the upload direction, only in the download direction and in both directions (see Figures 3.26 to 3.28). We call these configurations "upload QoS", "download QoS" and "Full QoS" respectively. As expected (see Figure 3.22), the results show that the QoS tasks are the bottleneck of the system. Additionally, in the "upload QoS" and "download QoS" configuration, the impact on performance in one direction allows for an increased throughput in the other direction which was mainly limited by PCIe bandwidth. All these configurations have similar architecture to the one given in Figure 3.25. From Figure 3.25, we see that the QoS has been positioned as close as possible to the customers. Below, we discuss the design choice for placing the QoS task at those positions in the pipeline.

[16]Even though the smallest packet at the CPE side is 54 bytes, padding is added to make these packets 64 bytes in size as required for Ethernet. The padding is removed before encapsulation.

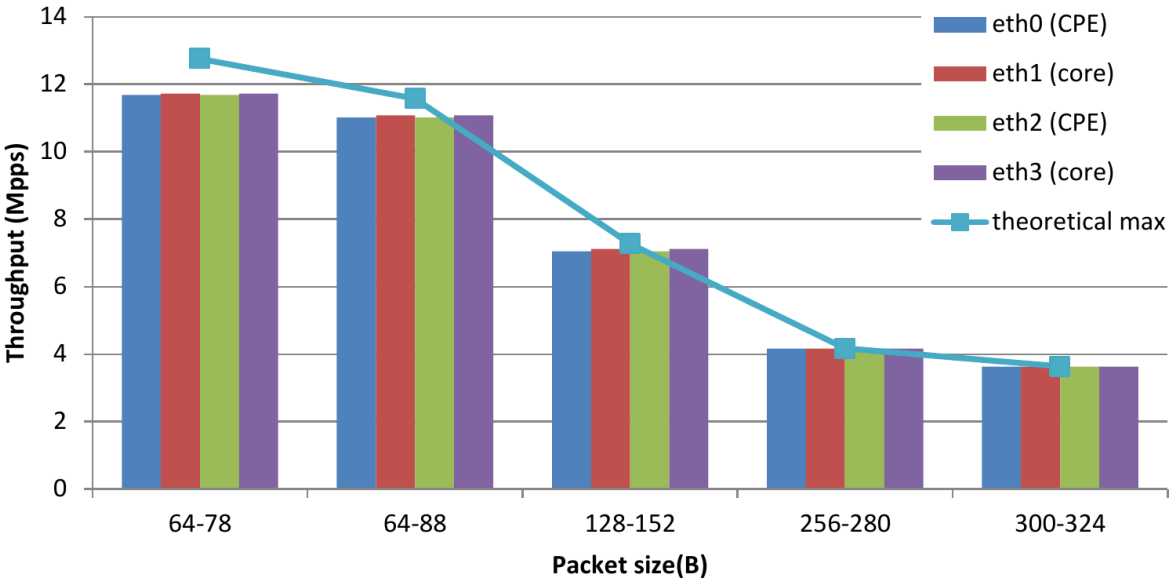


Figure 3.24: vBRAS throughput without QoS.

Consider if we were to use two QoS instances (one per core network facing interface) at the end of the pipeline in the upload direction (i.e. after the packets have been routed). Now, take a user that is transmitting traffic which is routed to both the first and the second output interface. To limit traffic of such a user would require us to exchange messages between the two QoS instances to assure that the user's traffic stays within the configured rate. Clearly, this is not feasible with acceptable performance. Another architectural choice would be to configure a QoS instances in each of the worker threads and pass the packets through the QoS before packets are routed. This would require a total of 10 QoS instances and would inherently split users into partitions. Taking performance impact of running multiple QoS instances into account, we have not explored this path. The advantage of placing the QoS at the beginning of the pipeline is that nonconforming traffic is dropped by the QoS and prevents any unnecessary load on the system. The disadvantage is that the precise control of the QoS on the timings and ordering of the packets is partially lost.

For the download direction, we have put the QoS instance at the end of the pipeline. For the same reason as noted for the upload traffic, putting the QoS instance before the load balancers would require exchanging messages between the two QoS instances as traffic destined for a given user may enter the system at either of the core network facing interfaces. At the end of the pipeline, traffic belonging to a given user is aggregated at the correct output interface. The aggregation is achieved by keeping track behind which interface a given user resides. This information is stored in the MAC table and is consulted by the worker thread. The worker thread additionally performs the classification of the packet for the QoS after it has restored the QinQ tags.

Finally, we change the input traffic stream to contain 32K, 16K, 8K and 4K users. We can reduce the number of active users by half if we set the most significant random bit in the SVLAN and the second most significant random bit in the GRE ID to be fixed. Changing the most significant bits will allow the load balancer to keep balancing the input stream equally.

Figure 3.29 shows the performance results. As we expect (see results from Figure 3.18 and previous sections), reducing the number of active users reduces the load on the system. For the first four tests, we have configured each user to be limited at 1 Mbps. For the last two tests, we have changed the rate to 4 Mbps. By reducing the number of active users by half, we double the rate at which each user

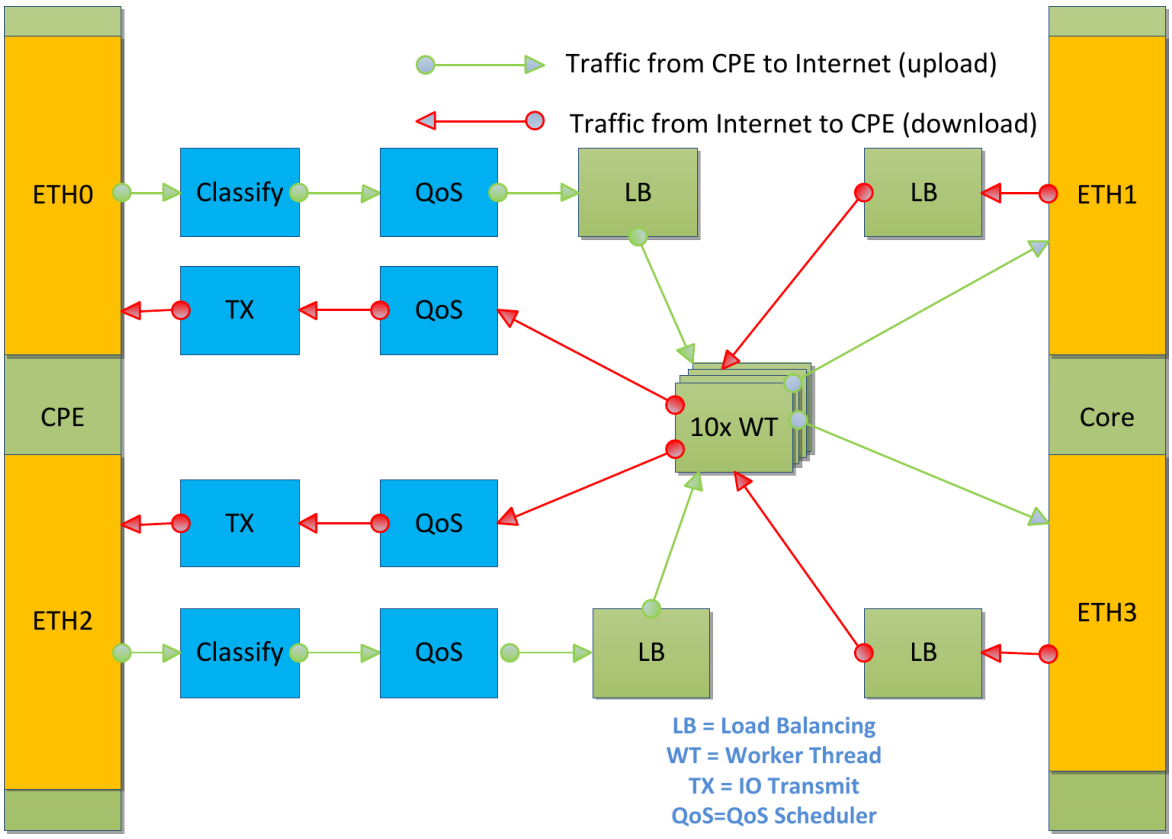


Figure 3.25: vBRAS + QoS architecture. QoS in upload and download direction.

is transmitting. Starting at the first measurement, each user is transmitting at 305.18 Kbps while with 4096 active users, the rate is 2441.40 Kbps. As long as QoS can maintain the throughput, we can increase the rate at which users are limited without impact on performance. We have not measured the maximum packet rate that can be maintained at a small number of users as our use case focuses on 32768 users per QoS instance.

In the two measurements with 4096 active users, the users were transmitting at 2441.40 Kbps. In the first measurement, this is more than twice the configured rate. In the second measurement, the rate at which users are limited is configured above the traffic rate of each user. As expected, the difference in throughput between the two measurements confirms that QoS is limiting the rate. The discrepancy between the throughput in the upload and the download direction can be explained by the different core allocation schemes in use. The increase in difference between the measured throughput of the two core allocation schemes indicates that the scheme used in the upload direction is limited more by the number of users than the scheme used in the download direction. There is influence between the allocation schemes as we are using two ports per NIC. Consequently, the actual increase in performance with less users could be higher than what we measure.

3.9.4 vBRAS with QoS latency

Before we discuss the latency results for the vBRAS configuration with and without QoS, we discuss factors that influence the maximum and minimum latency. The discussion is also relevant for other

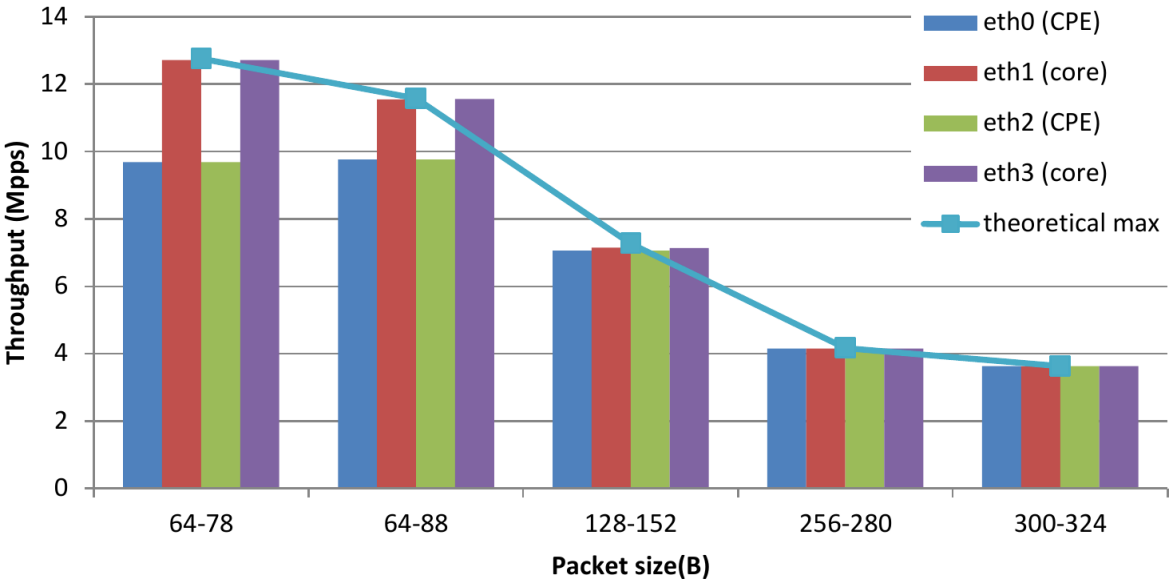


Figure 3.26: vBRAS + QoS throughput with QoS only in the download direction.

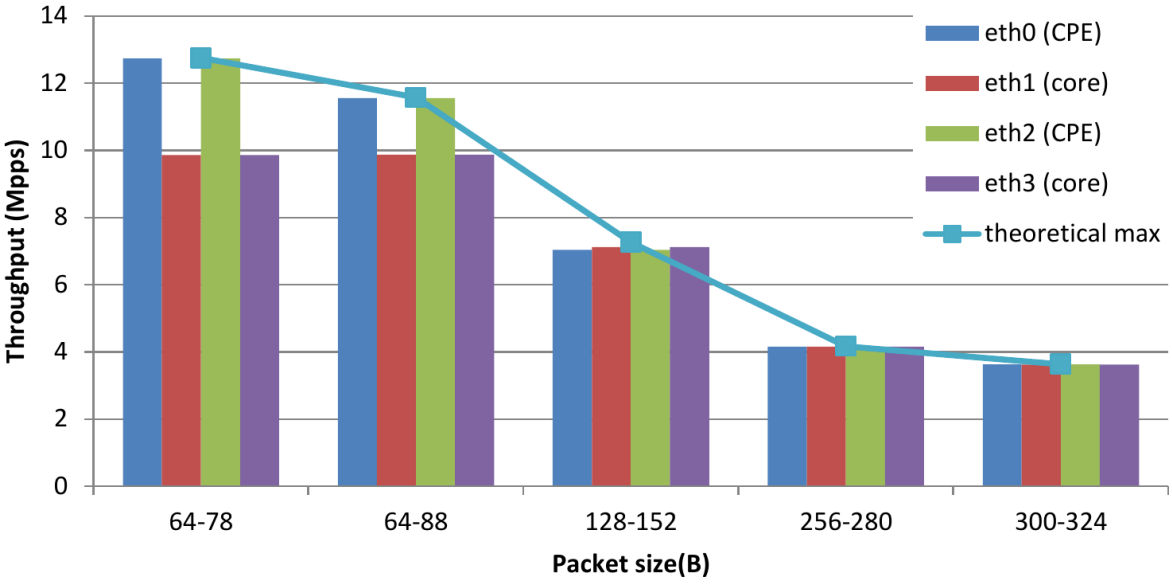


Figure 3.27: vBRAS + QoS throughput with QoS only in the upload direction.

packet handling applications.

The first thing we note is that applications using Intel® DPDK fall in to the store-and-forwarding category. Compared to cut-through forwarding, the whole packet has to arrive before it can be handled and this brings with it the latency to receive the whole packet. For example, the minimum latency for a 64 byte packet is 67.2 ns at 10 Gbps (See Section 2.8 for more details).

Secondly, we are handling packets in bulk. The latency for each packet is increased by the time needed to transmit all packets in a bulk. The reason for this is that the first packet needs to wait

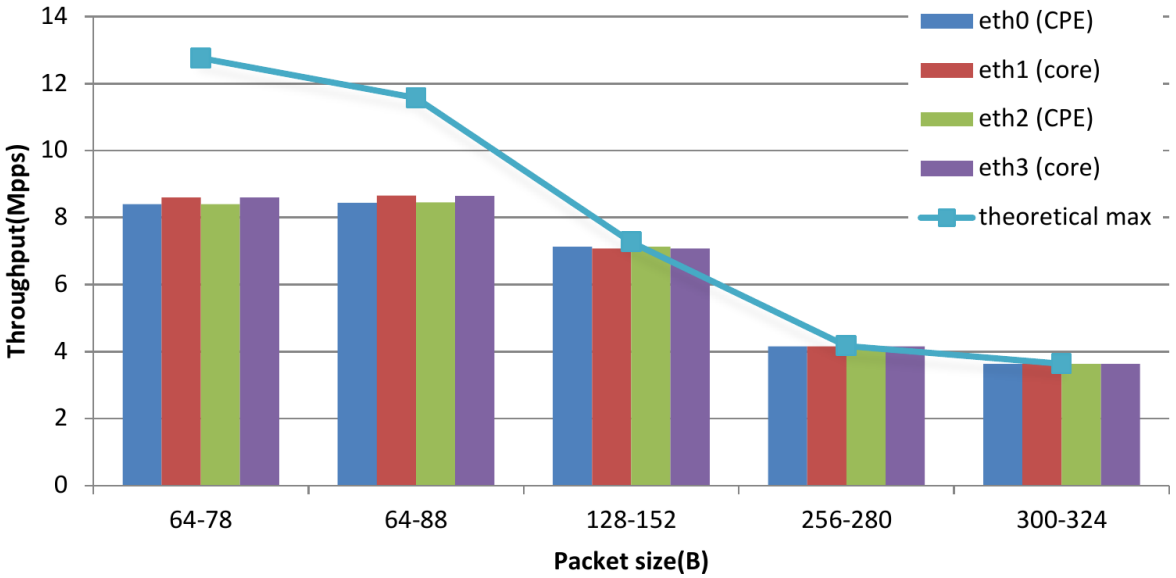


Figure 3.28: vBRAS + QoS throughput with QoS in both directions.

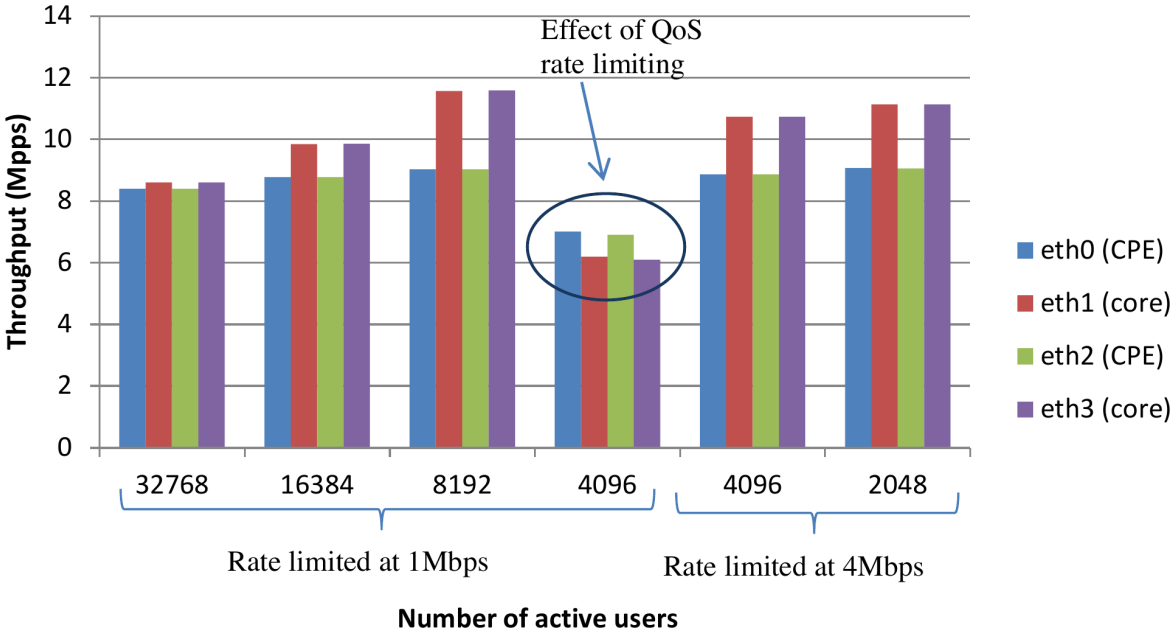


Figure 3.29: vBRAS + QoS throughput with QoS in both directions and changing number of active users.

before all packets in the bulk arrive before being handled and the last packet needs to wait before all packets are transmitted before it is transmitted. For example, the minimum latency with a bulk size of 64 and 64 byte packets is 4.3008 μs (See also Section 2.9).

