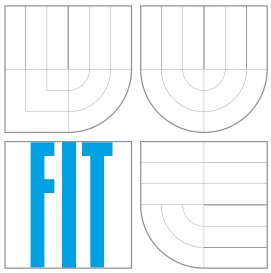
VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY



FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV INFORMAČNÍCH SYSTÉMŮ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF INFORMATION SYSTEMS

40GBE SMĚROVAČ PRO OPERAČNÍ SYSTÉM GNU/LINUX

TOWARDS 40GBE GNU/LINUX ROUTER

DIPLOMOVÁ PRÁCE
MASTER'S THESIS

AUTOR PRÁCE
AUTHOR

Bc. JOSEF LUŠTICKÝ

VEDOUCÍ PRÁCE
SUPERVISOR

Ing. MATĚJ GRÉGR

BRNO 2015

Brno University of Technology - Faculty of Information Technology

Department of Information Systems Academic year 2014/2015

Master Thesis Specification

For: **Lučtický Josef, Bc.**
Branch of study: Computer Networks and Communication
Title: **Towards 40GbE GNU/Linux Router**
Category: Networking

Instructions for project work:

1. Study 40GbE protocol and its support in GNU/Linux operating system.
2. Describe routing process in GNU/Linux kernel.
3. Setup a laboratory for testing 40Gbps routing performance. Describe methods and software used for traffic generation.
4. Test routing performance of GNU/Linux kernel with full BGP table and 40GbE network interface card.
5. Evaluate results, discuss necessary system settings for achieving maximum routing performance.

Basic references:

- Gördén, Bengt, Olof Hagsand, and Robert Olsson. Towards 10Gbps open-source routing. Linux Kongress, Hamburg, October, 2008
- Olsson, Robert. Control and forwarding-plane separation of an open-source router. 2013. online, http://www.histfilfak.uu.se/digitalAssets /21/21239_opensourcerouting.pdf
- Hagsand, Olof, Robert Olsson, and Bengt Görden. Open-source routing at 10Gb/s. *Swedish National Computer Networking Workshop, SNCNW 2009, Uppsala, Sweden*. 2009.
- IEEE P802.3ba 40Gb/s and 100Gb/s Ethernet Task Force. *official web site*. IEEE. June 19, 2010.

Requirements for the semestral defense:
Items 1 to 3.

Detailed formal specifications can be found at http://www.fit.vutbr.cz/info/szz/

The Master Thesis must define its purpose, describe a current state of the art, introduce the theoretical and technical background relevant to the problems solved, and specify what parts have been used from earlier projects or have been taken over from other sources.

Each student will hand-in printed as well as electronic versions of the technical report, an electronic version of the complete program documentation, program source files, and a functional hardware prototype sample if desired. The information in electronic form will be stored on a standard non-rewritable medium (CD-R, DVD-R, etc.) in formats common at the FIT. In order to allow regular handling, the medium will be securely attached to the printed report.

Supervisor: **Grégr Matěj, Ing.**, DIFS FIT BUT
Beginning of work: November 1, 2014
Date of delivery: May 27, 2015

L.S.

Dušan Kolář
Associate Professor and Head of Department

Abstrakt

Účelem této práce je popis protokolu 40Gb Ethernet, popis směrovacího procesu v jádře Linux a navrhnout a provést testování výkonnosti směrování se síťovým adaptérem pro 40Gb Ethernet. Výsledky a nastavení pro získání maximální výkonnosti směrování jsou dále popsány v této práci.

Abstract

The purpose of this thesis is to describe 40Gb Ethernet, describe routing process in the Linux kernel and to design and perform benchmark of routing performance with a 40Gb Ethernet network interface card. The results and system settings for achieving maximum routing performance are further described in the thesis.

Klíčová slova

GNU, Linux, ethernet, směrovač, software, IP, síť, měření, propustnost, operační systém

Keywords

GNU, Linux, ethernet, router, software, IP, network, measurement, throughput, operating system

Citace

Josef Luštický: Towards 40GbE GNU/Linux Router, diplomová práce, Brno, FIT VUT v Brně, 2015

Towards 40GbE GNU/Linux Router

Prohlášení

Prohlašuji, že jsem tuto diplomovou práci vypracoval samostatně pod vedením pana Ing. Matěje Grégra.

.....
Josef Luštický
May 24, 2015

Poděkování

Děkuji vedoucímu práce Ing. Matějovi Grégrovi z FIT VUT za poskytnutí praktických rad, vybaveného pracovního místa v laboratoři a pomoc při sestavování hardwaru. Děkuji Ing. Viktorovi Pušovi, Ing. Štěpánovi Friedlovi a Ing. Martinovi Špinlerovi ze sdružení CESNET a projektu Liberouter.org za poskytnutí měřícího vybavení. Děkuji Ing. Pavlovi Kislingerovi z VUT za poskytnutí výkonného serveru.

Rozšířený abstrakt

Diplomová práce popisuje standard IEEE 802.3ba z roku 2010, který definuje protokol 40 a 100 Gigabit Ethernet, konkrétně se v kapitole 2 věnuje protokolu 40 Gb Ethernet, jehož aplikace se dnes postupně dostává do praxe. V současnosti je provoz tohoto protokolu možný pouze na optických spojích, nicméně plánovaný standard IEEE 802.3bq počítá i s provozem 40 Gb Ethernetu po metalickém vedení. 40 Gb Ethernet zůstává do značné míry zpětně kompatibilní se staršími standardy Ethernetu, zejména formát rámce zůstal zcela nezměněn. Mezi počítači využívajícími protokol 40 Gb Ethernetu tak může být v jednom směru posíláno až přibližně 59 milionů rámců za sekundu o minimální velikosti 72 bajtů, nebo až přibližně 4,6 GB přenášených dat za sekundu při posílání rámců o maximální velikosti 1526 bajtů (bez rozšíření typu 802.1Q apod.).

Současný vývoj výkonu procesorů nestačí držet krok s narůstající rychlostí komunikačních protokolů na linkové vrstvě ISO/OSI modelu. Zatímco rychlost procesorů se zdvojnásobí přibližně jednou za 2 roky, rychlost linkových protokolů se zdvojnásobí za 18 měsíců. Při přenosu 59 milionů rámců za sekundu je interval mezi dvěma po sobě příchozími rámci přibližně 16 nanosekund. Za tuto dobu musí být systém schopen daný rámec korektně zpracovat, jinak hrozí jeho zahlcení dalším síťovým provozem.

Operační systém GNU/Linux se snaží držet krok se zrychlováním síťové komunikace pomocí množství optimalizací. Kapitola 3 popisuje zpracování síťového provozu tímto operačním systémem se zaměřením na směrování IP paketů. Síťový stack zodpovědný za zpracování paketů a směrování je implementován v jádře opračního systému Linux. Síťový stack jádra Linux používá pro reprezentaci síťových paketů strukturu *sk_buff* ve všech vrstvách síťového stacku. Při přechodu mezi jednotlivými vrstvami je předáván pouze ukazatel na tuto strukturu s patřičným pozměněním hlaviček struktury dle dané vrstvy.