Finally, the buffers inside the Ethernet controller can add to the latency. From the Intel® 82599 10 GbE controller datasheet [Int14h] we find the memory size for the receive buffer and the transmit

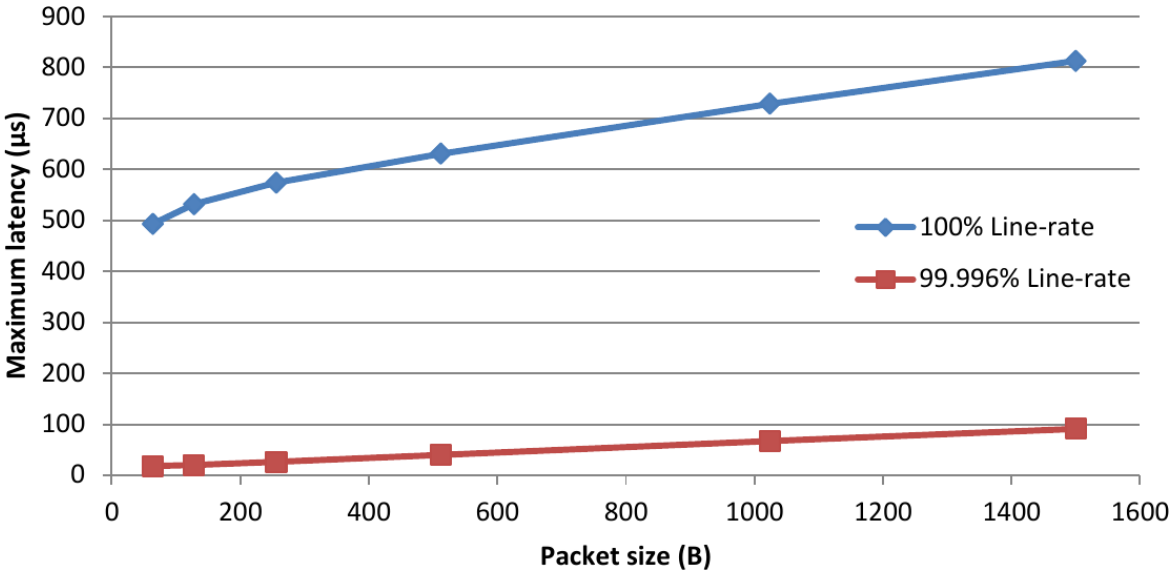


Figure 3.30: Maximum latency measured for basic forwarding (128 RX descriptors and 128 TX descriptors).

buffer per port (512 KB and 160 KB respectively). Arriving packets are buffered by the Ethernet controller and each packet is stored with auxiliary information requiring overhead. For this, no intervention from software is needed. At the next level, the Intel® DPDK receive descriptors are used. On the transmission side, the application transfers mbufs to the transmit buffer through Intel® DPDK. The minimum IPG is 9.6 ns at 10 Gbps but depending on the error introduced by the manufacturing process, the IPG for transmitted packets will be bigger. In turn, this lowers the transmission rate of the Ethernet controller.

Using a configuration where packet drop is prevented (i.e. all packets entering the system will be transmitted), the receive buffers in the Ethernet controller will be full due to the receive rate being higher than the transmit rate. The result is that packet latency increases and the value of the MPC register will increase. For example, if the input rate (as measured by the packet generator) is 14880952 pps, and the output rate is 14880705 pps, the ierrors statistics will report an average value of 247 pps.

Using basic forwarding, we measure the latency for different packet sizes transmitted at line rate (as measured by the traffic generator). The results are shown in Figure 3.30 where each data point shows the maximum latency. At line rate, the receive buffers (512 KB),the transmit buffers (160 KB) and the system buffers (receive descriptors + transmit descriptors + bulk handling buffer) are full. If we measure the maximum latency when we configure the traffic generator at 99.996% of line rate[17], we can see the latency introduced by the system itself instead of the buffers in the NIC. When we are measuring latency, we have to take this factor into account. The same situation occurs when throughput is limited by PCIe. At 99.996% of line rate, the latency is acceptable. Most of the latency is caused by bulk handling as discussed in Section 2.9.

The discussion above stresses the need to use RFC 2544 [BM99] (Benchmarking Methodology for Network Interconnect Devices) for testing latency. In practice, we should search for the input rate at which no packets are dropped by the SUT. If we were to measure latency under maximum input rate, we would essentially be measuring the total size of the buffers in the system.

[17]We use 99.996% of line rate as at this rate no packets are being dropped (as reported by the ierrors statistics).

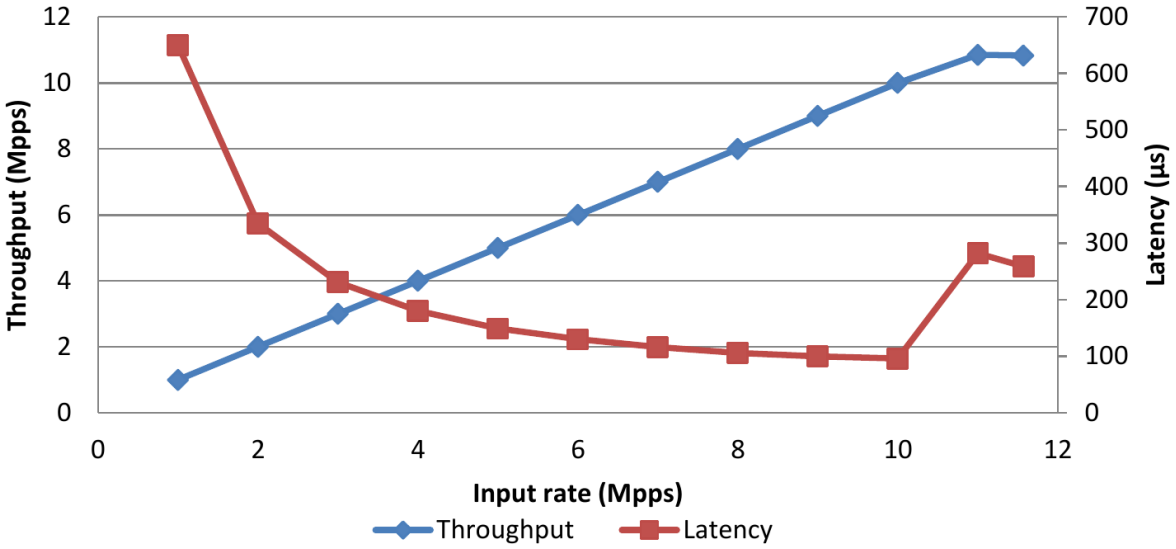


Figure 3.31: Average latency (red) and throughput (blue) for vBRAS without QoS. The increase in latency at 11 Mpps is caused by congestion (increased queue occupancy).

We look at latency in the vBRAS configuration with and without any QoS functionality. The input rate is variable while the packet size is kept constant. Figures 3.31 and 3.32 show the average latency. An increase in latency is expected when QoS is used (due to more packets being buffered). This is confirmed by comparing the latency shown by Figure 3.31 with the latency shown by Figure 3.32. Note the different scales used in Figures 3.31 and 3.32. The 64-88 byte packet traffic i used for this test. Additional bytes are needed by the packet generators to track packets when the built-in latency measurement features are used. Without these additional bytes (i.e. with 64-78 byte packets), the only way packets could be tracked is through the use of existing fields which is not supported by the packet generators. When the system becomes congested, latency increases suddenly. The system is congested when the input rate is higher than the output rate (See the throughput shown in Figures 3.31 and 3.32). As suggested by the throughput measurements of vBRAS with and without QoS (See Figures 3.24 and 3.26 to 3.28) congestion occurs at lower rates with QoS. At this point, the queue occupancy rises due to bottlenecks. The measurement at 11.57 Mpps shows a reduction in latency in Figure 3.31 compared tot he measurement at 11 Mpps. When new packets can be handled by the system, these packets arrive more quickly at higher rates. Assuming there are no other bottlenecks at higher rates, the latency reduction would be smaller for each consecutive measurement.

The increase in latency for low rates is caused by handling packets in bulks. By checking periodically if packets have been processed, we can reduce the latency for small packet sizes. This technique is used to detect if no packets have been processed during a predefined interval. The interval is called the drain timeout. If no packets have been sent since the last drain timeout occurred, we start processing any available packets. This is in contrast to *always* waiting for a complete bulk of packets. Waiting for a bulk results in higher latency for rates below line rate (see Section 2.9). Additionally, we guarantee that even a single packet can progress through the pipeline[18]. If the drain timeout is too small, performance can be impacted at high input rates. The configurations in Figure 3.31 and Figure 3.32 used a drain timeout of 2.2 ms. Using a drain timeout of 22.2 μs and setting the input rate at 8 Mpps, the latency increases from 10.68 ms to 11.05 ms with QoS enabled. Handling smaller bulks in the QoS does not permit to fully benefit from all the prefetching and increases the number of

[18]Packets are not stored indefinitely in the system.

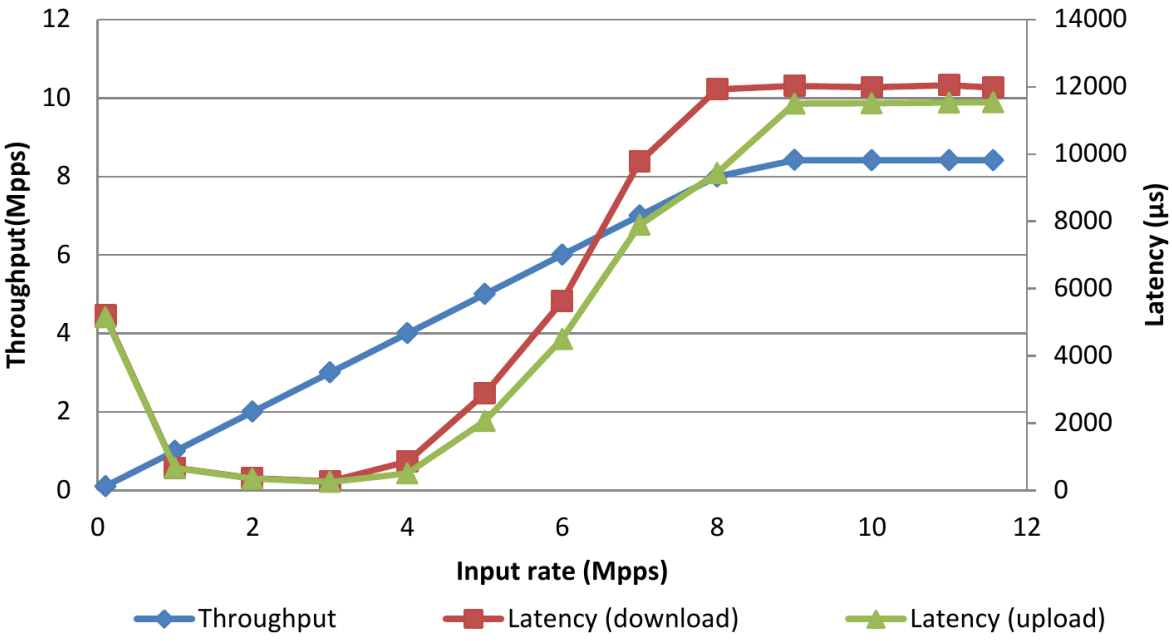


Figure 3.32: Average latency for vBRAS with QoS in both directions.

packets that are buffered due to the performance impact. On the other hand, the average latency of 4.74 ms (shown on the left in Figure 3.32) is reduced to 54 μs when we transmit a single packet per second. Note that the latency is higher than the drain timeout as the pipeline consists of multiple cores. The actual optimum depends on the specific workload and the SUT.

Assuming that the input rate for a given core is constant and that we can find the rate at which a core can handle packets, we can calculate the drain timeout that should not have any impact at that rate. For example, if a bulk size of 64 is used and 5 Mpps throughput can maintained, a drain timeout of 12.8 μs should not have any impact. If the input rate would then drop below 5 Mpps, the system would start draining packets. Draining packets requires more cycles but, at the same time, less packets to be handled.

Next, we show that bigger packets introduce latency if QoS is disabled and reduce latency if QoS is enabled. The amount of latency introduced depends on the drain timeout. In essence, if the drain timeout is high, all packets inside a bulk have to arrive before they are passed through the system. As noted in the previous paragraph, a high drain timeout is desirable to improve throughput at small packet sizes. The configurations for which the results are shown in Figure 3.33 used a drain timeout of 2.22 ms. For the case where QoS is used, latency for progressively larger packet sizes decreases to the point where it is determined by the time it takes the vBRAS portion of the pipeline to handle a packet. The input traffic was limited to 86.4% of line rate in the download direction and 67.2% in the upload direction. For the 64-88 byte packet configuration, this translates to 10 Mpps.

3.9.5 QoS with vBRAS

Finally, we study the quality of the QoS. We only consider the upload direction as that direction has extra steps in the pipeline after the packet is handled by the QoS scheduler. We look at the distribution of latency. The results are obtained by capturing two streams. One stream contains CPE traffic sent to the SUT while the other contains what the SUT sends out to the core. The streams are then processed offline through an external application (see Section 2.7). The same application is

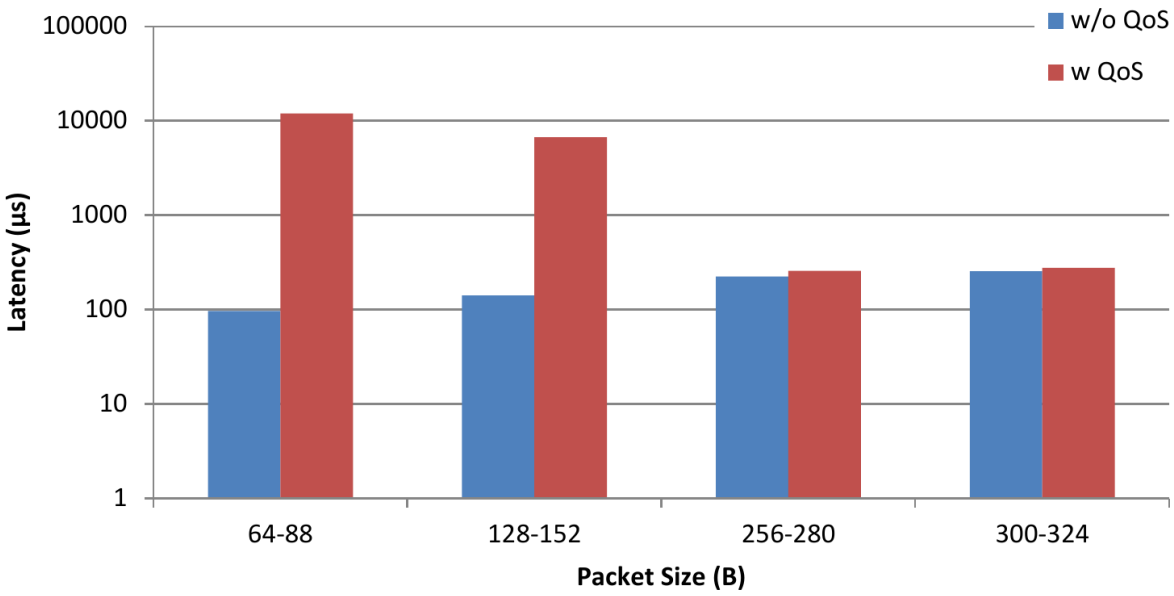


Figure 3.33: Latencies for different packet sizes for vBRAS with and without QoS.

used when fine grain results are required. The application uses existing fields (i.e. source/destination UDP port) and is aware of routing and the mapping between VLANs and GRE ID's. This allows it to track every single packet in the vBRAS/QoS workload. If the clocks used in the two streams cannot be synchronized, the packet delay variation metric should be used instead (RFC 3393 [DC02]).

Figure 3.34 and Figure 3.35 show the distribution for latency when the TC period is set to 240 ms and Figure 3.36 and Figure 3.37 show latency distribution for TC period set to 40 ms. The user was limited at 1 Mbps. When the user is oversubscribing, the effect of the TC period becomes visible. Packets are enqueued and dequeued while credits are available but when all credits have been consumed, packets wait in the queue until TC period time passes. This causes the two peaks visible in Figure 3.35 and Figure 3.37. The credits awarded when TC period expired allowed the user to transmit at twice the allowed rate during half the TC period. This explains why we see a latency of 120 ms.

The latency distribution that we measure for TC period set to 40 ms is more desirable (average latency is lower compare to TC period set at 240 ms), but the inherent latency is higher. In this scenario, the number of credits awarded after TC period expired is not enough to dequeue all packets. As a consequence, packets will always have to wait for (at least) one TC period to expire. The number of TC periods that packets have to wait in the queue depends on the queue size and the number of packets dequeued after TC period expires.

Not only is the latency high for packets that wait until the TC period expires, but the average latency of the first peak is also higher than the average latency when the user is not oversubscribing. Once again, this demonstrates how the system behaves for a user that is oversubscribing.

We now look at latency when the system is loaded at 8 Mpps with 32768 users on all interfaces and each user is sending an equal share (20.51 KBps) of the traffic. The metric of interest is the gap size which we define as the number of packets transmitted between transmitting two packets of the same user. The resolution of the measurements is limited due to the maximum number of packets that can be captured by the traffic generator before memory is filled. The result is shown in Figure 3.38 for an arbitrary user (Figure 3.39 shows the peak between 184000 and 186400 with smaller bins).

If the precision inside the QoS is not high enough, i.e. multiple packets arrive between two

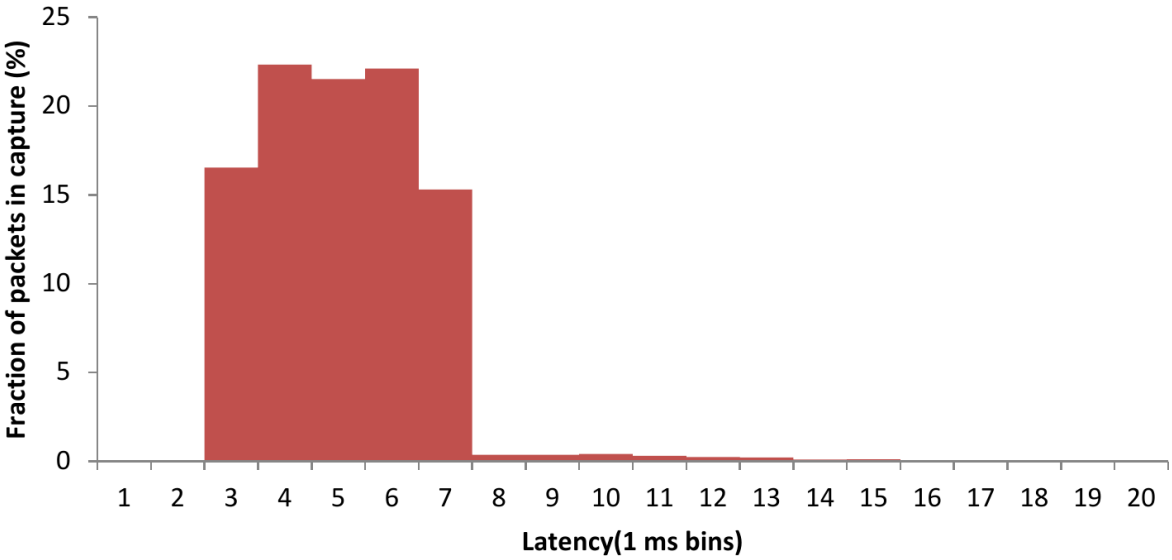


Figure 3.34: Latency distribution for single user sending at 1 Mbps, (TC period is 240 ms).