Při zpracování paketu vrstvou zodpovědnou za síťový protokol IP dochází ke zpracování na základě rozhodnutí směrovacího subsystému. Směrovací subsystém využívá interní směrovací databázi (Forwarding Information Base) k rozhodnutí o následující funkci, která bude daný paket zpracovávat. Forwarding Information Base je v Linuxu implentovaná pomocí struktury Trie. Linux využívá algoritmus Longest Prefix Match k prohledání této struktury. Výsledkem rozhodnotí může být zahození paketu, předání funkci k lokálnímu doručení nebo předání funkci *ip_forward*, čímž dochází ke směrování daného paketu. Obdobně funguje i směrování paketů protokolu IPv6.

Kromě tohoto zpracování, které je specifické pro směrování IP paketů, je zpracování síťového provozu spojeno s další režií jako je oznámení o příchozím paketu pomocí přeřušení, počítání kontrolních součtů, přiřazení do front atd. O snížení této režie se snaží jak vývojáři jádra pomocí mechanismů jako je NAPI nebo Generic Receive Offload, tak výrobci síťových karet pomocí hardwarového počítání kontrolních součtů nebo podporou vícefrontového zpracování. Právě podpora více front v síťových kartách umožnuje společně s vlastnostmi sběrnice PCI-Express distribuování zpracování paketů na více procesorů a tím škálování síťové propustnosti. Škálování je hlavním tématem současného vývoje a má největší vliv na celkovou propustnost systému.

Pro účely měření byla využita síťová karta Mellanox ConnectX-3 EN se 2 fyzickými porty a hardwarový generátor síťového provozu Spirent. Kapitola 4 popisuje jakým způsobem lze testování s poskytnutým hardwarem provést a jaká je možnost konfigurace parametrů jádra Linuxu s ohledem na popsané zpracování paketů v předchozí kapitole.

Kapitola 5 popisuje postup zapojení a zprovoznění testovací sítě pro účely měření. Instalovaný operační systém je CentOS 7 s jádrem verze 3.10.0-123.20.1 a také upstream jádrem verze 4.0.2. Dále je zde popsána konfigurace operačního systému, instalace nového

firmware síťové karty a konfigurace hardwarového generátoru paketů Spirent.

V kapitole 6 jsou prezentovány výsledky měření a vliv jednotlivých konfiguračních možností na výkon směrování paketů v jádře Linux. Tyto výsledky jsou dále stručně komentovány v kapitole 7, kde jsou také shrnuty nabyté poznatky a identifikovány hlavní problémy zamezující lepší propustnosti.

Součástí diplomové práce jsou i přílohy s návodem importování internetových směrovacích záznamů z protokolu BGP, aktualizaci firmware síťové karty Mellanox ConnectX-3 EN a stručný souhrn kroků pro dosažení maximálního výkonu směrování v operačním systému GNU/Linux.

Contents

Chapter 1

Introduction

The growth of Ethernet from 10 Mbit/s to 10 Gbit/s has already surpassed the growth of microprocessor performance. The 40 Gigabit Ethernet makes the performance gap even larger, but it is still the original Ethernet underneath - an old technology with a lot of compatibility issues for high-speed packet processing. The recent 40 and 100 Gigabit Ethernet standard opens doors to high-speed networking, but it requires other parts of the network to scale within.

The GNU/Linux operating system is used in a wide range of computers interconnected with high-speed Ethernet. An important task of the Linux network stack is to forward traffic. This is relevant especially when discussing core routers, which operate in the Internet backbone. Forwarding occurs on Layer 3 of the ISO/OSI network model. The performance of a software-based solution that uses GNU/Linux, cannot compete with commercial products that can count on the help of specialised hardware. However, various stack bypass solutions have shown, that the Linux kernel is not using the CPU optimally.

The purpose of the thesis is to provide a comprehensive performance analysis of the Linux kernel in IP packet forwarding. The 40 Gigabit Ethernet protocol is able to transmit up to 59 million frames per second and 4.6 GB of L2 data per second. Such speed can easily burden the CPU with a large amount of TCP/IP protocol processing required.

Apart from the 40 Gigabit Ethernet protocol itself, the packet processing in the Linux kernel is described in the thesis. Since the emerge of 100 Mbps Ethernet, the Linux kernel engineers have been optimising the network stack towards high-speed packet processing. Hardware vendors have made various optimisations, which help the operating system to lower the amount of processing required.

In the thesis, a high-end server with a 40 GbE network interface card and 2 Intel Xeon CPUs was setup to measure the routing performance of the Linux kernel. The measurements presented in this thesis demonstrate performance influences of various system settings such as scaling mechanisms, Reverse path filtering or SELinux. The measurements presented in this thesis also include comparison of IPv6 processing performance against IPv4 processing performance. Additionally, the thesis presents the Linux routing performance with imported routes from the Internet BGP protocol. At the time of writing, there are approx. 538 000 routes announced in the public BGP, which leads to expensive software lookups in the Forwarding Information Base of the Linux kernel.

Spirent hardware packet generator was used to generate the traffic and to collect the results. Unfortunately, the measurements must have been configured manually, since the provided Spirent does not contain licenses to automate the testing scenarios.

Measuring performance of the software IP routing using GNU/Linux-based operating

system with 40 Gigabit Ethernet can reveal bottlenecks that need to be eliminated on the way to a full-speed 40 Gigabit TCP/IP processing. If the system processes packets on Layer 3 fast enough, next step is to optimise higher layers of the network stack.

Chapter 2

40 Gigabit Ethernet

The 40 and 100 Gigabit Ethernet were ratified in 2010 as the IEEE 802.3ba standard [50]. These speeds open doors to significantly higher-capacity networks and enable networks to scale in ways that were previously impossible. The 40 and 100 GbE operate in full duplex mode only. This is also the first time two speeds have been included in an Ethernet standard.

A rough estimate of the CPU processing required to handle a given Ethernet link speed is that one hertz of CPU processing is required to send or receive one bit. This general rule of thumb was first stated by PC Magazine in the mid 1990's, and it is still used as a rule of thumb today [1].

The core networking doubles its speed approximately every 18 months, whereas CPU I/O performance doubles every 24 months. The growth of Ethernet speed has already surpassed the growth of microprocessor performance [1]. Figure 2.1 shows, that the 802.3ba Ethernet standard further extends the performance gap.

The 802.3ba standard specifies the physical coding sublayer that is common to both 40 and 100 Gb/s physical layer implementations [50]. This physical coding sublayer is known as 40GBASE-R and 100GBASE-R. 802.3ba further specifies the following 40 GbE port types. All of them use 4 fiber pairs and 40GBASE-R physical coding.

- 40GBASE-SR4 runs over multimode fiber with Short reach for at least 100 m range
- 40GBASE-LR4 runs over single mode fiber with Long reach, with the required operating range at least 10 km
- 40GBASE-CR4 is a Copper type over twinaxial cable
- 40GBASE-KR4 is a port type for bacKplanes
- 40GBASE-ER4 with an Extended operating range at least 30 km, which is currently under discussion by the IEEE P802.3bm 40GBASE-ER4 Task Force [52].

The 40GBASE-SR4 runs on Quad (4-channel) Small Form Factor Pluggable (abbreviated as QSPF or QSPF+). QSPF is a high-density fiber connector with 12 strands of fiber. Each channel has a dedicated transmit fiber and a dedicated receive fiber. The middle four fibers remain unused [8]. The channel layout is shown in figure 2.2.

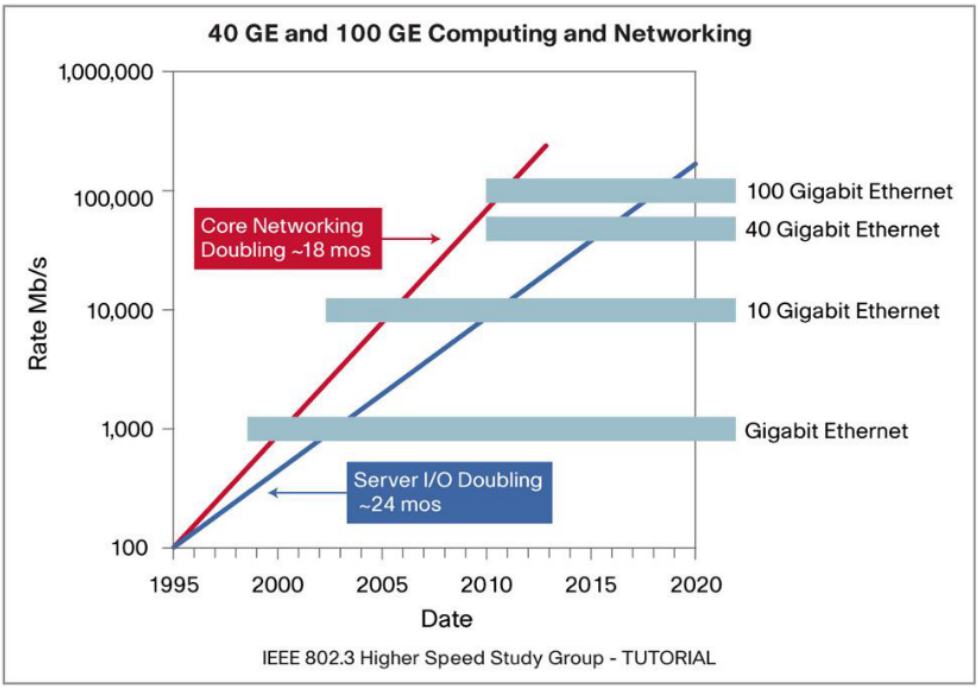


Figure 2.1: Network and CPU performance gap (source: [8])

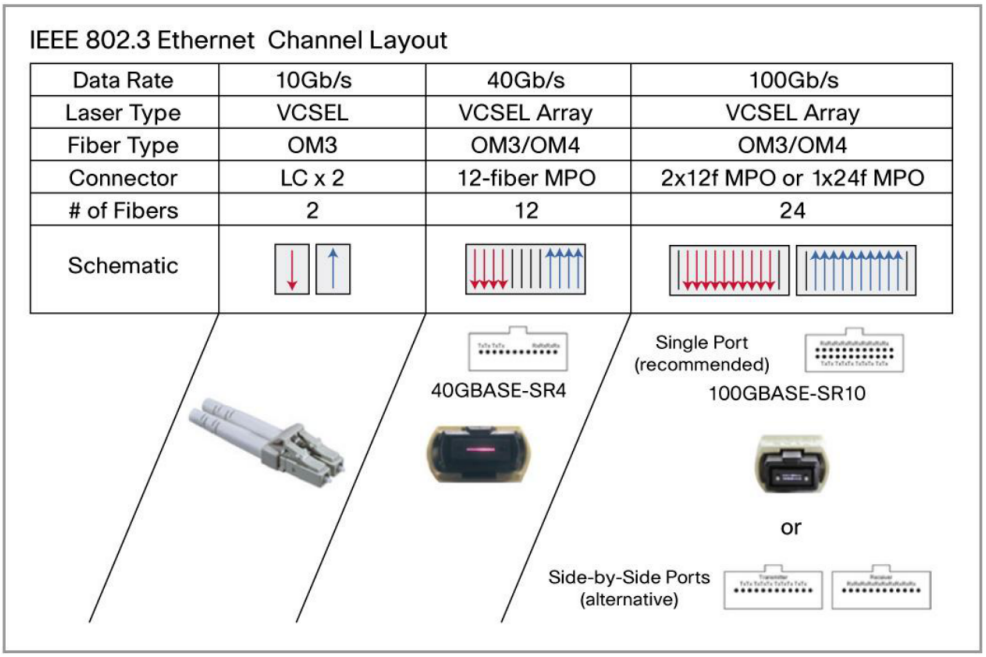


Figure 2.2: IEEE 802.3 Ethernet Channel Layout (source: [8])

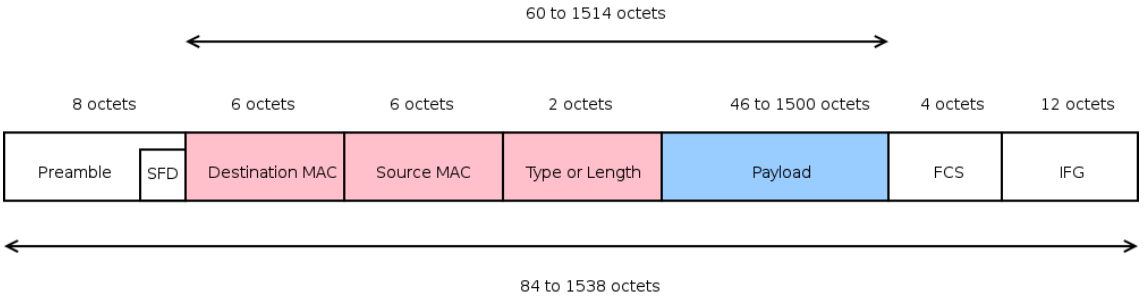


Figure 2.3: Ethernet frame format

2.1 Frame rates

At 40 Gigabits per second rate, it takes $\frac{8}{40\times10^9} = 0.2\ ns$ to transfer a single octet. The time to transfer a whole frame depends on its size. The standard Ethernet frame size remains unchanged in the IEEE 802.3ba standard and is between 72 and 1526 octets. Frames are separated by the Interframe gap (IFG) - a pause between two consecutive frames. In 40 and 100 Gigabit Ethernet, the length of IFG may vary due to clock tolerance and lane alignment requirements, but it must be at least 1 octet. The mean length of IFG is 12 octets [50].

Figure 2.3 shows the Ethernet frame format with no extensions (e.g. 802.1Q VLAN tag). Each frame starts with an 8-octet Preamble, which consists of a 7-octet pattern of alternating 1 and 0 bits, and 1 octet of 1010 1011 value called Start of Frame Delimiter (SFD). The SFD is designed to break the previous 7-octet pattern and to signal the start of the actual frame [50].

The Destination MAC address, Source MAC address and Type or Length fields form the link-layer Ethernet header. The Type or Length field represents Type in case of Ethernet II (DIX) frame and Length in case of IEEE 802.3 frame. Both formats are in use today [5].

The Payload field represents the transferred Layer 3 data of a variable size. The size of the data is bound by the standard Ethernet frame size and can be up to 1500 octets, which is the Maximum Transmission Unit (MTU) of Ethernet [50]. Frame Check Sequence (FCS) is a 4-octet cyclic redundancy code to check the integrity of the received frame.