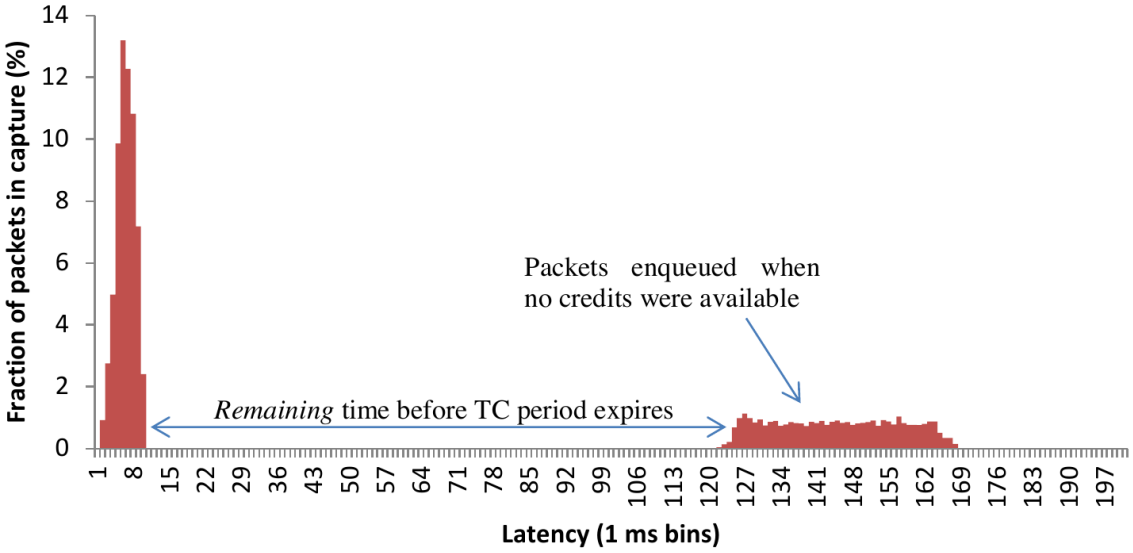


Figure 3.35: Latency distribution for single user sending at 2 Mbps (TC period is 240 ms), 11518/23024 packets dropped.

consecutive checks of the queue belonging to the same user, multiple packets will be dequeued. This explains the first peak in Figure 3.38. Note that other users have the same behavior and this is the cause for the other peaks. Figure 3.40 shows a plot of each packet and when it was received relative to the first packet. The slope gives the throughput for this specific user. The peaks can be related to the plot.

Looking at the packet gap distribution when a single user is oversubscribing on one interface (i.e. transmitting at twice the allowed rate) while the total aggregate traffic of the other users is set at

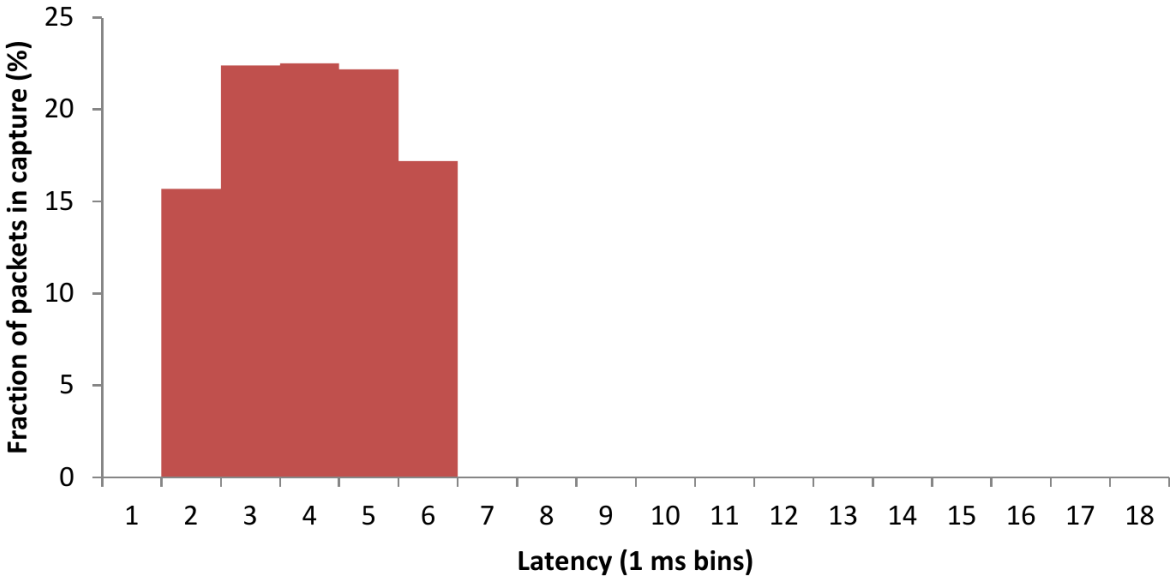


Figure 3.36: Latency distribution for single user sending at 1 Mbps (TC period is 40 ms).

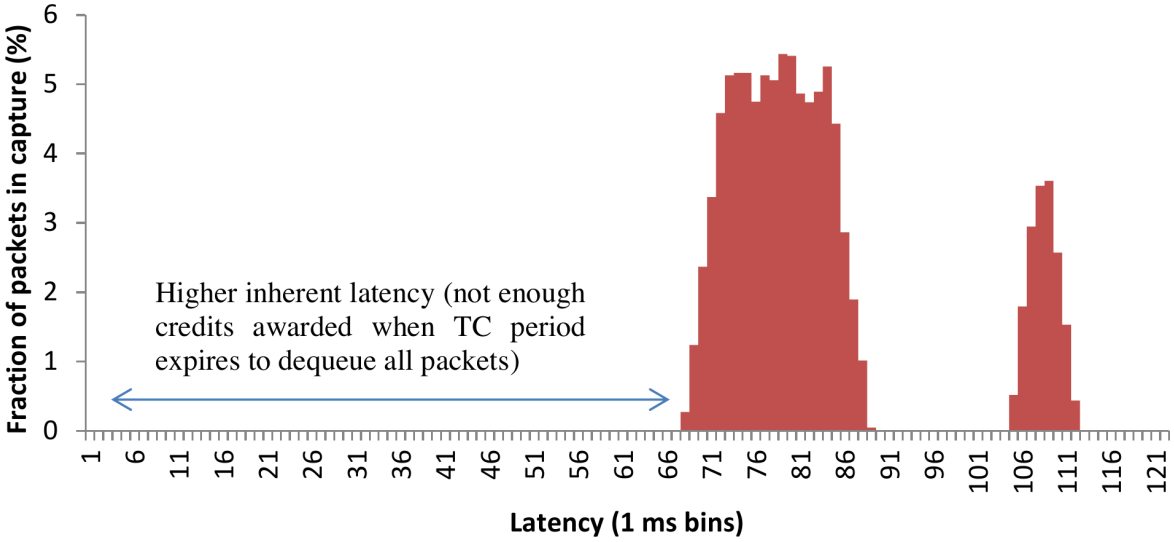


Figure 3.37: Latency distribution for single user sending at 2 Mbps (TC period is 40 ms), 8744/17230 packets dropped.

8 Mpps, Figure 3.41 shows behavior with 40 ms TC period.

Figure 3.41 shows the gap distribution for a user that is oversubscribing. Looking at the moments when the packets are transmitted, we see that a bulk of packets are transmitted (at this time, credits where awarded due to TC period expiration) followed by a gap which is approximately the size of the TC period. This is expected behavior for a user that is continuously oversubscribing. With TC period set at 40 ms, theoretically, there should be 25 bursts per second. After TC period expires, a burst is transmitted. Between each burst, we expect to see approximately 320K packets (8 Mpps/25 bursts per second). From the perspective of the user (1 Mbps) we expect to see 60 packets per burst

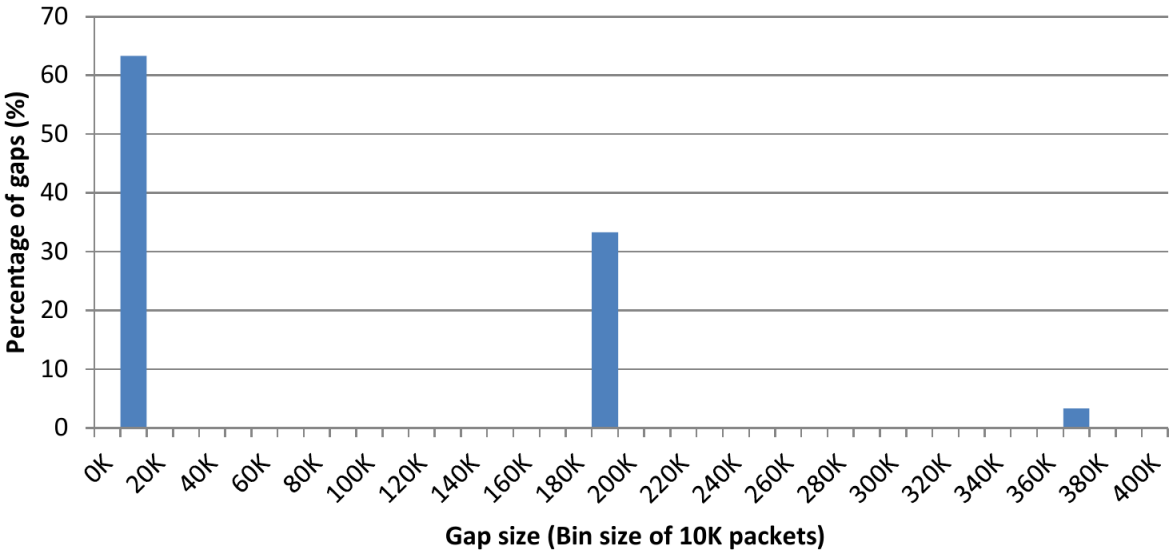


Figure 3.38: Gaps (number of packets belonging to other users between 2 consecutive packets) for a single user.

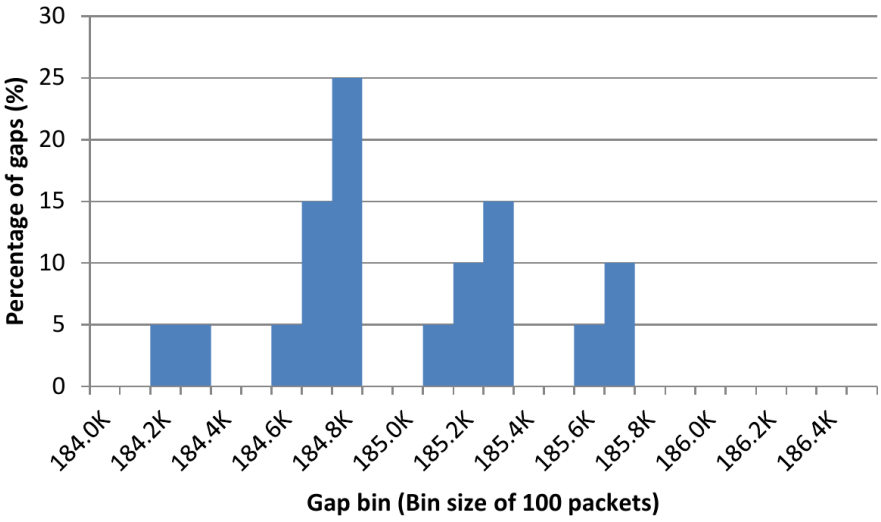


Figure 3.39: Details from Figure 3.38 in the range 184.0K-186.4K.

(1.4881Kpps/25 bursts per second). Statistically, these 60 packets will be spread equally between the two core network facing interfaces due to routing. Therefore, we expect two peaks containing approximately 96.67% and 3.33% of the packets respectively. The gaps reported for 29 out of 30 packets is relatively small as those packets are sent when there were enough credits. The gap reported for 1 out of 30 packets is relatively large due to the time between bursts causing the second peak.

Setting the TC period to 1 ms (this is the minimum) would, in theory, limit the burstiness in the output traffic but a lower TC period implies an increased cost per packet for the dequeue operation (more queues will have to be visited to find the requested number of packets). In turn, this implies an increased number of packets being buffered and consequently an increase in latency. There is no real

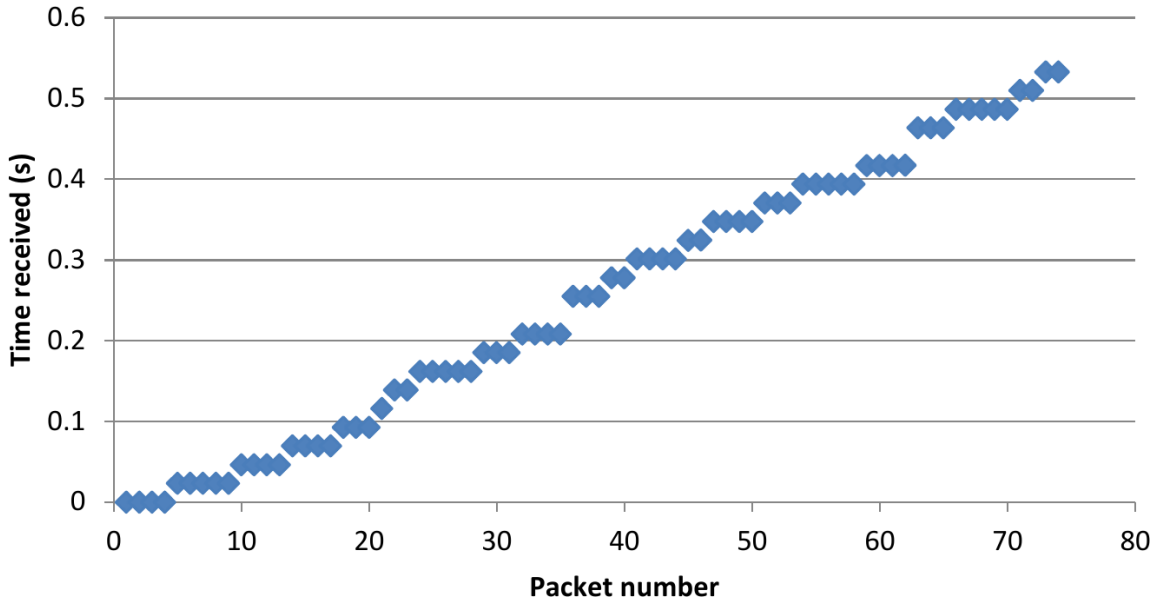


Figure 3.40: Timestamp when packets are received for a given user (rate shown by slope).

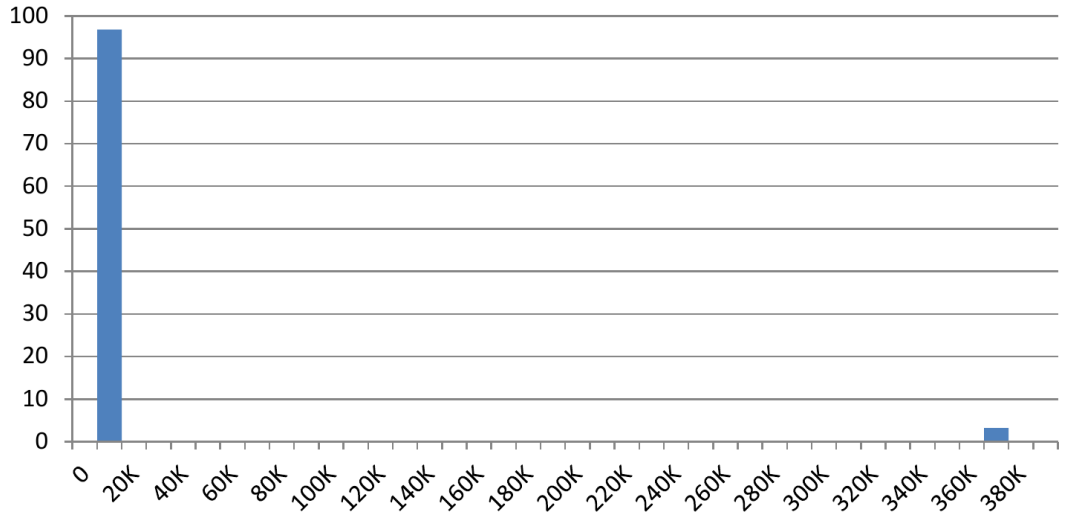


Figure 3.41: Gap distribution for single user oversubscribing (transmitting at 2 Mbps, rate limited at 1 Mbps) while system is loaded at 8 Mpps on other interfaces with small packets.

token bucket with tokens at the level of the TC and tokens can't accumulate in contrast to the token bucket used for the user. Credits will be rewarded only after 1 ms. Being late, a bigger fraction of the tokens is lost compared to having TC period set at 40 ms. This results in packet loss. The throughput using this configuration drops to 3.8-3.9 Mpps. As a last measurement, in Figure 3.42 we show results for the same configuration from Figure 3.41 but using 128-152 byte packets to demonstrate how the peaks shift.

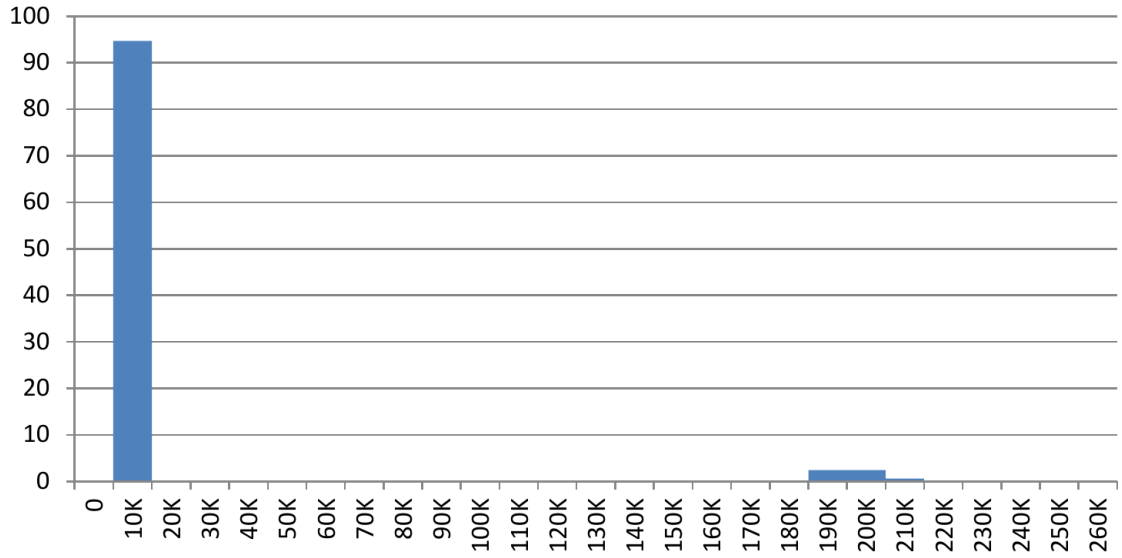


Figure 3.42: Same configuration as in Figure 3.41 with packet size set to 128-152 bytes.