The Preamble, FCS and IFG carry no data and provide a necessary overhead of 24 octets for each frame. The size of this overhead is independent from the Payload size. Upon arrival of a frame, the rest is transferred from a network adapter to the host CPU and processed. The data in the Ethernet header (Source and Destination MAC addresses and Type or Length) are used for L2 processing. The data in the Payload field are used for L3 processing (i.e. routing).

The maximum sized frame is 1538 octets including the Interframe gap (8-octet Preamble + 14-octet Ethernet header + 1500-octet Payload + 4-octet FCS + 12-octet Interframe gap). At 40 Gbps rate, a traffic consisting of only the maximum sized frames of 1538 octets produces $\frac{40\times10^9}{1538\times8} = 3\ 250\ 975$ frames per second. In reference to the minimum frame size of 84 octets including IFG, the 40 Gigabit Ethernet can transmit up to $\frac{40\times10^9}{84\times8} = 59\ 523\ 809$ frames per second.

Since each frame must be processed separately, such high frame rates require enormous fast processing speed. Despite the continual frame rates increases, the 1500 byte Maximum Transmission Unit (MTU) of Ethernet remains unchanged.

Extensions to allow larger frames were made by several vendors. The typical maximum sized jumbo frames in use are 9 038 octets (carrying 9 000 octets of Payload) [2]. The 40 GbE transmits $\frac{40\times10^9}{9038\times8} = 555\ 555$ of such jumbo frames per second at full rate.

2.2 Throughput

With 40 GbE, not only the per frame processing became a concern. In case of the maximum sized frames, 40 Gbps Ethernet NIC transfers $3\ 259\ 452 \times 1514 \doteq 4.6$ GB/s of L2 data (Ethernet header and Payload) to the host CPU. Since the frame overhead is independent from its size, the efficiency of Ethernet drops significantly with smaller frames. In case of the minimum sized frames, the transfer rate drops to $59\ 523\ 809 \times 60 \doteq 2.6$ GB/s while still operating at 40 Gbps rate. Such data rates require appropriate bus between the NIC and the CPU to operate at full speed.

A single PCI-Express 2.0 lane provides throughput of 5 Gigatransfers per second in each direction [39]. In this case, Gigatransfer per second is the same as gigabit per second, but it also includes the bits that are lost as a result of the interface overhead. PCI-Express 2.0 uses 8b/10b coding, that is, 8 bits of data cost 10 bits to transfer (the same as in SATA case) [20]. Therefore, the actual bandwidth is 500 MB/s per lane. A PCI-Express 2.0 link with 8 lanes provides 4 GB/s throughput, which is not enough to transfer the 40 GbE traffic between the NIC and the CPU. The widest 16-lanes link doubles the throughput to 8 GB/s, which is sufficient for a 40 GbE adapter.

PCI-Express 3.0 increases throughput to 8 Gigatransfers/s in each direction for a single lane [39]. Additionally, it uses more efficient 128b/130b coding. This way, a bandwidth of 984.6 MB/s per lane is achieved, almost twice the PCI-Express 2.0. An 8-lane PCI-Express 3.0 link provides 7 876.8 MB/s bandwidth which is sufficient to handle 40 GbE traffic.

The above calculations do not include an additional overhead of the PCI-Express link headers and signalling interrupts. The PCI-Express devices are required to support the Message Signalled Interrupts (MSI) feature [39]. MSI is a technique to generate interrupts by writing to a specified address, which has been written into the peripheral's configuration during initialisation. The interrupt signalling consists of sending a Write request over the PCI-Express link to the specified address [35]. There can be up to 32 MSI interrupts assigned to a single device, but the number of interrupts the device uses must be a power of 2 [38].

MSI-X is an extension to the MSI mechanism, which introduces various new features. MSI-X provides support for signalling an interrupt directly to a particular CPU and it allows to use up to 2048 interrupts and the number of interrupts is not restricted to a power of 2. This feature is used by modern 40 GbE adapters to spread the work related to traffic processing among multiple CPUs. The device can use either MSI or MSI-X, but not both simultaneously [38].

Although both PCI-Express 2.0 x16 and PCI-Express 3.0 x8 links can be used for 40 GbE cards, NIC vendors tend to produce 40 GbE PCI 3.0 x8 adapters, because a PCI-Express x8 adapter can be plugged into a x16 slot, but not the other way round [39]. An adapter with less lanes saves both costs and troubles finding an empty x16 slot.

2.3 Compatibility

The previous 10 Gigabit Ethernet standard was ratified by the IEEE 802.3 Working Group in 2002 [48]. As opposed to 40 GbE, the cabling plant of 10 GbE uses just a single pair of fiber. In 2013, Cisco introduced a replacement that does not require a change to the cabling plant and makes it possible to run 40 GbE over a single pair of multimode fiber [4].

Since its ratification in 2002, it took four years to standardise 10 GbE over copper twisted pair 10GBASE-T in 2006 [49]. 10GBASE-T can run over a Category 6 cable within the range of 55m. For full 100m range, a Category 6a cable is required [49]. One of the previous barriers to 10GBASE-T adoption was the power consumption per port compared to other 10 GbE variants. However, improvements in semiconductor manufacturing technology significantly decreased power use to the point where it is no longer a concern. Nowadays, 10GBASE-T and Category 6A cabling costs less than optical fiber technology [32].

The 40 GbE over Category 8 copper twisted pair is currently being discussed by the IEEE P802.3bq 40GBASE-T Task Force. The maximum range for 40GBASE-T is planned to be just 30 meters [51]. Such range should be sufficient for most switch-to-server connections in data centres. Because of the shorter distance, the power consumption of 40GBASE-T should be less than of 10GBASE-T [32].

40 Gigabit Ethernet is backward compatible with its predecessor. Due to concerns around vendor and equipment interoperability, IEEE has determined they will not support or define Jumbo frames [2]. Modern 100Mbps or higher Ethernet also uses constant signalling, which avoids the need for the preamble [59]. However, the frame format is preserved for today's Ethernet transmission speeds to avoid making any changes. With the upcoming 40GBASE-T variant, 10M/100M/1G/10G/40G link speed auto-negotiation can be expected in the future.

The 40 Gigabit Ethernet protocol support was introduced in the Linux kernel by various NIC vendors. Red Hat Enterprise Linux 7 supports 40 Gigabit network interface controllers since its initial release [42]. Additional support was also introduced in RHEL 6.6 [41].

Chapter 3

Networking in the Linux kernel

The Linux kernel version 3.10 consists of nearly 17 million lines of code and the networking is arguably half of the kernel. The GNU/Linux operating system is used in a wide range of routers. Ranging from home and small office routers, to enterprise routers and core high-speed routers on the Internet backbone [47].

In reference to the ISO/OSI layered model of network, the kernel does not handle any layer above L4. Those layers (the session, presentation and application layers) are handled solely by user-space applications. The physical layer (L1) is also not handled by the Linux kernel, but by the network interface card [45]. Each layer is handled by its corresponding subsystem in the kernel. Figure 3.1 shows the distribution of responsibilities for each layer of the ISO/OSI model.

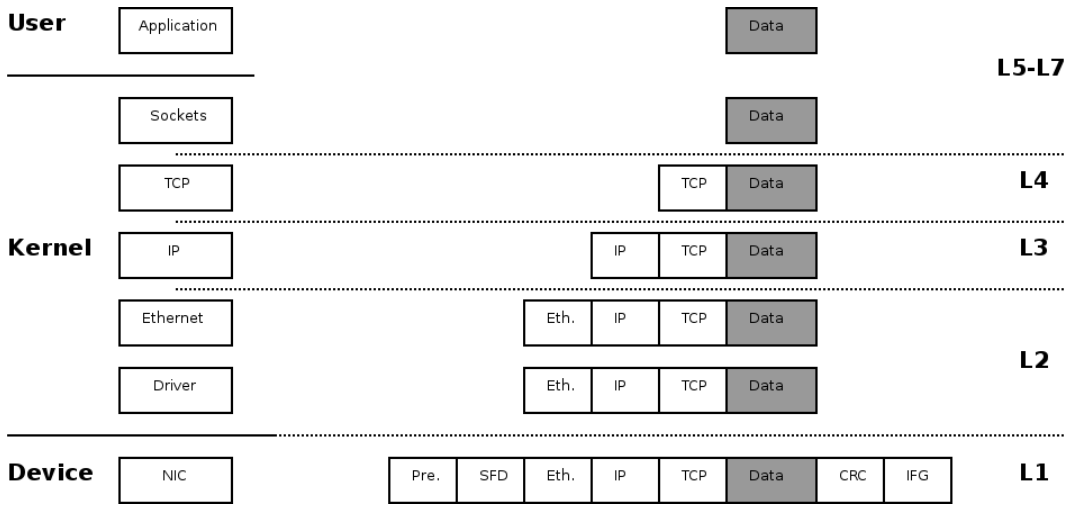


Figure 3.1: Linux kernel layers

Every received incoming frame is passed to the kernel. For a DMA-capable network card, the kernel allocates a buffer from its memory (receive ring buffer) and passes its descriptor to the device driver during the device initialisation. These descriptors are used to store the frames received on the network card by using DMA transfers [46]. A buffer descriptor indicates where the buffer resides in the kernel memory and how big the buffer is. The purpose of the DMA transfer is to move data without CPU intervention. Once a

complete frame is transferred using a device DMA transfer, the device raises an interrupt to inform the kernel about the received data [46].

Each received packet is then handled by a matching L3 protocol handler. An IPv4 packet is handled by the *ip_rcv()* function and an IPv6 packet by *ipv6_rcv()* [45]. Similarly, each outgoing packet is passed downwards through the layers of the network stack. The device driver is associated with a specific link type (e.g. Ethernet), so it knows how to interpret the L2 header and extract the information about which L3 protocol is encapsulated [5].

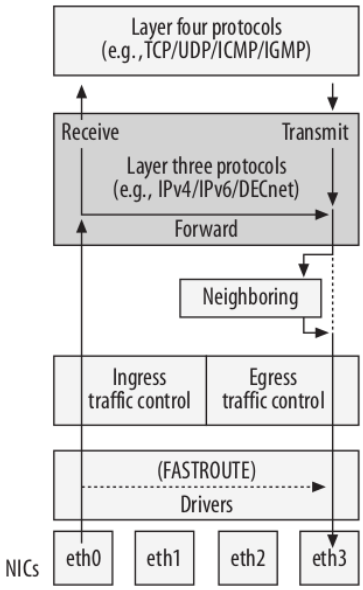


Figure 3.2: Linux kernel traffic flow (source: [45])

Figure 3.2 shows an overview of the traffic flow in the Linux kernel described above. Parts that are discussed in this thesis include the Forward path for IPv4 packets (routing), Ingress traffic control, Egress traffic control and the Fastroute feature, which is mentioned briefly as it is no longer supported [45]. The hardware of NICs, Neighboring and L4 protocols are not discussed in this thesis. As the packet passes through different layers of the stack, the kernel manipulates it using an internal data structure, called *socket buffer*. This structure contains the actual packet, but it is not copied between layers. Instead, a pointer to the structure is passed when passing the packet through the stack [5].

3.1 Socket buffer

The socket buffer (*struct sk_buff*, often abbreviated as *skb*) is an internal kernel data structure that represents an incoming or outgoing packet. The socket buffer *struct sk_buff* is defined in *include/linux/skbuff.h* of the kernel source code [22]. A single packet is always stored in its own *skb* as the packet crosses through the kernel layers. Some members of the *skb* are set sooner or later depending on the direction of the packet [45]. Listing 3.1 shows a part of the *struct sk_buff* definition.

```
struct sk_buff {
  . . .
  struct sock *sk;
  struct net_device *dev;
  . . .
  __be16 protocol;
  unsigned long _skb_refdst;
  . . .
  sk_buff_data_t tail;
  sk_buff_data_t end;
  unsigned char *head,
                *data;

  sk_buff_data_t transport_header;
  sk_buff_data_t network_header;
  sk_buff_data_t mac_header;
  . . .
};
```

Listing 3.1: Notable members of struct sk_buff

Every transmitted *skb* has an associated socket object **sk*, which points to the socket that the packet comes from. It is a network layer representation of sockets. If the packet is forwarded, then *sk* is set to NULL, because it was not generated on the local host. For incoming packets, the **dev* member of skb structure points to the incoming network device, and for outgoing packets to the outgoing network device [45].

The *protocol* member of *skb* represents the Type or Length field found in the Ethernet header. If the value is less than 0x0600 then the frame is interpreted as an Ethernet II frame and the field represents Type (e.g. 0x0800 in case of IP or 0x86DD in case of IPv6). Otherwise it is interpreted as an 802.3 frame and the field represents Length. The *__be16* data type denotes a 2-byte Big Endian value [22].

The *_skb_refdst* member is not assigned immediately upon frame reception, but it is assigned by a higher-layer protocol handler (e.g. in the IPv4 stack). The *_skb_refdst* member is used to store a reference to the result of a routing decision, called destination entry object. This object is created by the routing subsystem and it points to the next packet processing function [45].

The *head* and *end* point to the beginning and end of the space allocated to the buffer, whereas the *data* and *tail* are set while moving through the stack according to the layer which currently processes the packet [5]. The *sk_buff_data_t* data type is a typedef to either *char** or *unsigned int*. In the first case it is a direct pointer to the data, in the later case it represents offset [22].

The *mac_header* member is a link layer encapsulation and points to the start of the L2 header in the frame. Similarly there are *network_header* and *transport_header* members of *struct sk_buff*.

Figure 3.3 shows the discussed members of the *struct sk_buff* and their relationship to the Ethernet frame carrying a TCP/IP packet.