3.9.6 Performance per task

Performance per logical core is given in Tables 3.6 and 3.7 for the vBRAS without and with QoS functionality respectively. The input rates are set at 12.75 Mpps in both the upload and download direction (i.e. the system is overloaded). While all active cores are polling their input queues, conventional tools like `top` will report 100% utilization. They cannot be used to measure utilization. To estimate idleness, we count the number of cycles not spent usefully by taking the difference of time stamps taken before and after operations that can fail[19]. If the operation fails, the cycles are accumulated as "idle". The idle cycles are report in the column "% idle cycles" in the Tables 3.6 and 3.7. From Table 3.7 we can conclude that the bottleneck in the whole system is the QoS. This is expected from the measurements of QoS without vBRAS and vBRAS without QoS. As multiple tasks are executed within the system, a task can influence the performance of another task. For example, it could be possible that adding QoS reduces the performance of the existing vBRAS workload to a degree that the bottleneck is in the vBRAS part of the system. Table 3.7 confirms that this is not the case.

In real-world scenarios, the theoretical limit of the packet rate is not reached. Reaching theoretical limits would require all customers to use the smallest packets at the same time in both directions. Comparing the throughput of our implementation with the average packet size reported for IPv4 packets in 2013 (770 bytes) [Hic], we conclude that it is feasible to move to a software based solution. Note that the average packet size only serves as an indication for the actual traffic. However, it does show that links saturated with the smallest possible packets is unrealistic. The statistics also report that links are never saturated (highest reported utilization in 2013 by the same source is 53.2%) which further supports our conclusion. Also note that (unless noted otherwise) all users were active to put the most load on the system. Finally, for our tests, turbo frequency was disabled in order to reduce noise in the measurements. In a real-world implementation, turbo frequency should be left enabled as it could further improve performance.

[19] A detail needs to be taken into account when using time stamps to estimate idleness. Reading the time stamp itself introduces overhead. The overhead is estimated by taking the difference between the time measured in the following two situations. In the first situation, the time stamp is read twice and the difference is taken. In the second situation, the time stamp is read three times and the difference of the first and last time stamp is taken.

Core function	**Dedicated physical core?**	**Note**	**Load handled by each core/thread (in Mpps)**	**% idle cycles**	**instances**
Upload load balancer	No	HT core used by another upload LB.	11.68 Mpps	8%	2
Download load balancer	No	HT core used by another download LB.	11.68 Mpps	11-12%	2
Worker thread	No	HT core used by WT. Using 65K CPE.	• 2.3 Mpps up • 2.3 Mpps Down	1-2%	10

Table 3.6: vBRAS (without QoS functionality) performance per logical core.

Core function	**Dedicated physical core?**	**Note**	**Load handled by each core/thread (in Mpps)**	**% idle cycles**	**instances**
Upload load balancer	No	HT core used by another upload LB.	8.5 Mpps	25%	2
Download load balancer	No	HT core used by another download LB.	8.17 Mpps	17%	2
Worker thread	No	HT core used by WT. Using 65K CPE. Classifies packets in download direction.	• 1.7 Mpps up • 1.6 Mpps Down	6-7%	10
Upload QoS	No	HT core used by Classifier.	8.5 Mpps	0.5%	2
Download QoS	No	HT core used by TX core.	8.2 Mpps	0.5%	2
TX core	No	HT core used by download QoS.	8.2 Mpps	39-40%	2
Classifier	No	HT core used by upload QoS.	8.5 Mpps	22-23%	2

Table 3.7: vBRAS (with QoS functionality) performance per logical core.

Chapter 4

Optimization techniques

4.1 Introduction

The purpose of this chapter is to point out possible bottlenecks and solutions that may improve the performance of the system. The platform used was Intel® Xeon® Processor E5-2697v2 (codenamed Ivy Bridge) and Intel® 82599 10 Gigabit Ethernet Controller (codenamed Niantic). Techniques described in this chapter are not specific to this platform and could be applied to applications using Intel® DPDK running on similar Intel® Platforms. Actual performance is highly dependent on the type of application. Hence no performance numbers are included in this chapter. *All* the techniques described in this chapter were used to improve performance of the vBRAS with QoS implementation.

The Intel® DPDK Programmer's Guide gives some optimizations and tips for writing efficient code. This chapter discusses other optimizations and gives motivations behind the proposed techniques. Whether the techniques are applicable or not, depends on the type of application. For each optimization mentioned in this chapter, an example system and its behavior are discussed. This should help the reader decide if the specific optimization is relevant for his/her implementation.

We start by discussing the most important optimizations in Section 4.2. These optimizations fall in the "locality improvement" category [Dre07]. The impact of these optimizations can be as high as most packet processing applications are heavily memory bound. Next, we list several optimizations. Each optimization focuses on specific aspects of the system. We describe "no-drop" rings and how they define behavior in systems implementing this technique (Section 4.3), bottlenecks introduced by PCIe bandwidth limitations and how we can alleviate these bottlenecks (Section 4.4). We continue with optimizations that are related to locality but that can be used without significant changes to the system architecture (Sections 4.5 and 4.6). Next, we show how application code can be optimized (Sections 4.7 to 4.12). Finally, we refer to BIOS settings relevant for packet processing applications (Section 4.13).

An optimization might increase throughput but it could also increase latency. The two metrics should be studied as one because of the interplay between the two. In some cases, the gain in throughput might outweigh the increased latency. Thus, the choice to employ a specific technique or not has to be made on a case by case basis.

4.2 Locality Improvement

Generally, packet processing applications don't use CPU intensive algorithms but instead fall into the category of I/O bound applications. Handling a packet involves table lookups, encapsulations, changes to internal states of the system... Clearly, these applications can be classified as memory intensive. As in all contexts where there is significant communication between the processor and

memory, performance improvements are possible by improving locality. This means that we keep data close to where it is needed.

4.2.1 Taking memory hierarchy into account

Access user data from one core

Consider the example shown in Figure 4.1. Green boxes represent a core. Pink boxes represent tasks handled by a core. For example, a worker thread is assigned with two tasks (QinQ encapsulation and QinQ decapsulation). The thick arrows show one of the possible paths a packet can take. The two example packets belonging to the same user and their paths are shown.

Looking at the blue path, we can see the load balancer distributes packets to worker threads depending on the QinQ tag in the packet. The load balancer uses the QinQ tags to index into the worker thread table. The worker thread table contains the worker thread ID for each QinQ tag. The packet in the given example is sent to the first worker thread. The worker thread uses the QinQ tags as a key for the "QinQ to GRE ID" hash table. The resulting GRE ID will be used in the packet. Next, the worker thread adds an entry to the MAC/ARP/QinQ hash table using the GRE ID together with the source IP as a key. The entry describes the customer to which the packet belongs. Packets in the other direction (shown in red) will retrieve information from the same table. Consider a packet that arrives at the load balancer. The load balancer extracts the GRE ID and calculates which worker thread will handle the packet. The GRE ID and the destination IP are used as a key during the lookup by the worker thread. The resulting information is used to restore the QinQ tags and set the destination MAC address (= MAC address of the customer).

Consider a customer to which we associate a QinQ tag and a GRE ID. The system described in Figure 4.1 will translate QinQ packets to GRE tunneled packets by replacing the QinQ tag (or GRE ID) with the GRE ID (or QinQ tag). Knowing that, for a given user, an entry in the MAC/ARP/QinQ hash table will be accessed, we can assure that only one core accesses that entry. This is accomplished by always sending packets that belong to a given user (either following the red path or the blue path) through the same worker thread. In the system shown in Figure 4.1 we accomplish this by initializing the `wt_table` with the correct values. Using the same worker thread avoids having to use locks when accessing the MAC/ARP/QinQ hash table. Another advantage is that the table entries could be in cache of the core that the worker thread is running on. The impact on cache would be higher in applications that read/write more user related data than the example shown in Figure 4.1.

Consecutive cores in the pipeline

Through `sysfs`[1], we can find which logical cores are sharing Level 1 and Level 2 caches. Depending on the workload, reordering the cores in the pipeline might influence the performance. The cache hierarchies shown in this section are for the Intel® Xeon® CPU E5-2697 v2 @ 2.70GHz. Only part of the whole hierarchy is shown (6 out of 24 cores). On this processor, the difference between the core id of two hyper-threads is 12 (for example, core 5 and core 17 are two hyper-threads on the same physical core). Sharing L1 and L2 cache have two consequences. First, if the two cores are performing tasks that use different memory locations, the L1 and L2 cache size from their perspective is essentially divided by two. Second, a core can benefit from the low access latency of the L1 and L2 cache when it uses data previously fetched during processing done by the other core.

Figure 4.2 shows an example path that a *stream* of packets could follow when the pipeline consists of three cores[2]. In this example, core 5 receives packets from the physical interface. It passes the

[1] `cat /sys/devices/system/cpu/cpu0/cache/index3/size` can be used to find the level 3 cache (`index3`) size from the perspective of core 0. Information about other cores, caches and the topology can be queried in a similar manner.

[2] Note that the size of each cache level (L1, L2 and L3) depends on the processor. The size of the L1 D-cache, L2 cache and L3 cache is 32KB, 256KB and 30 MB respectively for the Intel® Xeon® CPU E5-2697 v2 @ 2.70GHz.

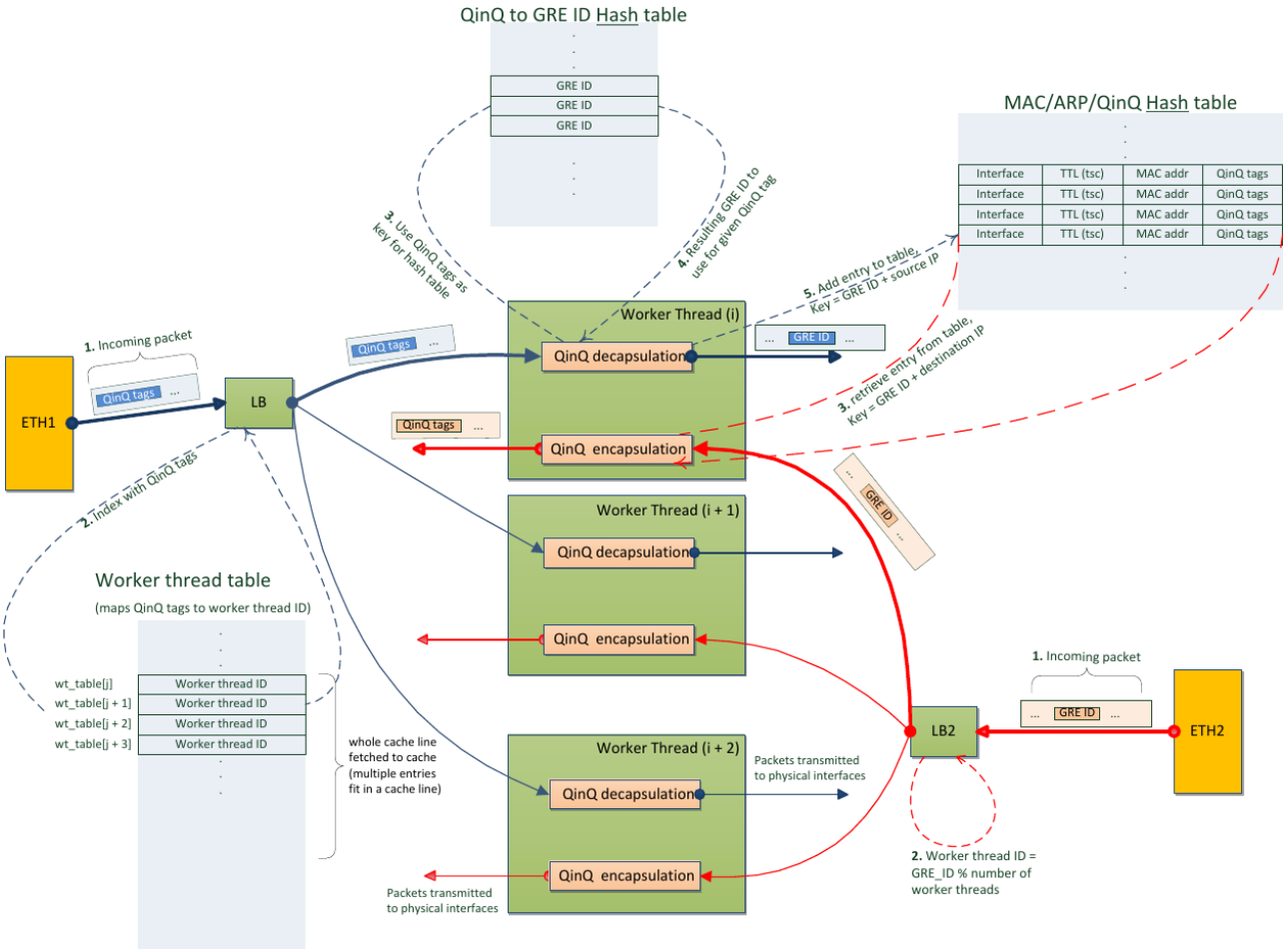


Figure 4.1: Path of packets in vBRAS (simplified). Two (red and blue) paths are shown.

packet to core 17 which in turn passes the packets to core 6. Finally, the packet exits the system after core 6 has transmitted it through a physical interface. Before a core can access the data of the packet, the data needs to be the Level 1 cache of that core. In the example given by Figure 4.2, core 17 could benefit from the packet being in its Level 1 cache as the last core that handled the packet was core 5.

Figure 4.3 shows an alternative pipeline. The same cores were used. Only their order in the pipeline was changed. Assuming that each core has the same workload, the path shown in Figure 4.3 is less optimal. Using the second path, we might run into a caching issue. Consider a workload where core 6 changes packets. This means that the packet that core 17 had in Level 1 cache will be invalidated. The packet will need to be fetched into the Level 1 cache for the *second time*. Another example workload consists of three cores that perform unrelated tasks. The first and second core perform cache intensive tasks while the last core performs less memory intensive tasks. In this scenario, the first and second core in the pipeline shown by Figure 4.3 are using the cache nearly exclusively. In the pipeline shown by Figure 4.2, the first two cores are sharing the cache. From the perspective of the cores, the cache is smaller. As a consequence, more cycles are spent on data access[3]. The workload needs to be considered when choosing one pipeline over the other.

[3]We will go into more detail on caching in Chapter 5.

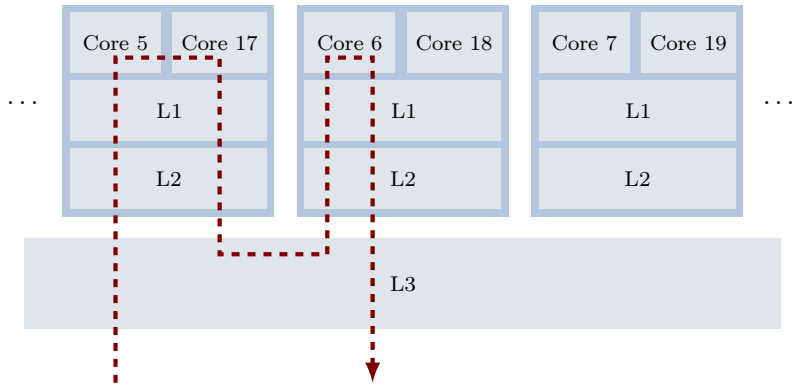


Figure 4.2: Cache hierarchy superimposed with an example pipeline (shown by the dashed arrow).

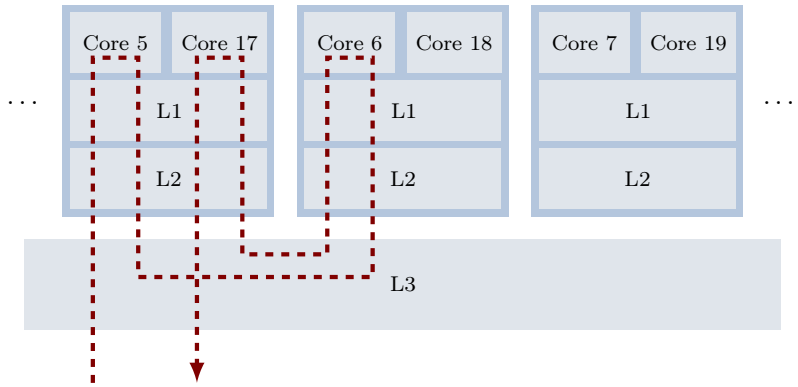


Figure 4.3: Cache hierarchy with an alternative pipeline.

A second example is shown in Figures 4.4 and 4.5. In this example, we have two parallel paths. Similar paths can exist in systems where packets arrive through two physical interfaces or through two queues on an interface. Using the same number of cores as in Figure 4.4, the configuration shown in Figure 4.5 could be more efficient. The reasons are similar to the ones described for the pipelines from Figures 4.2 and 4.3. The core allocation for QoS and the classifier in one direction and the TX core and QoS in the other direction (see Figure 3.25) use the pipeline shown by Figure 4.5 instead of the pipeline shown by Figure 4.4.

4.2.2 Hash table (patch)

The hash implementation provided by Intel® DPDK does not store any data. Instead, it is used to map values to indices. The values are first hashed using (by default) CRC32. The CRC32 instruction for 4 byte values introduced with the SSE 4.2 extensions is used by Intel® DPDK. We advise against using other hashing functions (for example JHash). Those hashing functions are comparatively slower than CRC32. The 4 byte result (called the signature) is used to index the hash table. At each index, a number of signatures and keys are stored (configured through the bucket size during hash table initialization). The signature is compared with the values in each bucket until a match is found after which the complete key is checked. The index is returned as a result of this operation.

From a cache perspective, the problem is that after the hash table has been searched and the

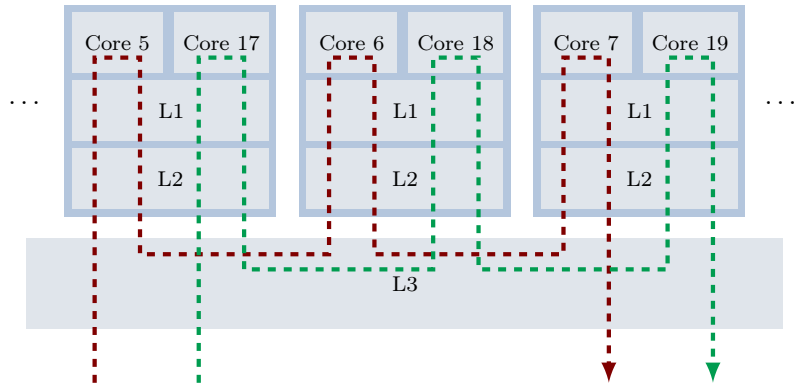


Figure 4.4: A configuration showing a pipeline consisting of two paths. Packets are handled in a similar manner by both paths.