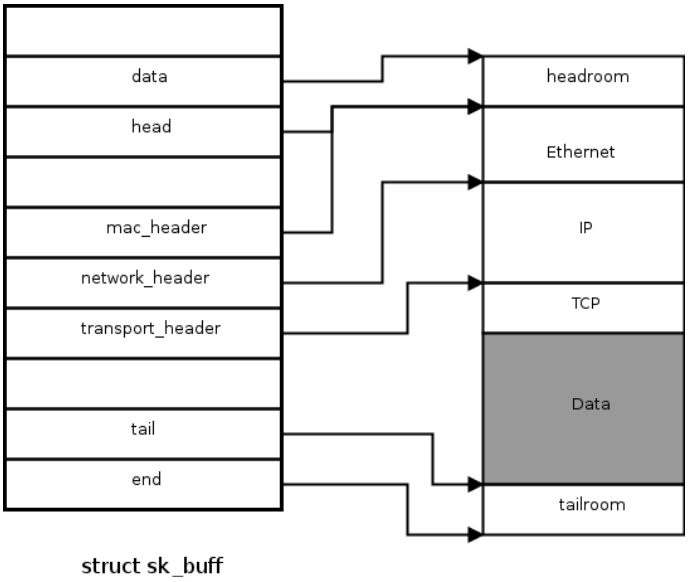


Figure 3.3: Socket buffer structure

According to the *protocol* member, a pointer to the *skb* of the received packet is passed to a higher-layer protocol handler. In case of an IPv4 packet, it is the *ip_rcv()* function, which is part of the Linux IPv4 stack.

3.2 IP stack

The Linux IPv4 stack, or simply the IP stack, claims to be the most RFC-compliant network stack available [27]. The functions found in the IPv4 stack are responsible for handling IPv4 packets only. IPv6 packets are handled by the IPv6 stack, which is different, however, most of the principles of IPv4 processing, apply to IPv6 processing as well [45].

Figure 3.4 shows the core functions of the IP stack for ingress packet processing in the Linux kernel. For the sake of simplicity, the figure shows no IPsec, Fragmentation or IP-Options processing. The NF_IP_* items represent places where the Netfilter hooks can be applied. Netfilter is an internal kernel framework for packet filtering, network address translation, port translation, and more [60]. Netfilter can be manipulated from user-space using the *iptables* utility.

The *ip_rcv()* function performs mostly sanity checks - IP version, IP checksum and header length are checked [22]. If the received packet passes all the checks, it proceeds to the NF_INET_PRE_ROUTING hook callback, if such callback is registered. If it was not discarded by the netfilter hook, the *skb* associated with the packet is passed to the *ip_rcv_finish()* function, where a lookup in the routing subsystem is performed. The routing subsystem mainly assigns the *skb→_skb_refdst* and it is further discussed in section 3.3.

Depending on the routing decision, the packet is either dropped with no further processing or passed to the *ip_local_deliver()* function in case the local host is the destination, or to the *ip_forward()* function in case it needs to be forwarded [45]. Packets to be delivered on the local host are passed to a higher-layer protocol handler (e.g. TCP) for further processing.

Packets that are going to be forwarded are passed to the *ip_forward()* function. This

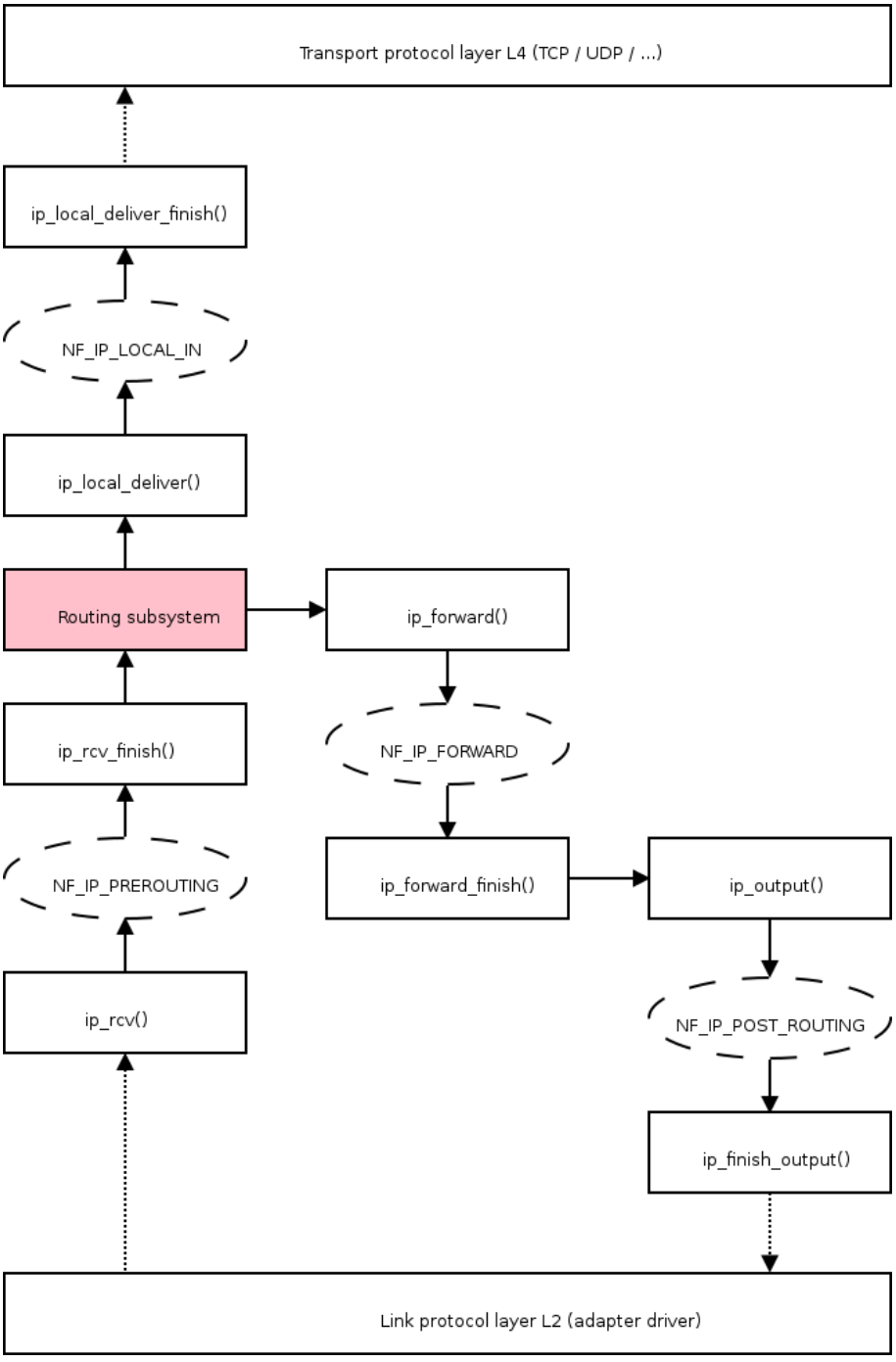


Figure 3.4: Ingress packet processing in the IPv4 stack

function checks and decrements the value of the Time To Live (ttl) field in the IPv4 header. If it reaches 0, the packet is discarded and an ICMPv4 message with „TTL Count Exceeded“ code is sent back [45]. Moreover, each time a packet is being forwarded and the TTL is decremented by 1, the checksum of the IPv4 header must be recalculated, as its value depends on the IPv4 header, and the TTL is one of the IPv4 header members. These tasks are done by the *ip_forward()* function [45]. The output processing function for the *skb* is further set to *ip_output()*. The packet is then passed to the *ip_forward_finish()* function, which updates forwarding statistics and invokes the output processing function [45].

The *ip_output()* function updates transmission statistics and assigns the output device to the *skb→dev* member. The packet is then passed to the *ip_finish_output()* function, which must fragment the packet in case it is larger than the MTU of the *skb→dev* device. The function further takes care of neighboring using Address Resolution Protocol (ARP) [22]. The neighboring subsystem is outside the scope of this thesis, but in case the link-layer address of the destination is not known, the packet transmission can be significantly delayed.

3.3 Routing subsystem

Routing takes place on L3, so it is entirely kernel's responsibility to forward packets to their destination. For each packet, incoming or outgoing, a lookup in the routing subsystem is performed [45]. The decision about whether a packet should be forwarded and which interface it should be sent on is done based on the result of the lookup in the routing subsystem. The routing subsystem is a component of the kernel's IP stack and is responsible for forwarding packets and maintaining the forwarding information base (FIB) [45].

Familiar routing daemons, such as Quagga or Bird, are entirely user-space applications. They are not responsible for routing any packet. Instead, these routing daemons manipulate the kernel's FIB to contain the selected routes based on the routing protocol and algorithm they use (OSPF link-state, BGP best path, etc.). To use these protocols and algorithms, these user-space daemons usually maintain routing tables of their own, which should not be confused with the FIB used by the kernel [45]. The kernel's FIB is manipulated from user-space either by *ioctl()* or by modern *Netlink sockets* [33].

Advanced routing topics, such as multipath and multicast routing are not covered in this thesis. Multipath routing provides ability to add more than one nexthop to a route [45]. Otherwise only one nexthop can be specified for a destination. Multicast routing provides ability to route packets destined to multicast addresses [45].

The Linux kernel supports up to 255 routing tables that can be used for policy routing. However, the use of multiple routing tables can make a router very complex and therefore policy routing is beyond the scope of this thesis. With no policy routing, there are two routing tables created by the kernel while booting - the *local* FIB table and the *main* FIB table. The *local* table contains routing entries of local addresses. Routing entries can be added to the local table only by the kernel. Adding routing entries to the *main* table is done by a system administrator or by routing daemons or as a result of an ICMP Redirect message [45].

The routing entries of the kernel's FIB table are organised as a Trie structure. The routing lookup in the Linux kernel uses longest matching prefix lookup algorithm called FIB TRIE (also known as LC-trie), which performs good for large routing tables. The routing lookup can consume much of CPU time, depending on the size of the FIB table. It can also consume much of the memory as the algorithm is rather complex [45].

A lookup is done by the *fib_lookup()* function, defined in the *include/net/ip_fib.h* file of the kernel source code [22]. When the *fib_lookup()* function finds a proper entry in the FIB table, it builds a *fib_result* object, which consists of various routing parameters, including the next hop associated with the outgoing interface [45]. Listing 3.2 shows implementation of *fib_lookup* when multiple routing tables configuration is disabled.

```
int fib_lookup(struct net *net, const struct flowi4 *flp, struct fib_result *
    res)
{
  struct fib_table *table;

  table = fib_get_table(net, RT_TABLE_LOCAL);
  if (!fib_table_lookup(table, flp, res, FIB_LOOKUP_NOREF))
    return 0;

  table = fib_get_table(net, RT_TABLE_MAIN);
  if (!fib_table_lookup(table, flp, res, FIB_LOOKUP_NOREF))
    return 0;

  return -ENETUNREACH;
}
```

Listing 3.2: Implementation of the fib_lookup() function

The *flowi4* object consists of fields that are important to the IPv4 routing lookup process, including the destination address, source address, Type of Service (TOS), and more [45]. In fact the *flowi4* object defines the key to the lookup in the routing tables. For IPv6 there is a parallel object named *flowi6*. Both are defined in the *include/net/flow.h* file of the kernel source code [22]. The *fib_lookup()* function first searches the *local* FIB table. If the lookup fails, it performs a lookup in the *main* FIB table [45]. If that fails as well, an error code representing network unreachable is returned.

After a lookup is successfully done, a destination entry object is built and associated with the *skb* [45]. The destination entry object is implemented by *struct dst_entry*, defined in the *include/net/dst.h* file of the kernel source code [22]. The result of the lookup is referenced by the *struct fib_result *res* pointer, which indirectly references the created *dst_entry* object.

The most important members of the *dst_entry* structure are two callbacks named *input* and *output*. These callbacks are assigned to be proper handlers according to the routing lookup result. For incoming unicast packets destined to the local host, the *input* callback is set to *ip_local_deliver()*, and for incoming packets that should be forwarded, this input callback is set to *ip_forward()*. For a packet generated on the local machine and sent away, the output callback is set to *ip_output()* [45]. Listing 3.3 shows a part of the *struct dst_entry* definition.

```
struct dst_entry {
  . . .
  int (*input)(struct sk_buff *);
  int (*output)(struct sk_buff *);
  . . .
}
```

Listing 3.3: Destination callback members of struct dst_entry

A reference to the *dst_entry*, which was created as a result of the routing decision, is assigned to the *skb→_skb_refdst* member. The *ip_rcv_finish()* function further calls the

input callback which passes the *skb* either to the local host processing or to forwarding, as described in section 3.2. In case of forwarding, the *output* callback is further set to the *ip_output* function by the *ip_forward_finish()*, as described in section 3.2,

In terms of performance, there is currently not much space for improvements as each *skb* must be handled separately and must be passed to the described functions of the IP stack. In kernels prior to 3.6, there was an IPv4 routing cache with a garbage collector [45]. The IPv4 routing cache was removed in kernel 3.6 (released in July 2012), as it proven to be inefficient and vulnerable to DoS attacks [36].

There used to be a feature called Fastroute that allowed device drivers to route incoming traffic during interrupt context using a small cache. The packets were forwarded to the outgoing interface without having to pass through the higher layer (IP) [45]. However, this feature is not compatible with other important features, such as Netfilter firewall or advanced routing, for the simple reason that this low-level forwarding would bypass them. Starting with the 2.6.8 kernel, Fastroute is no longer supported and its implementation was removed from the Linux kernel [45].

As discussed above, the L3 packet processing and routing handle packets associated with their own *skb*. However, the lower-layer part of the networking code can provide significant improvements before the *skb* enters the *ip_rcv()* function. The improvements depend on how this code handles the ingress frames.

3.4 Ingress traffic processing

The traditional way of processing frames from NIC is interrupt-driven [45]. Each incoming frame is an asynchronous event which raises a hardware interrupt. Interrupt handlers run asynchronously with either the current interrupt level disabled or with all interrupts disabled [34]. The handlers could interrupt other potentially important code, therefore they need to run as quickly as possible. Interrupt handlers immediately respond to hardware and perform time-critical actions, however, other less critical work should be deferred to a later point when interrupts are enabled [34].

Upon frame reception, the hardware interrupt handler of the network adapter performs the following immediate tasks: [5]

1. Copies the frame into an *sk_buff* data structure. If DMA is used by the device, the kernel needs only to initialise a buffer and pass its descriptors to the driver, which instructs the device to use DMA. The received frame is then copied by a DMA transfer.

2. Initialises some of the *sk_buff* members for later usage by upper network layers, notably the *protocol* member, which identifies the higher-layer protocol handler and will play a major role later.