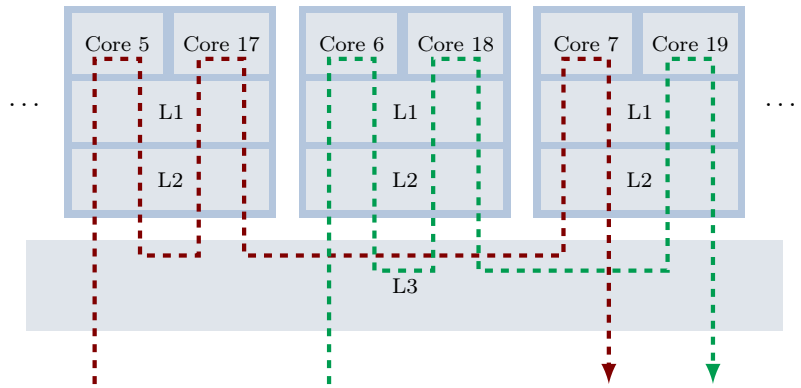


Figure 4.5: An alternative to the pipeline shown in Figure 4.4 that could improve performance.

resulting index has been returned, a second table (containing the data) must be accessed. By patching the Intel® DPDK hash library, we can reduce stress on the cache. The patch allows to store values (that would otherwise be stored in a second table) beside keys in the hash table. If the length of the key is small enough, the value fits in the same cache line. The original Intel® DPDK hash implementation uses padding to keep all data cache aligned. Aligning data means that bytes are wasted in the cache line (these bytes don't contain useful information) which are fetched during a hash lookup. The lookup function in the patched version of the hash library returns the value instead of an index to the value. Another related hash table optimization is to use 4 bucket entries. The details are discussed in RA 3.0.

4.2.3 Bulk handling as a locality improvement

Handling packets in bulk is a common technique used in Intel® DPDK applications to improve locality. Instead of executing all steps required to process a packet, we can execute one step on multiple packets (typically 32 or 64) before moving on to the next. For example, if a core in the pipe line performs a hash lookup followed by routing, we can restructure the code so that all hash table lookups are performed first followed by all routing related actions for the 32 or 64 packets. Depending on the size of the hash table entries, accessing one entry will cause multiple entries to be fetched. One of

the following packets in a small time frame could require one of these entries[4]. Comparing this to an implementation where the core processes packet by packet, actions on the packet after the hash table lookup can evict the hash related data which will need to be fetched again for consecutive packets. In addition, prefetching can be used more efficiently when using bulk handling: as each operation on one packet is followed by operations on other packets, we can make sure, by using software prefetching, that data is available in cache when it's needed. We suggest considering this optimization only if Intel® VTune™ detects hash table lookups as a bottleneck.

4.2.4 Avoid partially used data structures

By avoiding partially used data structures, we can improve locality. When we use data structures, where not all fields are used, the useful data is spread in memory. Accessing data that would otherwise fit in the same cache line could involve access to multiple cache lines. An example use of this technique is to implement specific data structures that contain only the necessary fields for the task at hand in contrast to a general data structure used by all task types.

4.2.5 Use huge page heap memory

All data structures (not only memory used for mbufs) should be allocated in huge pages. The benefit of keeping all data in the huge pages allows minimizing DTLB misses. For example, allocating the memory needed for a lookup table in huge pages reduces DTLB misses. More memory (more entries in the lookup table) can be addressed through one DTLB entry.

4.2.6 Use lookup tables in Load balancer

The load balancer is responsible for distributing packets to multiple worker threads. If the load balancer is not fast enough, the worker threads can become idle. The load balancer has to be as simple as possible. Usually, the choice to which worker thread the packet will be passed is based on a field (or a set of fields) from the packet (see Section 4.2.1 for the motivation). The field is used either in some simple calculation (bitmask, or possibly modulo division) or during a more complex mapping. For more complex mappings, lookup tables should be used. Data structures that are spread out in memory should be avoided (linked lists, trees...).

4.2.7 Minimize maximum number of mbufs

During the initialization step of applications that use Intel® DPDK, the queues of the interfaces and their associated mbuf memory pools are set up. One configuration parameter is the number of mbufs which puts a hard limit on the number of packets in the system at any given time. By reducing this number, performance can be improved as the memory range spanned by the mbufs is reduced.

We can calculate the minimum number of mbufs required in a system. For this, we take the sum of the number of mbufs that can be used by receive descriptors, transmit descriptors, rings and mempool cache (there is a cache per mempool and per thread using the mempool). The buffer used to store packets during the actual packet processing step also needs to be taken into account. The size of this buffer usually depends on the chosen bulk size. We are essentially calculating how many mbufs will be used by the system at the moment that all buffers in all data structures are full. Theoretically, more mbufs are never needed as there are simply no buffers to hold them. If we configure less mbufs, performance might be impacted as the lack of free mbufs causes the packet reception to fail.

[4]Note that the same applies to management structures used for the hash table itself. These management structures will be required for all hash table lookups.

Behavior	Dropping allowed	Dropping not allowed
PCIe bandwidth wasted	More (depends on pipeline)	Less
Cycles wasted	Depends on pipeline	Depends on pipeline
Latency	Lower	Higher
mbuf reordering degree	Higher	Lower

Table 4.1: Overview of (dis)advantages of either allowing dropping packets or not.

Although we suggest starting the optimization by reducing to the minimum number of mbufs, we also note that this may not be the optimal configuration. Using even less mbufs might give better performance. A trade-off between cycles lost to failed packet reception and improved locality has to be made. The trade-off depends on the use case and pipeline. One example situation where we might deliberately configure less than the minimum number of packets is a workload where even with the calculated minimum number, the performance is impacted

4.3 No-Drop rings

Packets are either received from or sent to the Ethernet controller. Depending on the workload, packets are passed between cores using rings. *Traditionally*, the design choice with rings is to drop packets when the rings are full. We propose another design choice where it is guaranteed that once a packet enters the system, it will pass through the complete pipeline without being dropped. The advantages and disadvantages of both methods are discussed below. The overview is given in Table 4.1.

Consider a pipeline of two or more cores. Depending on the workload, the first core might process packets quicker or slower than the second core. If the second core is processing packets at a rate lower than the first, the ring connecting the two cores will become full. This is the situation we will use for our discussion.

Trivially, packets that enter or leave the system through the Ethernet controller consume PCIe bandwidth. If packets are dropped by a core, some of the PCIe bandwidth is wasted. If PCIe bandwidth is limiting throughput (See Section 2.6) wasting it will have a negative impact on performance. PCIe bandwidth waste can be minimized by ensuring that packets are not dropped in the pipeline. The packets will be dropped by the NIC instead. It is important to note that packet reception has an influence on both the RX and TX lanes. If packets are dropped by the NIC, transmission performance could be improved.

Looking at a longer pipeline consisting of multiple steps, we see another problem. Dropping packets just before the packet is passed to the last core in the pipeline means that we are essentially throwing away all the effort that the previous cores in the pipeline have put into the packet.

Average latency *might* increase when packet drops are prevented. With no-drop rings, congestion at a given core in the pipeline causes all preceding cores to slow down. The preceding cores are spending more cycles on enqueuing packets. As a result, the packets spend more time waiting in the queues and new packets arriving at the Ethernet controller are dropped. Essentially, the action of dropping is offloaded to the hardware. On the other hand, by allowing packets to be dropped, the packets in the pipeline can be kept fresh; reducing the latency.

When packets are dropped in the pipeline, the mbufs are returned to the ring in the memory pool. Initially, the addresses of the mbufs are ordered. The benefit of this ordering is that the range in which the addresses fall for the working set is minimal. By allowing packets to be dropped (and mbufs

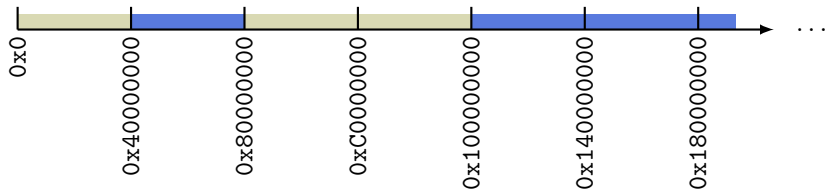


Figure 4.6: Physical memory layout with blue address ranges showing memory regions accessible through huge pages.

to be returned to the memory pool) in the middle of the pipeline, the order cannot be maintained and performance could be impacted. This issue has been discussed in RA 3.0 [Int14b] and in Chapter 3. Note that with some configurations (for example when a software load balancer distributes packets), mbufs will always be reordered.

4.4 Reduce effect of PCIe limitation

PCIe limits the throughput achievable with the Intel® 82599 10 GbE controller when small packets (i.e. 64 byte packets) are used (see Section 2.6). Using 32-bit addresses to store packet data allows transmitting more packets during a time interval compared to using 64-bit addresses. A 3 Double Word (DW) header is used for the TLP when data is stored at 32-bits addresses while 4 DW headers are used when dealing with data stored at 64-bit addresses. Generally, to reduce the effect of PCIe limitation, mbufs should be allocated below the 4 GB boundary.

The huge page allocation is shown by Figure 4.6 in blue. Each huge page is 1 GB in size. The memory layout is taken from a system with 32 GB of memory and kernel boot parameters to allocate 32 huge pages (29 are actually allocated). We can see that there is one huge page below the 4 GB boundary. To reduce the PCIe limitation, we need to allocate the mbufs in the first huge page (`0x40000000-0x80000000`). The only way to achieve this is to create the memory pools before calling other functions that use the Intel® DPDK memory allocation library. If the required number of mbufs occupy more than 1 GB of memory (including padding), this trick cannot be used. There are no continuous memory regions larger than 1 GB below the 4 GB boundary. If multiple memory pools are allocated, a fraction of the memory pools could be allocated above the 4 GB boundary. This could cause a difference in the maximum achievable throughput depending on which interface is considered.

4.5 Prevent false sharing

The smallest unit in the MESI cache coherence protocol is a cache line (typically 64 bytes in size). If two cores are changing parts of the same cache line, coherency messages are exchanged between these cores[5]. This behavior is called false sharing [Int13]. If it is known upfront that some data will only be used by a single core, the data should be aligned to cache line boundaries to prevent or minimize false sharing.

4.6 Prefetching during encapsulation

A packet is stored in an mbuf data structure. A more elaborate explanation of the memory structure is given in the Intel® DPDK programmer's guide but the relevant information is repeated in Figure 4.7.

[5]In Section 5.5, we will explain why coherency messages are needed.

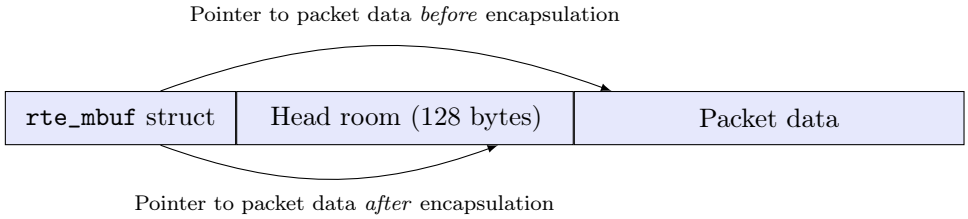


Figure 4.7: `rte_mbuf` memory layout.

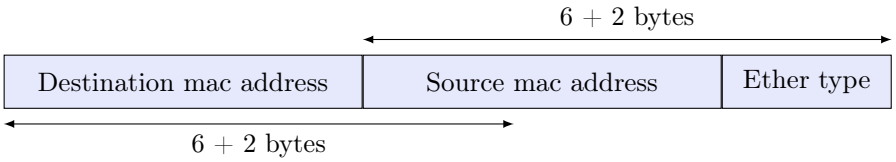


Figure 4.8: Ethernet header.

The first step during encapsulation is to "allocate" memory (call to `rte_pktmbuf_prepend`). The size of the `rte_mbuf` structure is 64 bytes and is typically cache aligned. The reason for the alignment is to prevent packet headers to cross cache line boundaries (assuming they are not resized and they fit in a cache line). Even if we prefetched the packet headers, we should prefetch the prepended buffer separately to load the cache line before the start of the original packet header data. Another motivation for this optimization is that with partial writes, the unchanged portion of the cache line needs to be present in cache.

4.7 Avoid conversion from network to host byte order

In some cases, conversion from network byte order to host byte order is not needed. Consider an implementation that stores a field from a packet and compares the same field from subsequent packets with the stored value. Instead of storing the value in host byte order, we can simply store the value directly. This allows us to eliminate a byte swap that would otherwise be required before the comparison.

4.8 Copy 8 bytes of MAC address

Figure 4.8 shows the Ethernet header. The `ether_addr_copy` function provided by Intel® DPDK writes exactly 6 bytes. By looking at the generated assembly, we can see that this is done by writing 4 bytes followed by 2 bytes. In applications where we write both the destination and the source address, performance can be improved by writing 8 bytes instead of 6 bytes. To prevent unneeded data dependencies when writing the source mac address after the destination mac, we can use the extra 2 bytes that are written with the destination address to write the first part of the source mac address. If more source addresses are used[6], we need to create multiple destination mac addresses for each destination address and source address combination. In the case where all source addresses start with the same bytes (i.e. when using NICs from the same vendor), a single 6 + 2 byte combination can be used. An alternative is to write the source mac address together with the Ethernet type.

[6]The system could use multiple source mac addresses (one for each port on which packets are transmitted).

```
lea     $0x12(%rdi),%rax
movw    $0x1,0x2(%rax)
movw    $0x2,0xe(%rdi)          movw    $0x1,0x14(%rdi)
movw    $0x3,(%rax)             movw    $0x2,0xe(%rdi)
add     $0x16,%rdi              movw    $0x3,0x12(%rdi)
movl    $0x4,0xc(%rdi)          movl    $0x4,0x22(%rdi)
movl    $0x5,0x10(%rdi)         movl    $0x5,0x26(%rdi)
retq                            retq
```

Figure 4.9: Assembly generated with `-O1`.

```
movw    $0x1,0x14(%rdi)
movw    $0x2,0xe(%rdi)          movw    $0x1,0x14(%rdi)
movw    $0x3,0x12(%rdi)         movw    $0x2,0xe(%rdi)
add     $0x16,%rdi              movw    $0x3,0x12(%rdi)
movl    $0x4,0xc(%rdi)          movl    $0x4,0x22(%rdi)
movl    $0x5,0x10(%rdi)         movl    $0x5,0x26(%rdi)
retq                            retq
```

Figure 4.10: Assembly generated with `-O2` and `-O3`.

4.9 Use packet structures instead of moving pointers

Below we give two different approaches to write fields in packets. The first approach uses pointers to the contents of the packet headers. The second approach uses a structure to index into the packets. The first uses `ether_hdr`, `vlan_hdr` and `ipv4_hdr` provided by Intel® DPDK and the second uses `cpe_pkt` to describe the structure of the packet. gcc (Ubuntu/Linaro 4.4.7-2ubuntu2) was used to generate the assembly. Newer versions of gcc behave differently. In the tested cases, those versions generate the same assembly both for packet structures and for moving pointers. Assembly generated for more complex situations has not been compared.

4.10 Remove branches and loop unswitching

Loop unswitching is a method used by compilers to move branches outside of loops. An analogous optimization can be applied by the programmer. Each task in a packet handling application can be seen as a loop. While some decisions have to be made depending on the contents of each packet (i.e. hash lookup), other decisions can be made upfront. The most trivial example of this is where at initialization time a value is set depending on how the system is configured. At runtime, the value is checked to decide how the packet should be handled.

As another example, take a load balancer and a set of worker threads. In this example, we can remove a branch by reusing knowledge from the pipeline. The load balancer decides which worker thread to send the packet to depending on a field in the packet. The choice of which field is used depends on the packet type (i.e. IPv4, IPv6, QinQ, ...). After the worker thread receives the packet, the worker thread handles the packet in different ways depending on the type of packet. The decision made by the load balancer can be reused. A single worker thread task is split in two separate tasks. The first task only handles IPv4 packets and the second task handles only IPv6 packets. The IPv4 worker thread task does not have to check if the packet is an IPv4 packet (this is already ensured

```
test_func:
        pushq   %rbp
        movq    %rsp, %rbp
        subq    $16, %rsp
        movq    %fs:40, %rax
        movq    %rax, -8(%rbp)
        xorl    %eax, %eax
        movl    $0, %eax
        movq    -8(%rbp), %rdx
        xorq    %fs:40, %rdx
        je      .L3
        call    __stack_chk_fail
.L3:
        leave
        ret
```

```
test_func:
        pushq   %rbp
        movq    %rsp, %rbp
        movl    $0, %eax
        popq    %rbp
        ret
```

Figure 4.11: Assembly generated with (left) and without (right) stack protector feature.

by the load balancer). All IPv4 packets are handled together and less auxiliary data structures are used. This can also been seen as a locality improvement. While reusing information found by the LB can be useful, the LB must not perform actual work of the worker threads. Furthermore, the information found by the LB should not be stored in memory as increased memory usage might impact performance.

4.11 Stack protector (distribution dependent)

The stack protector is a security feature that prevents buffer overflow attacks that overwrite the return address on the stack. This is accomplished by pushing a known value (called the canary) onto the stack at the beginning of the function. A buffer overflow attack would have to overwrite the canary value before it can overwrite the actual return address. At the end of the function, the value is compared with what was originally pushed onto the stack. This feature might have negative impact on the performance.

Depending on which Linux distribution is used, the stack protector might be enabled by default. It is recommended to add `-fno-stack-protector` to the compile flags to disable this feature. The flag has no effect if the feature is disabled by default. If the kernel configuration has `CONFIG_CC_STACK_PROTECTOR` enabled, gcc will use the stack protector by default. By adding `-fstack-protector-all` we enable the feature for all functions. We can then study the difference in the generated assembly to see how the feature is implemented. As expected, space on the stack is being allocated for the canary and a branch is used to test if it has the expected value. A function that simply returns 0 was used to generate the assembly shown in Figure 4.11.

4.12 x86-64 ABI on Linux systems

Depending on the number and type of function arguments, the x86-64 Application Binary Interface (ABI) permits function calls to pass arguments only using registers. In some cases, the arguments need to be passed using the stack. Below we give three example situations that should be avoided. The exact details and rules are described in the System V ABI document [MHJM13]. With an “eightbyte”, we are referring to types that fit in a general purpose register.

- %rdi, %rsi, %rdx, %rcx, %r8 and %r9 are used to pass the first 6 (integer) arguments. If more arguments are used, they will always be passed through the stack.
- An aggregate (array or structure) that is larger than 4 eightbytes or that contains unaligned fields will be passed through the stack.
- If an aggregate is passed, each eightbyte is classified separately and depending on the types, the whole aggregate might be passed through the stack.

4.13 Prefetchers

Software prefetchers should be used whenever possible. These hints are under the control of the programmer who is aware of code that could cause cache misses. On the other hand, hardware prefetchers are not under the control of the programmer and they could prefetch data that is left unused. The list bellow shows the hardware prefetchers that can be disabled through the BIOS on Intel® Xeon® Processor E5-2697v2.

- Mid-Level Cache (MLC) Streamer
- MLC Spatial Prefetcher
- Data Cache Unit (DCU) Data Prefetcher
- DCU Instruction Prefetcher

The hardware prefetchers try to guess what data will be required by the application. If memory bandwidth utilization is causing a performance bottleneck, disabling these hardware prefetchers can improve performance only if software prefetching is implemented correctly [Heg14]. Prefetched data, that is not used, increases memory bandwidth utilization without any benefits and can worsen the already existing bottleneck. Furthermore, data already present in the cache would need to be evicted and will need to be fetched again when referenced.

We give an example situation where the DCU prefetches data that is otherwise not needed. When fields in a packet are written to, the DCU prefetcher can be triggered to load the next cache line (which could contain the payload that is of no interest). For more information about hardware prefetchers and their behavior, refer to the Intel® 64 and IA-32 Architectures Optimization Reference Manual.

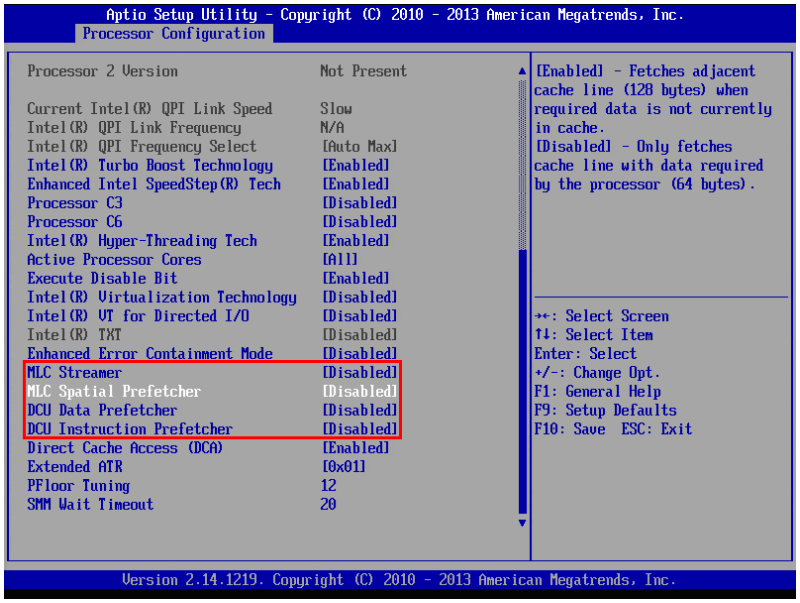


Figure 4.12: Hardware prefetcher settings in the BIOS on Intel® Server Board S2600IP.

Chapter 5

CPU caches and their effects on performance

5.1 Introduction

Writing functionally correct software does not require the programmer to be aware of the underlying caching systems as those systems work transparently. However, if performance is important, caches need to be taken into account while making software design choices. Bottlenecks can be predicted and existing bottlenecks can be explained. We start by introducing basic notions of caching and why it has such a big influence on performance. We examine how memory locations are mapped to caches in Section 5.2. Furthermore, we discuss how the most common cache implementation (set associative caches) works. In Section 5.3, we continue with descriptions of situations where caching affects performance and how the effects can be measured from software that uses these caches. In Section 5.4 we will show one of the commonly used techniques related to caching to improve performance and how this is applied in packet handling applications. In Section 5.5 we partially deviate from actual caching as we discuss the MESI cache coherence protocol. Section 5.6 points out how cache behavior is different in recent processors. We close the chapter by discussing a feature that may be introduced in future processors for monitoring cache and we present a way to use the feature in the context of packet processing applications in Section 5.7.

Caches are used to alleviate bottlenecks caused by the discrepancy between processor speed and memory speed. Multiple levels of cache are used. Each level is smaller in size and more expensive than the previous level, but it is also faster. Actual access time to these cache levels depends both on the processor and on the actual software. The basic notion of caches is that as long as data is in the cache, it can be accessed quickly (cache hit). If the required data is *not* present in cache, it needs to be fetched from slower memory (cache miss). Due to significant differences in access latencies between caches[1], reduction in cache misses can increase performance dramatically. At the highest level, the principle of locality explains how to benefit from caches. If the data accessed is close together in memory, the data will be in cache (spatial locality). Each data access could cause data to be evicted but if accesses to a memory location are clustered in time (temporal locality), the data will probably be in cache.

[1] As an example, the Intel® Nehalem processor has a latency of 4 cycles for access to the first level data cache and 35-40 cycles (approximately 10 times slower) for the third level cache [Int14g].

5.2 Cache organization

In the same way memory is divided into blocks (pages), cache is divided into cache lines. The size of this basic unit is processor dependent but 64 bytes is typical for recent processors. Whenever data is brought into a cache level, the whole cache line is brought in. When a cache line is placed in the cache, an existing cache line needs to be evicted from the cache level. The algorithm that decides which cache line is chosen for eviction is *based on* Least Recently Used (LRU) supporting the principle of locality. Different cache implementations exist. In a system with multiple caches, a combination of cache implementations is used. We will discuss each of these cache types as they are relevant to describe how memory can be mapped to caches.

The most obvious way to implement a cache would be to allow each memory location to be mapped to any location in the cache. A Cache using this implementation is called a fully associative cache. In this ideal situation, the least recently used cache line can be evicted each time a new cache line needs to be placed in the cache. As long as caches are small, this implementation is feasible. A different approach is taken with bigger caches. Another way to implement a cache is to decide up front where a given memory location will be mapped in cache. As an example, consider a cache containing n cache lines. Using $log(n)$ bits from the address, the first n memory locations are mapped to these n cache lines. The next n memory locations are mapped to the same n cache lines and so on. This implementation is called a directly mapped cache. The downside with directly mapped cache is that two memory locations, that map to the same cache location, will always cause evictions and the two locations are never cached at the same time. In the last cache implementation, the cache is divided in sets. This implementation is a trade-off between a fully associative cache (preferable to reduce cache evictions) and a directly mapped cache (preferable to reduce required hardware and implementation costs). A memory location can be mapped to any cache line within a set. Clearly, by limiting a memory location to a set, the cache does not always evict the least recently used cache line[2]. When a cache is said to be N-way set associative, the number of cache lines in a set is N[3].

Next, we describe a common way to map memory addresses to cache lines. A cache with capacity c, a cache line size of l and a set associativity of N is used in the example. Note that a fully associative cache is one where a set spans the whole cache capacity ($N = \frac{c}{l}$). The fact that programs only use virtual memory addresses while (most) caches work with physical addresses is ignored in this example. As long as addresses fall *within* a memory page, the given example still applies. Figure 5.1 shows a memory address. Each part of the memory address has a specific purpose in the context of caches. The lowest order bits of the address define the offset within the cache line. The next lower order bits define which set the memory address maps to. It is within this set that a cache line will need to be evicted. The remaining bits are used to identify which memory location is cached by the cache line.

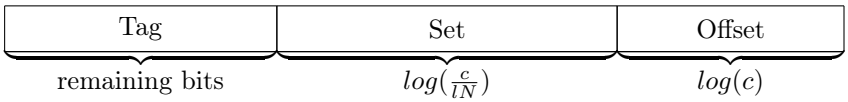


Figure 5.1: Parts of a memory address used during mapping to caches.

[2]A conflict miss is a cache miss caused by a previous cache line eviction where if a different cache line was evicted, the miss could have been avoided. Clearly, restricting the mapping to a set reduces the number of possible cache lines to chose from for eviction. Miss rates are higher in this case compared to a fully associative cache where any cache line could be evicted.

[3]We can find set associativity for a L1 cache through `/sys/devices/system/cpu/cpu0/cache/index0/ways_of_associativity`. Information about L2 and L3 can be found in a similar manner.

5.3 Effects of working set size

The effects of the working set size on caches has been studied extensively in the literature. The actual latency numbers are highly dependent on the processor, the cache capacities and the algorithms used for placement. We will limit our description to a conceptual level with the goal to give background for the discussion in Section 5.6. Clearly, when the working set size is bigger than the cache capacity, cache misses will occur[4]. Intel® processors use caches that are inclusive. With inclusive caches, if data is cached in a specific cache level, it is also present in higher level caches. Using this knowledge, the cache sizes can be measured.

The idea is to create a pointer chasing program where we can control the offsets between the elements in memory. The program walks through a set of elements imitating the behavior of programs with different working set sizes. The memory access pattern depends on how the pointers are initialized. By randomizing the pointers between elements, we can work around any optimizations that detect sequential walks. The program is configured with predetermined number of pointer dereferences. The total execution time is divided by the configured number of dereferences estimating the number of cycles spent on each memory access. By increasing the number of pointer dereferences, the impact of compulsory misses[5] on measurement results can be minimized. Figure 5.2 conceptually describes the results obtainable with the pointer chasing program. System specifics have been annotated for clarity. The described methodology can be used to measure the size of each level of cache and their access latency. Although we expect to see shapes comparable with the one shown in Figure 5.2 on actual systems, the final shape will be influenced by other factors (hardware prefetching, TLBs, hyper-threading, ...). Details on how the shape changes from these factors will not be discussed further as they are less relevant.

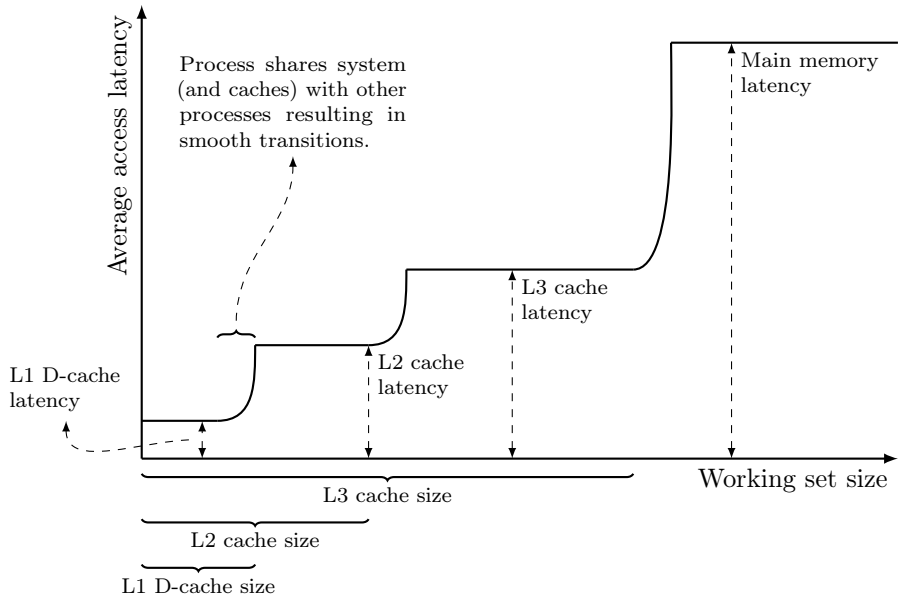


Figure 5.2: Influence of caches on access latency for increasing working set sizes (conceptual).

[4]These type of cache misses are called capacity misses.

[5]A miss caused by the *first* reference to a memory location is called a compulsory miss. Normally, the data is not present in the cache before the first reference.

5.4 Prefetching as a technique for hiding memory access latency

A technique to reduce the number of compulsory misses (and capacity misses) is prefetching. As already noted in Section 5.3, hardware prefetchers detect predefined access patterns. If these hardware prefetchers predict the access pattern correctly, latency will be reduced. Software can benefit from these hardware prefetchers without any changes as this is done transparently by the hardware. Another method is to use software prefetching hints. These prefetchers explicitly tell the hardware that memory will be referenced in the (near) future. The prefetch instruction[6] executes while program execution continues with data that is already present in cache. By the time the program needs the data, the prefetch has completed and the data will be available to the program.

Next, we will show a commonly used method of prefetching in packet processing applications. As we have discussed in Section 2.9, packets are handled in bulks. In this case, prefetching can be used to hide access latency to packet data. This is accomplished by prefetching packet n while processing packet $n - k$ where k is an offset so that the time it takes to process k packets is less than the time it takes to prefetch a packet. Choosing k too big will cause more packets to be in cache and packets could even be evicted before they are processed. The structure of bulk handling with software prefetching is shown by Algorithm 1.

Algorithm 1: General structure of bulk handling packets with software prefetching.

input : A packet bulk of size b with prefetch offset k

for $i \leftarrow 0$ **to** $min(k, b)$ **do**
    `prefetch(`*bulk[i]*`);`
for $i \leftarrow 0$ **to** $max(0, b - k)$ **do**
    `prefetch(`*bulk[i + k]*`);`
    `handle_packet(`*bulk[i]*`);`
for i **to** b **do**
    `handle_packet(`*bulk[i]*`);`

5.5 Cache coherency protocol

We briefly describe the MESI cache coherence protocol with the goal to show how and when it can have an effect on performance. The protocol is used in multi-processor and multi-core systems. As opposed to write-through caches where a write to a cache is immediately reflected in main memory, write-back caches delay the write to the moment when the cache line is evicted. With this, the need for a coherent view, on the data cached by the cores, arises. The protocol, to assure this coherent view, is the MESI protocol. The name is an abbreviation for the four states it uses: Modified, Exclusive, Shared and Invalid. A cache line is always in one of these four states. The situation that could cause the protocol to impact performance is described next. Whenever a cache line is possibly residing in the cache of other cores, the cache line is in the shared state. If a core writes to the cache line which is in this state, a Read For Ownership (RFO) message is broadcast transitioning the state of the cache line in other cores to "invalid". The local state of the cache line moves to the "modified" state. The important thing to note here is that transmitting the RFO message adds overhead to the write operation which could impact performance.

[6] For completeness, we note that different prefetch instructions exist for fetching data up to a specific cache level or fetching data in a non-temporal way.

5.6 Caching in recent processors

As already noted, caching behavior and its effects are processor dependent. The purpose of this section is to show that we cannot easily predict how caches will be utilized. We will describe two aspects of caches in recent (Sandy Bridge and Ivy Bridge) processors that complicate predicting cache utilization.

5.6.1 Intel® Data Direct I/O Technology

Whenever data (control messages and packets) needed to be transferred to the NIC, the data had to be in main memory before the NIC had access to the data. When the processor had a packet that needed to be transmitted, the NIC was be notified. Then, the NIC issued a read which caused the data to be transferred to main memory. In the converse direction, whenever the NIC had data to deliver, it wrote it to main memory. If the location, to which the NIC wrote the data, was cached, that part of the cache was invalidated and a next access from the processor to that location caused a cache miss to occur. As a consequence, the data was moved to the cache from main memory after which the processor was able access the data.

With with Intel® Data Direct I/O Technology (Intel® DDIO) present in Sandy Bridge and Ivy Bridge processors, the detour to main memory can be avoided [Int12a]. It allows the NIC to directly read and write to the cache of the processor bypassing the main memory completely. To use Intel® DDIO, we do not need to change our applications. The feature is operating transparently if the processor supports it. We can verify that Intel® DDIO is working correctly by measuring a reduction in cache misses and memory bandwidth utilization using Intel® Performance Counter Monitor (Intel® PCM) [Int12b]. With basic forwarding applications, we expect to see nearly no memory bandwidth utilization with Intel® DDIO enabled. We use basic forwarding so that it is unlikely that packet data is evicted from cache before we finished processing the packet. Data delivery through Intel® DDIO is limited to 10% of the last-level cache. If there would be no limit, all data could be evicted from cache by arriving packets. However, it is not known which 10% of the cache is used for Intel® DDIO data delivery. We are currently not aware of methods that could be used to determine which part is used for Intel® DDIO.

5.6.2 Cache replacement algorithms

Although the general idea shown in Figure 5.1 is applicable to determine which cache lines could be evicted by the replacement algorithm, caching in recent processors has become more complex. With LRU, we expect to see sharp edges as shown in Figure 5.2. If the same tests are repeated on an Intel® Ivy Bridge processor, different behavior is visible compared to Sandy Bridge processors. There is significant evidence that a combination of cache replacement algorithms are used [Won13]. From this, we conclude that it is even more difficult to predict how cache will be used.

5.7 Cache QoS Monitoring

Cache QoS Monitoring (CQM) is a feature that may be present in future processors [Int14c]. We will first describe what we can monitor through this feature. We will then show how it can be implemented and used in the context of packet processing applications. CQM is part of the more general platform QoS monitoring feature but we only consider L3 cache occupancy monitoring. The goal of using CQM is to gather cache statistics at a higher level of detail then what is possible through Intel® PCM.

CQM is designed to be used by the OS or the Virtual Machine Manager (VMM) to determine cache utilization at the level of an application or a set of applications. Virtual Machines (VMs)[7] are out of the scope of this thesis. The OS can gather statistics at the level of an application by instructing CQM to start monitoring a different application during a context switch. It can do this by changing the active Resource Monitoring ID (RMID) during the context switch. It is up to the OS to manage the associations between RMIDs and applications. At any given moment, the processor can be queried for statistics at the level of RMIDs. In case of multi-threaded applications, the same RMID can be set on multiple cores. The RMID is global within the physical package of the processor[8]. After the OS correctly using the RMIDs, it can query the cache occupancy per application. At the time of this writing (May 2014), there was no support within the Linux kernel for CQM. The remainder of this section will show how the feature can be implemented from user-space and how it can be used in the context of packet processing applications. The advantage of adding support within the application instead of relying on the kernel is that no specific kernel version or patched kernel is required.

In our implementation, each core is dedicated to specific tasks. Therefore, this feature can be used to determine the amount of cache that each core is using. Consequently, the amount of cache required per task can be measured. First, the availability of the feature is tested through the `cpuid` instruction[9]. In our implementation, we are assuming that the OS has no support for this feature. If newer kernel versions are providing functionality based on this feature, it should be explicitly disabled to prevent conflicts with our implementation. After detecting that the feature is provided by the processor, an RMID is associated with each active core. RMID 0 is not used as this is the default RMID used by all cores. When the system is reset, the active RMID is explicitly set to 0 on all cores to return to a known state. After setting up all RMIDs, the occupancy per core can be queried while reporting statistics.

It has been noted that the feature is supposed to be used from the OS. For this reason, it is not surprising that setting the RMID requires the use of Model-Specific Register (MSR)[10] which can only be executed with `ring0` privileges[11]. To set up RMIDs from user space, we can use the `msr` kernel module [Int09b].

We have described one way to use this feature. At the time of this writing, we could not collect results as there were no processors in production that provided this feature. After a processor that provides this functionality becomes available, the implementation that we have described can be used to get a better understanding of how cache is utilized.

[7] In the context of VMs, the feature can be used by the VMM to detect a misbehaving VM (a VM that is using more cache than what was provisioned). The VM can then be migrated to another system.

[8] This is important if processes are moved between processors in a multi-socket system.

[9] This instruction is used to query processor specifics.

[10] MSRs are registers can be used to control processor specific features.

[11] Only code running in kernel mode can run instructions that require `ring0` privileges.

Chapter 6

Conclusion and future work

Before starting with this thesis, we were wondering what the achievable performance would be if software based solutions where used in the context of data plane packet processing. If sufficient performance could be obtained, service providers could potentially consider following the same path when building new equipment. In order to estimate performance, we have described a specific use case (vBRAS), we have implemented additional features like QoS; we have proposed an architecture for a system that combines QoS with vBRAS and we have presented optimization techniques. We have introduced methods to measure some of the performance data (`pkt-stat`). We have measured the performance obtained and explained the different bottlenecks. We encountered various bottlenecks: PCIe limitations, memory considerations, cache aspects... Most of the observations, bottleneck analyses and optimizations are also relevant for other data plane processing applications. Some are even relevant outside the scope of data plane processing. We have shown that, using the right programming APIs and techniques, with a good understanding of the underlying hardware and with the right optimizations in place, the performance is more than acceptable for all packet sizes including the smallest ones. With relatively small packet sizes it is close to the theoretical limit. In a real world implementation, average packet sizes are much bigger and the Ethernet links are not saturated. Our prototype software demonstrates that software based solutions for complex network functions like vBRAS with QoS are viable.

Furthermore, we have introduced methods for measuring internal system behavior (caches) within the context of packet processing applications. To the best of our knowledge, this feature is not yet officially supported by any production processor. We have not gone into detail using CQM to find which task occupies most of the cache and how this relates the actual task. We think that studying cache occupancy will teach more on both how caches are shared between the tasks and how Intel® DDIO influences cache occupancy. Using CQM, we could study how multiple QoS instances interact from a caching perspective. Such knowledge could help in reaching even higher performance. Another subject for future work is related to PCIe. We have shown that we hit the available PCIe bandwidth. Next steps in reverse engineering the protocol would be to measure the number of packets transferred to the NIC within a fixed time interval as opposed to measuring the total throughput. We suspect that more aspects of the underlying protocols can be revealed in this way. We think that with packets below 64 bytes, we could create more accurate measurements of the overhead involved during communication with the NIC. Once we have a better understanding of the protocol and all the overhead involved, we could consider optimizations where access to the NIC's queues from multiple worker threads is scheduled to minimize overuse of the available bandwidth.

In this thesis, we were limited to 10GbE cards. Another topic for future work is implementing and analysing systems that use 40GbE NICs. Following the trend of the increase in the number of cores per processor, we would need to search for ways that allow scaling the system further than what

is currently possible. Even the most basic workloads become four times more difficult due to the available bandwidth. Therefore, we expect, that even in the basic cases of forwarding, new techniques would need to be developed before more realistic workloads can be considered.

Finally, NFV concepts are becoming more popular. Within the same context, the idea of using IT virtualization features to virtualize network functions is becoming more relevant. Future work could study the feasibility and advantages of such an approach; it could analyze the performance aspects in different configurations of virtualization and provide recommendations to service providers.

Appendix A

Packet header structures

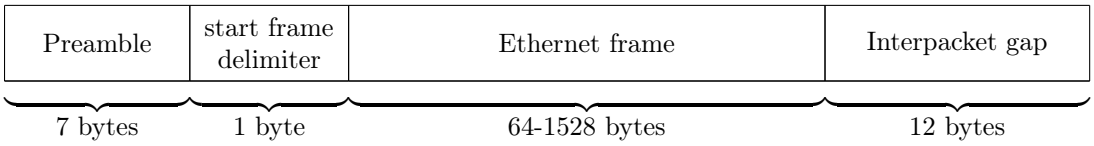


Figure A.1: Preamble, start of frame delimiter and interpacket gap (total of 20 bytes overhead for each packet).

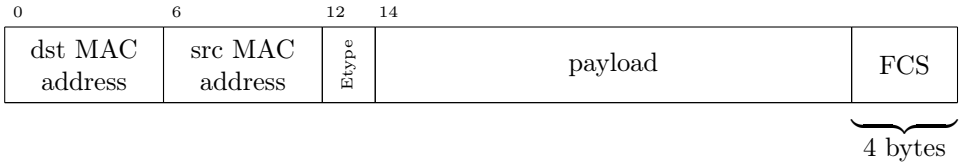


Figure A.2: The IEEE 802.3 Ethernet header. We assume that the frame check sequence (FCS) is part of the Ethernet header at layer 2.

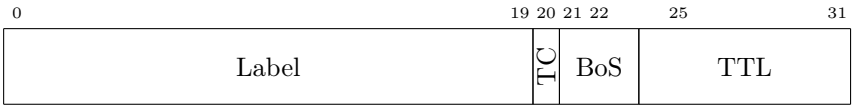


Figure A.3: The MPLS header.

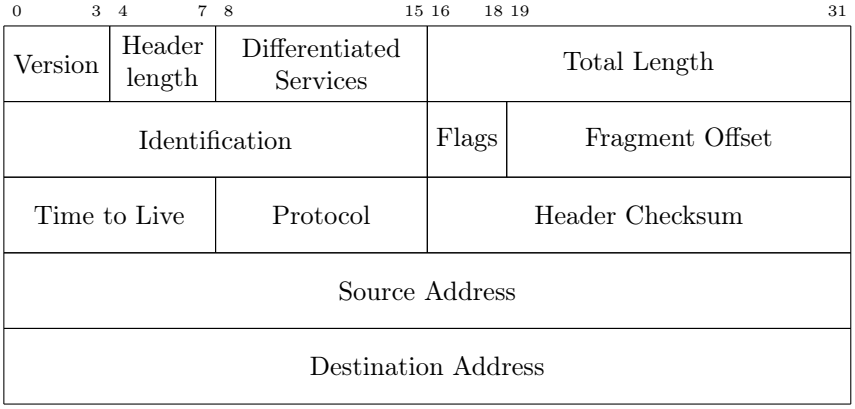


Figure A.4: The IPv4 header. Optional fields are not shown.

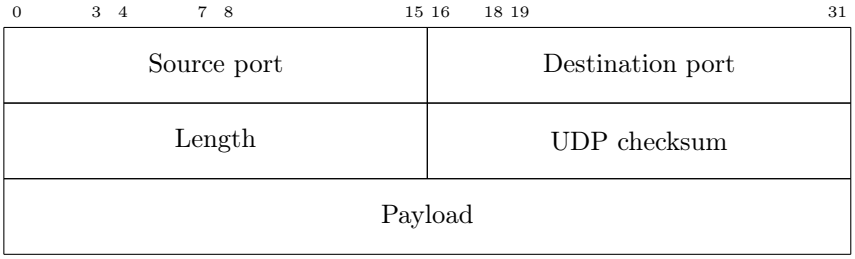


Figure A.5: The UDP header.

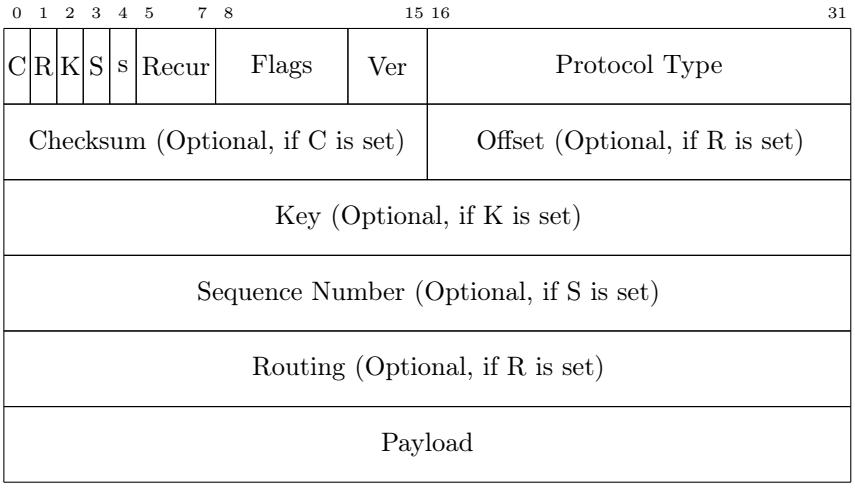


Figure A.6: The GRE header.

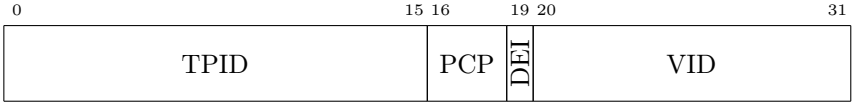


Figure A.7: The VLAN header.

Appendix B

Dutch Summary

Bij de internetaanbieders is er een sterk stijgende vraag naar bandbreedte, dit door klanten die steeds aan hogere snelheden materiaal willen raadplegen. Internetaanbieders gebruiken netwerkappartuur die ontwikkeld is voor slechts één functie. De apparatuur wordt geïmplementeerd met ASICs (hardware ontwikkeld voor slechts één doel). De gespecialiseerde apparatuur is nodig om alle data tijdig te verwerken. Om bijvoorbeeld firewall functionaliteit te voorzien, moet er een nieuwe toestel geïnstalleerd worden dat deze functionaliteit aanbiedt. Tegelijkertijd wordt er momenteel ook aan een versneld tempo geïnnoveerd op gebied van netwerk protocollen. In sommige gevallen zorgt de beperking van de aangeboden functionaliteit van de apparatuur ervoor dat de apparatuur vervangen moet worden. Een oplossing voor de beperking in functionaliteit is het gebruik van programeerbare ASICs. Hierbij kan (een deel van) de functionaliteit geprogrammeerd worden. In de praktijk wordt er een combinatie van ASICs (voor de functionaliteit die vast staat) en programmeerbare ASICs gebruikt. Op het eerste zicht lijkt dit de ideale oplossing te zijn, maar de kosten zijn te hoog.

Daarom wordt er verder gezocht naar goedkopere alternatieven en overwegen internetaanbieders om over te stappen naar software oplossingen die geïmplementeerd worden op standaard servers. Door softwareoplossingen te gebruiken wordt het, dankzij hun flexibiliteit, mogelijk vernieuwingen aan te brengen in de netwerkinfrastructuur, dit met een kostverlagend resultaat. Deze thesis is een haalbaarheidsstudie die gericht is op de volgende vraag: *"Kan netwerkappartuur, die gebruikt wordt door internetaanbieders, vervangen worden door software implementaties op processoren die dienen voor algemene doeleinden, zonder de kwaliteit van de aangeboden diensten negatief te beïnvloeden?"*. Hierbij richten we ons op één netwerkfunctie (Quality of Service). Quality of Service functionaliteit laat de internetaanbieders toe om de kwaliteit van de aangeboden diensten te garanderen en te verhogen. Deze functie wordt beschouwd in combinatie met een broadband remote access server implementatie. De broadband remote access server is apparatuur waarlangs alle klanten van een internetaanbieder met het Internet verbonden zijn. Tijdens de analyse van het systeem, wordt de software en de interactie met de onderliggende hardware bestudeerd. De studies zijn niet eigen aan de specifieke netwerkfunctie en dezelfde werkwijzen, technieken en tests kunnen gebruikt worden voor andere implementaties. De doeltreffendheid van het systeem wordt bestudeerd aan de hand van meetresultaten. Zowel de technieken als hun motivaties worden besproken. Door telkens de motivatie te bespreken wordt aangetoond dat de technieken niet gebonden zijn aan een specifiek systeem. Ze zouden gebruikt kunnen worden in andere implementaties.

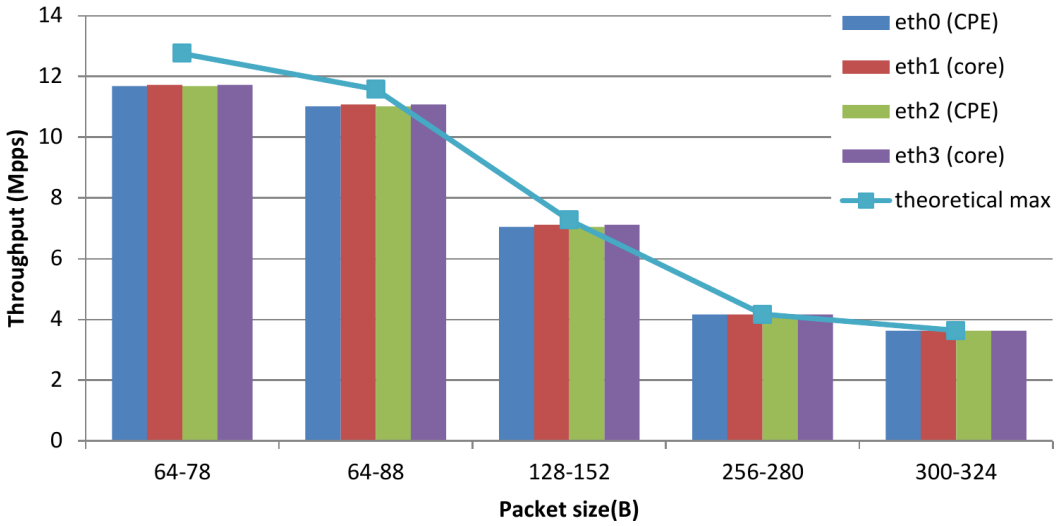
De thesis is onderverdeeld in vijf hoofdstukken. Het eerste hoofdstuk bespreekt de doelstellingen van de thesis en overloopt de gehanteerde structuur. In het tweede hoofdstuk worden de algemene concepten aangehaald. Er wordt in de volgende hoofdstukken teruggegrepen naar deze concepten. De Intel® DPDK softwarebibliotheek staat centraal in het 2^{de} hoofdstuk. De netwerkfunctie implementatie, in deze thesis, zal uitvoerig gebruik maken van deze bibliotheek. In het derde hoofdstuk wordt besproken

hoe we de Quality of Service functionaliteit hebben toegevoegd aan een bestaande broadband remote access server implementatie. We richten ons vooral op aspecten van het onderliggende systeem die de doeltreffendheid van de software implementatie beïnvloeden. In het vierde hoofdstuk halen we optimalisaties aan die gebruikt zijn in de software implementatie. Dankzij deze optimalisaties was het mogelijk om de doeltreffendheid van het systeem te verhogen en acceptabele resultaten te behalen. In het laatste hoofdstuk bespreken we caching. Dit aspect van het onderliggend systeem is van groot belang bij het bepalen van de uiteindelijke resultaten. Er wordt eerst achtergrondinformatie gegeven. Het doel is om aan te tonen dat het niet eenvoudig is om te voorspellen hoe caching door de software gebruikt zal worden. Tenslotte wordt processor functionaliteit, die eventueel beschikbaar zal zijn in toekomstige processoren, besproken. Deze functionaliteit laat toe om te analyseren hoe caching door de software gebruikt wordt. Er wordt besproken hoe deze functionaliteit zou kunnen gebruikt worden in de context van netwerkfunctie implementaties.

Het witboek over netwerkfunctie virtualisatie werd gepubliceerd in oktober 2012 en beschrijft de voordelen van implementaties van netwerkfunctioniteit in software. Hierbij wordt er verwezen naar Intel® DPDK, een softwarebibliotheek die deze implementaties mogelijk maakt. Intel® DPDK biedt functionaliteit aan voor software die gebruikt zal worden in pakketgeschakelde netwerken. Intel® DPDK bestaat uit een verzameling van softwarebibliotheken. Elke softwarebibliotheek biedt een deel van de functionaliteit (bijvoorbeeld rechtstreekse toegang tot de netwerkkaarten) aan die gebruikt zou kunnen worden in de volledige implementatie van een netwerkfunctie. Software die netwerkgerelateerde taken uitvoert, maakt typisch gebruik van functionaliteit aangeboden door het besturingssysteem. Intel® DPDK laat toe om software te schrijven die tien tot elf keer performanter is dan de standaard Linux Kernel op vlak van netwerkfunctionaliteit. Door kennis over de onderliggende hardware toe te passen zijn de aangeboden software bibliotheken geoptimaliseerd. De programmeur heeft controle over welke deel van zijn programma op welke CPU core wordt uitgevoerd, dit in tegenstelling tot andere software waarbij er typisch op het niveau van threads gewerkt wordt.

De doeltreffendheid van het systeem wordt beschreven aan de hand van metrieken. De metrieken kunnen gebruikt worden om verschillende systemen te vergelijken en het effect van aanpassingen aan implementaties te analyseren. De eerste en belangrijkste metriek bepaalt de doorvoer van het systeem. In de meeste gevallen (en in alle gevallen in deze thesis) wordt er enkel gekeken naar het begin van de pakketten (header). Daarom wordt doorvoer uitgedrukt in termen van pakketten per seconde. Aangezien we werken met 10GbE (deze snelheid wordt gebruikt in het netwerk van de internetaanbieder), kunnen we bepalen hoeveel pakketten er *maximaal* per verbinding verstuurd kunnen worden. Hierbij kijken we naar de kleinst mogelijke pakketten (64 bytes). Voor 10GbE is dit 14.88 miljoen pakketten per seconde (Mpps). Merk op dat, in praktijk, deze limiet zelden wordt behaald. Om deze snelheden te halen moeten de verbindingen *volledig* gesatureerd worden met de kleinst mogelijke pakketten. Buiten een Denial-of-service aanval is dit onwaarschijnlijk. Als het systeem pakketten niet aan deze snelheid kan afhandelen, kan het systeem alsnog gekarakteriseerd worden als een systeem dat pakketten van, bijvoorbeeld, 128 bytes aan maximale snelheid kan afhandelen. Om de grootte van pakketten in perspectief te brengen, werd er door Cooperative Association for Internet Data Analysis (CAIDA), voor de gemiddelde pakketgrootte, 770 bytes vermeld voor het jaar 2013. Merk op dat de gemiddelde grootte enkel *een indicatie* is van de distributie van de pakketgrootte. Deze grootte wordt hier vermeld enkel om aan te geven dat 14.88 miljoen pakketten onrealistisch is. Bovendien wordt er door dezelfde bron vermeld dat de verbindingen nooit volledig gesatureerd waren. De tweede metriek is vertraging. Vertraging is de tijd die verstrijkt tussen het versturen en het ontvangen van een pakket. Er is inherent vertraging aanwezig binnen de beschouwde netwerkapparatuur. Deze vertragingen worden berekend zodat ze in acht kunnen genomen worden bij de studie van de meetresultaten.

Om de implementaties te testen en te evalueren gebruiken we gespecialiseerde en erkende testapparatuur. De apparatuur laat toe om de meeste metingen uit te voeren. Met de meetappaertuur konden een aantal meetingen niet worden uitgevoerd omwille van beperkingen in de functionaliteit. Om deze



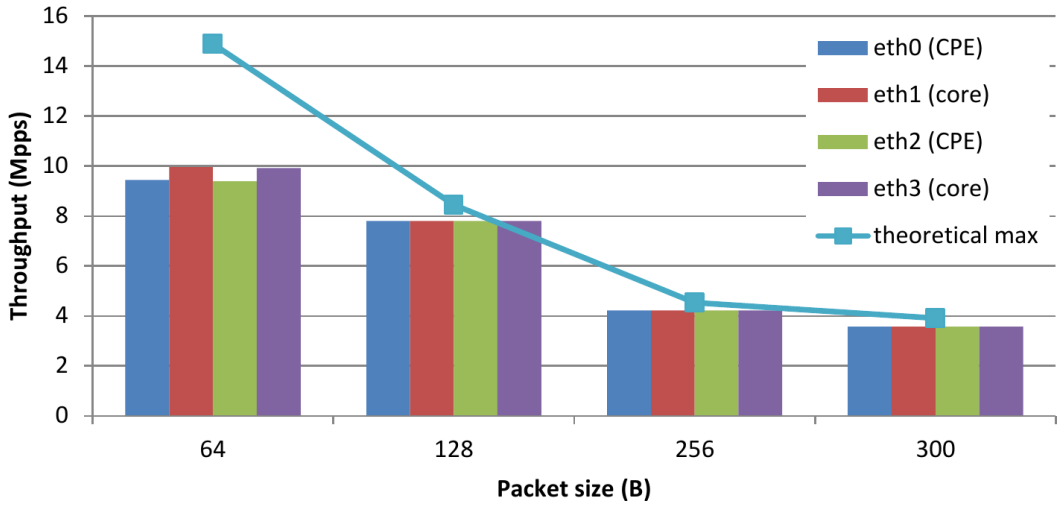
Figuur B.1: Doorvoer voor het systeem dat Broadband Remote Access Server functionaliteit voorziet. Merk op dat het theoretisch maximum bepaald wordt door het grootst gebruikte pakket tijdens een test. In de test met pakketten van 64 bytes en pakketten van 78 bytes bepalen de pakketten van 78 bytes het theoretisch maximum.

beperkingen te omzeilen moesten we software (`pkt-stat`) ontwikkelen waarmee de metingen op het correcte niveau van detail uitgevoerd kunnen worden.

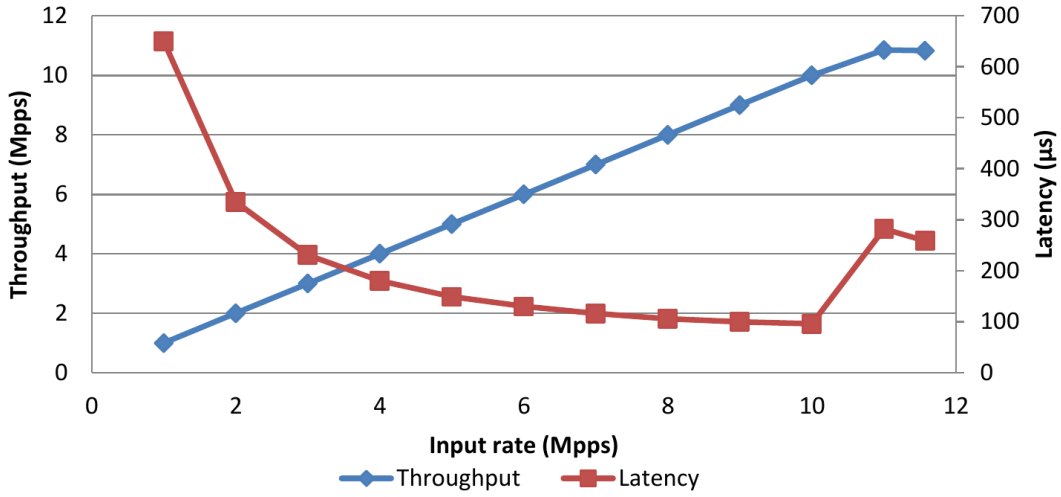
Na het introduceren van basisconcepten en metrieken wordt er overgegaan tot de implementatie en studie van Quality of Service. Quality of Service wordt als losstaand systeem geanalyseerd. Eerst worden de voordelen van Quality of Service besproken. Hierna wordt er overgegaan naar alle parameters die betrokken zijn bij de configuratie. Er wordt besproken wat de verwachtingen zijn van het systeem voor de gegeven parameters. Er wordt aangegeven hoe het systeem de stroom van pakketten zal beïnvloeden. Het systeem, dat als eerste wordt bestudeerd, heeft slechts één taak wat de analyse deels vergemakkelijkt. De analyse dient om de uiteindelijke implementatie te begrijpen en te evalueren. Tijdens de studie worden bottlenecks besproken en wordt de interactie van de software met de onderliggende hardware bestudeerd.

Na deze beschrijvingen wordt er overgegaan naar het bestuderen van het 2^{de} systeem waarbij er, naast Quality of Service, ook Broadband Remote Access Server functionaliteit wordt voorzien. De architectuur van het systeem wordt besproken en gemotiveerd. Hierbij wordt er zowel teruggegrepen naar het systeem dat enkel Quality of Service voorziet als naar de bestaande implementatie. De resultaten zijn te zien in Figuren B.1, B.2, B.3 en B.4. De Figuren B.1 en B.2 tonen de gemeten doorvoer per verbinding. Het theoretisch maximum wordt getoond. Hier is te zien dat, na het toevoegen van Quality of Service, het theoretisch maximum gehaald wordt voor pakketgroottes van minstens 128 byte. De Figuren B.3 en B.4 tonen de vertraging geïntroduceerd door het systeem. Doorvoer wordt ook aan de hand van de figuren getoond daar vertraging hiermee in verband staat.

De onderzoeksvraag was of het mogelijk zou zijn om, met een implementatie in software, een systeem te creëren dat voldoende prestaties behaald. Als de prestaties aanvaardbaar zouden zijn, dan zouden internetaanbieders het eventueel overwegen om gelijkaardige systemen te gebruiken binnen hun infrastructuur omwille van de verhoogde flexibililteit en mogelijkheid tot het verlagen van de kosten. We hebben ons enkel gericht op één specifiek geval waarbij we verschillende problemen zijn tegengekomen. We hebben de resultaten geanalyseerd en technieken voorgesteld om de prestaties te verhogen. De oplossingen zouden eventueel zelfs voor andere doeleinden kunnen gebruikt worden

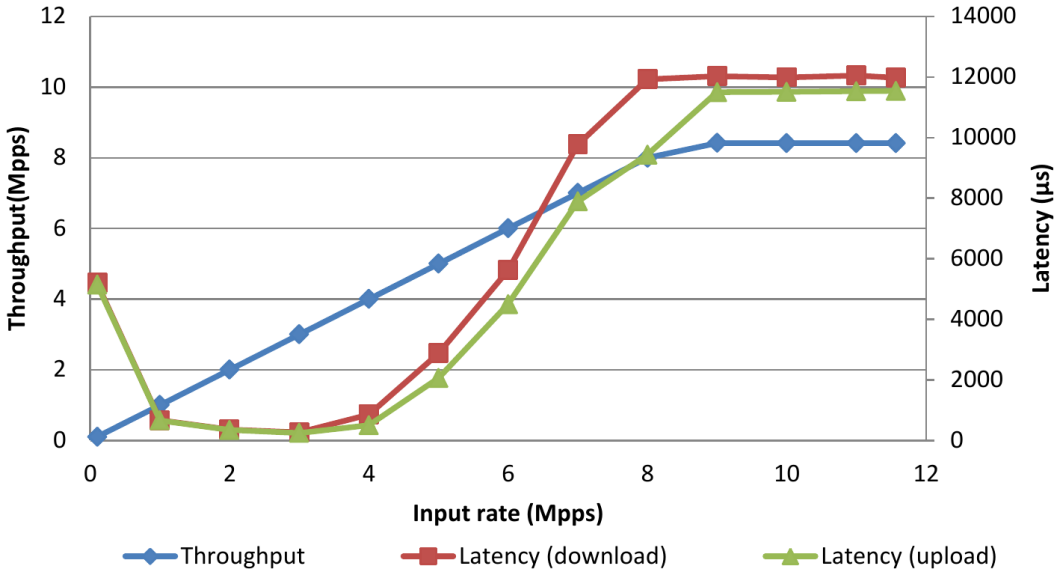


Figuur B.2: Doorvoer voor het systeem dat, naast Broadband Remote Access Server functionaliteit, Quality of Service voorziet.



Figuur B.3: Gemiddelde vertraging (rood) en doorvoer (blauw) voor het systeem dat Broadband Remote Access Server functionaliteit voorziet.

(buiten het domein van netwerkfuncties). We hebben aangetoond dat, met de juiste technieken en met begrip van de onderliggende hardware, meer dan acceptabele prestaties, voor de kleinste pakketten, gehaald kunnen worden. Voor pakketten die minstens 128 bytes groot zijn halen we zelfs het theoretisch maximum. Hiernaast hebben we een methode beschreven die gebruikt zou kunnen worden om caching binnen de processor verder te bestuderen in functie van het uitgevoerde werk. We hebben geen resultaten van deze methode omdat de functionaliteit die nodig is niet beschikbaar was in de gebruikte processoren. De functionaliteit zou eventueel voorzien kunnen worden in toekomstige processoren. We zouden dan de beschreven methode kunnen gebruiken om de bestaande implementatie verder te bestuderen. De analyse zou gebruikt kunnen worden voor verder doorgevoerde optimalisaties. Een ander onderwerp waar we in de toekomst aandacht aan zouden willen besteden is het verder onderzoeken van de bestaande bottlenecks. Als we deze bottlenecks in meer detail begrijpen, dan



Figuur B.4: Gemiddelde vertraging (rood en groen) en doorvoer (blauw) voor het systeem dat, naast Broadband Remote Access Server functionaliteit, Quality of Service voorziet.

zouden implementaties verder geoptimaliseerd kunnen worden. Aangezien we ons beperkt hebben tot 10GbE zouden we, in de toekomst, systemen die gebruik maken van 40GbE willen analyseren. Tenslotte, in de context van netwerkfunctie virtualisatie, is er interesse voor virtualisatie op dezelfde wijze waarop het gebruikt wordt in de IT industrie. Het zou daarom relevant zijn om virtualisatie, en de verschillende configuraties die hierbij mogelijk zijn, te analyseren tijdens ons toekomstige werk.

Bibliography

[AKZ99] Guy Almes, Sunil Kalidindi, and Matthew J. Zekauskas. A One-way Delay Metric for IPPM. RFC 2968, RFC Editor, September 1999.

[Bar07] Michael Baron. *Probability and Statistics for Computer Scientists*. Taylor & Francis, 2007.

[BM99] Scott Bradner and Jim McQuaid. Benchmarking Methodology for Network Interconnect Devices. RFC 2544, RFC Editor, March 1999.

[Bud07] Ravi Budruk. Pci express® basics. `http://www.pcisig.com/developers/main/training_materials/get_document?doc_id=4e00a39acaa5c5a8ee44ebb07baba982e5972c67`, 2007.

[Cis05] Cisco. Stacked vlan processing. `http://www.cisco.com/c/en/us/td/docs/ios/12_0s/feature/guide/qinq.pdf`, 2005.

[Cis12] Cisco. Understanding switch latency. `http://www.cisco.com/c/en/us/products/collateral/switches/nexus-3000-series-switches/white_paper_c11-661939.pdf`, June 2012.

[Cor11] Jonathan Corbet. Transparent huge pages in 2.6.38. `http://lwn.net/Articles/423584/`, January 2011.

[DC02] Carlo Demichelis and Philip Chimento. IP Packet Delay Variation Metric for IP Performance Metrics (IPPM). RFC 3393, RFC Editor, November 2002.

[Dre07] Ulrich Drepper. What every programmer should know about memory, 2007.

[ETS12] ETSI. Network functions virtualisation - introductory white paper. `http://portal.etsi.org/NFV/NFV_White_Paper.pdf`, October 2012.

[Gor10] Mel Gorman. Huge pages parts 1 to 5. `http://lwn.net/Articles/374424/`, February 2010.

[Hat] Red Hat. Huge pages and transparent huge pages. `https://access.redhat.com/site/documentation/en-US/Red_Hat_Enterprise_Linux/6/html/Performance_Tuning_Guide/s-memory-transhuge.html`.

[Heg14] Ravi Hegde. Optimizing application performance on intel® core™ microarchitecture using hardware-implemented prefetchers. `https://software.intel.com/en-us/articles/optimizing-application-performance-on-intel-coret-microarchitecture-using-hardware-implemented-prefetchers`, 2014.

[HG99a] Juha Heinanen and Roch Guerin. A Single Rate Three Color Marker. RFC 2967, RFC Editor, September 1999.

[HG99b] Juha Heinanen and Roch Guerin. A Two Rate Three Color Marker. RFC 2968, RFC Editor, September 1999.

[Hic] Paul Hick. Statistical information for the CAIDA Anonymized Internet Traces (collection). http://imdc.datcat.org/collection/1-06XR-Z=Statistical+information+for+the+CAIDA+Anonymized+Internet+Traces.

[Int] Intel Corporation. How to call vtune resume and pause api from fortran and c/c++ code? https://software.intel.com/en-us/articles/how-to-call-resume-and-pause-api-from-fortran-code.

[Int22] Intel Corporation. Telefónica i+d: Unleashing network-based service innovation. http://www.intel.com/content/www/us/en/high-performance-computing/high-performance-computing-xeon-e5-telefonica-study.html, 20122.

[Int08] Intel Corporation. Design patterns for packet processingapplications on multi-core intel® architecture processors. http://www.intel.com/content/www/us/en/intelligent-systems/intel-technology/ia-multicore-packet-processing-paper.html, December 2008.

[Int09a] Intel Corporation. *Intel® 64 and IA-32 Architectures Software Developer's Manual*. Number 253669-033US. December 2009.

[Int09b] Intel Corporation. msr - x86 cpu msr access device. http://man7.org/linux/man-pages/man4/msr.4.html, 2009.

[Int10] Intel Corporation. What are forwarding modes and how do they work? http://www.intel.com/support/express/switches/sb/cs-014410.htm, 2010.

[Int12a] Intel Corporation. Intel® data direct i/o technology (intel® ddio): A primer. http://www.intel.com/content/dam/www/public/us/en/documents/technology-briefs/data-direct-i-o-technology-brief.pdf, February 2012.

[Int12b] Intel Corporation. Intel® performance counter monitor - a better way to measure cpu utilization. https://software.intel.com/en-us/articles/intel-performance-counter-monitor-a-better-way-to-measure-cpu-utilization, August 2012.

[Int13] Intel Corporation. Avoiding and identifying false sharing among threads. http://software.intel.com/en-us/articles/avoiding-and-identifying-false-sharing-among-threads, 2013.

[Int14a] Intel Corporation. Intel Network Builders Reference Architecture: Network Function Virtualization: Quality of Service in Broadband Remote Access Servers with Linux and Intel® Architecture. http://networkbuilders.intel.com/docs/Network_Builders_RA_NFV_QoS_June2014.pdf, June 2014.

[Int14b] Intel Corporation. Intel Network Builders Reference Architecture: Network Function Virtualization: Virtualized BRAS with Linux and Intel® Architecture. http://networkbuilders.intel.com/docs/Network_Builders_RA_vBRAS_Final.pdf, Februari 2014.

[Int14c] Intel Corporation. Intel® 64 and ia-32 architectures software developer's manual volume 3b: System programming guide, part 2. `http://www.intel.com/content/www/us/en/architecture-and-technology/64-ia-32-architectures-software-developer-vol-3b-part-2-manual.html`, February 2014.

[Int14d] Intel Corporation. Intel® data plane development kit. `http://www.intel.com/content/www/us/en/intelligent-systems/intel-technology/packet-processing-is-enhanced-with-software-from-intel-dpdk.html`, January 2014.

[Int14e] Intel Corporation. Intel® data plane performance demonstrators. `https://01.org/intel-data-plane-performance-demonstrators`, April 2014.

[Int14f] Intel Corporation. Intel® vtune™ amplifier xe 2013. `http://software.intel.com/en-us/intel-vtune-amplifier-xe`, 2014.

[Int14g] Intel Corporation. *Intel® 64 and IA-32 Architectures Optimization Reference Manual*. Number 248966-029. March 2014.

[Int14h] Intel Corporation. *Intel® 82599 10 Gigabit Ethernet Controller: Datasheet*. Number Revision 2.9. January 2014.

[Int14i] Intel Corporation. *Intel® Data Plane Development Kit: Programmer's Guide*. Number 326003-006. January 2014.

[ker] Transparent huge pages in 2.6.38. `https://www.kernel.org/doc/Documentation/vm/hugetlbpage.txt`.

[lib13] Tcpdump/libpcap public repository. `http://www.tcpdump.org/`, 2013.

[MHJM13] Michael Matz, Jan Hubička, Andreas Jaeger, and Mark Mitchell. *System V Application Binary Interface AMD64 Architecture Processor Supplement*. Number Draft Version 0.99.6. October 2013.

[Spi12] Spirent. How to test 10 gigabit ethernet performance. `http://ekb.spirent.com/resources/sites/SPIRENT/content/live/FAQS/10000/FAQ10597/en_US/How_to_Test_10G_Ethernet_WhitePaper_RevB.PDF`, March 2012.

[WI] Wind River and Intel Corporation. High-performance multi-core networking software design options. `http://embedded.communities.intel.com/servlet/JiveServlet/previewBody/7070-102-1-2281/7785_PerformanceDesignOptions_WP_1111.pdf`.

[Win13] Wind River. Hands-on lab: Intel® data plane development kit and wind river intelligent network platform with deep packet inspection. `https://intel.activeevents.com/sf13/connect/fileDownload/session/A2DEBBD0E050F8F6FB2399BBE772C5BA/SF13_COML001_100.pdf`, 2013.

[Won13] Henry Wong. Intel ivy bridge cache replacement policy. `http://blog.stuffedcow.net/2013/01/ivb-cache-replacement/`, January 2013.

[WWS13] Carl Wallén, Krister Wikstrom, and Petri Savolainen. Open event machine 1.2. `http://sourceforge.net/projects/eventmachine/`, 2013.

Auteursrechtelijke overeenkomst

Ik/wij verlenen het wereldwijde auteursrecht voor de ingediende eindverhandeling:
Network infrastructure optimization

Richting: **master in de informatica-multimedia**
Jaar: **2014**

in alle mogelijke mediaformaten, - bestaande en in de toekomst te ontwikkelen - , aan de Universiteit Hasselt.

Niet tegenstaand deze toekenning van het auteursrecht aan de Universiteit Hasselt behoud ik als auteur het recht om de eindverhandeling, - in zijn geheel of gedeeltelijk -, vrij te reproduceren, (her)publiceren of distribueren zonder de toelating te moeten verkrijgen van de Universiteit Hasselt.

Ik bevestig dat de eindverhandeling mijn origineel werk is, en dat ik het recht heb om de rechten te verlenen die in deze overeenkomst worden beschreven. Ik verklaar tevens dat de eindverhandeling, naar mijn weten, het auteursrecht van anderen niet overtreedt.

Ik verklaar tevens dat ik voor het materiaal in de eindverhandeling dat beschermd wordt door het auteursrecht, de nodige toelatingen heb verkregen zodat ik deze ook aan de Universiteit Hasselt kan overdragen en dat dit duidelijk in de tekst en inhoud van de eindverhandeling werd genotificeerd.

Universiteit Hasselt zal mij als auteur(s) van de eindverhandeling identificeren en zal geen wijzigingen aanbrengen aan de eindverhandeling, uitgezonderd deze toegelaten door deze overeenkomst.

Voor akkoord,

Nemeth, Balazs

Datum: **18/06/2014**