3. Updates some other parameters private to the device, such as variables for statistical purposes.

4. Signals the kernel about the new frame by scheduling the NET_RX_SOFTIRQ softirq for execution.

To keep the execution of the handler as short as possible, further frame processing is performed later in the NET_RX_SOFTIRQ routine. This softirq routine actually performs an interrupt-related work not performed in the hardware interrupt handler [5]. The routine further passes the received frame to the corresponding L3 protocol handler according to the

protocol member of the *skb* [34]. In case of IPv4, this is the *ip_rcv()* function described in section 3.3. Moreover, the softirq routine is threaded and can run concurrently on different CPUs [34].

However, such method of packet processing became insufficient with the emerge of high-speed network cards. Even a moderately busy interface can handle thousands of packets per second and per-packet interrupts quickly overwhelm the processor with interrupt-handling work [16]. On the way towards high-speed packet processing on the host CPU, packet processing in the network stack must have been adapted.

NAPI („New API“, though not so new anymore) is an interrupt mitigation mechanism used with network devices [12]. NAPI mixes interrupts with polling and gives higher performance under high traffic load than the old approach, by reducing significantly the load on the CPU [5].

3.4.1 NAPI

NAPI was first introduced during the Linux kernel 2.5 development cycle as an extension to the device driver packet processing framework, which is designed to improve the performance of high-speed networking [26].

A NAPI-compliant device driver must implement a *poll()* function used by the kernel to fetch the received frames. The first frame received causes a hardware interrupt and its handler to run as usual. In the handler, however, the driver disables interrupts from the device and calls the *netif_rx_schedule()* function. This function adds the device to the kernel's *poll_list* and schedules the NET_RX_SOFTIRQ routine. From now on, the task of delivering more incoming frames from the device's queue is delegated to the kernel [5].

In the NET_RX_SOFTIRQ routine, the kernel iterates over the *polll_list* and calls the *poll()* function of the device driver to fetch the frames from the device's ingress queue (receive ring buffer). The kernel fetches the packets and passes them to the higher-layer protocol handler for further processing [45]. The *poll()* function is called with a maximum number of packets (*budget*) it is allowed to feed into the kernel. It should process up to that many packets and return [12]. Figure 3.5 shows the NAPI workflow described above.

When the kernel is ready to deal with more packets, the *poll()* function of the next device in the *poll_list* will be called. The scheduled devices are probed in a round-robin manner. The total number of packets fetched from devices in the *poll_list* is limited. If it was not sufficient to serve all devices in the *poll_list* and the kernel should release the CPU, the devices have to wait for the next NET_RX_SOFTIRQ run [5]. The softirq NET_RX_SOFTIRQ processing for NAPI-compliant drivers is implement by the *net_rx_action()* function defined in *net/core/dev.c* [22]. The function overview is shown in figure 3.6.

When a device driver uses NAPI, it is up to the driver to implement any congestion control mechanism. This is because ingress frames are kept in the NIC's memory or in the receive ring buffer managed by the driver, and the kernel cannot keep track of traffic congestion [5]. When the host becomes congested, the packets are lost because of not enough space in the ring buffer. The packets that are going to be lost are not fed into the network stack, so they take no CPU time [44].

When a device cannot clear out its ingress queue in a single poll, it has to wait until the next call. The kernel keeps calling the driver's *poll()* function until it empties the device's ingress queue out [5]. At that point, there is no need anymore for polling. The device is removed from the *poll_list* and the device driver can re-enable interrupt notifications for the device [5]. Nowadays, almost every driver supports the NAPI feature [45].

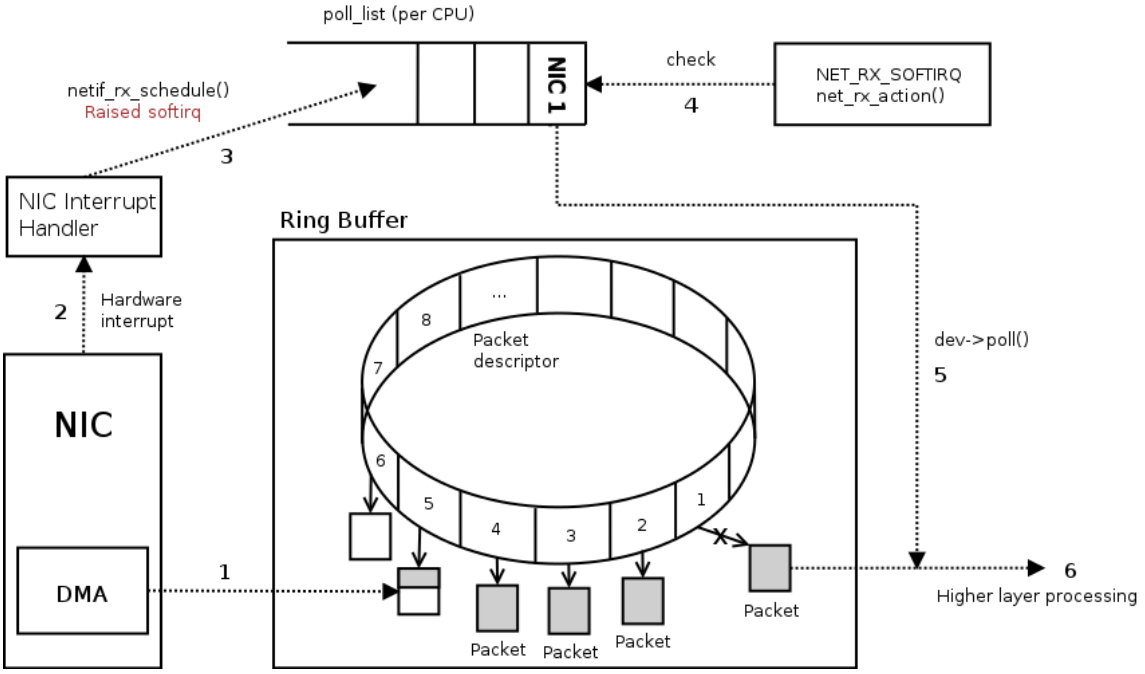


Figure 3.5: NAPI workflow

NAPI reduces interrupt load on the system and lowers the CPU utilisation under heavy load, but it increases latency as packets are not processed as quickly [26]. User-space applications that need the lowest possible latency and are willing to pay a cost of higher CPU utilisation, can use a capability for busy polling on sockets (called Low Latency Sockets), which was added in kernel 3.11 [45]. Low Latency Sockets eliminate the cost of the interrupt and context switch and provide latency very close to the hardware latency [18].

In addition to NAPI, various offload engines were designed to take over some responsibilities of the networking code and implement them in hardware.

3.4.2 Receive offloads

Network adapter vendors have been adding protocol support to their cards. This support can vary from the simple (checksumming of packets, for example) through to full TCP/IP implementations [11].

To mitigate CPU load spent on TCP overhead completely, the TCP Offload Engine (TOE) is implemented in several NICs. TOE features a full TCP/IP implementation in adapter, including the TCP connection management. However, Linux has never supported the TOE features of any network cards [11]. Vendors have made modifications to the Linux kernel to support TOE, and these changes have been submitted for kernel inclusion but were rejected [27].

Linux kernel engineers currently feel that the full network stack offload that TOE provides has little merit [27]. TOE shorts out much of the Linux networking code, described in the previous sections. In the process, it cuts out features like Netfilter, traffic control, and more. The Linux networking stack is easy to fix when a bug or security issue comes up [11]. If a security problem turns up in a TOE adapter, instead, there is very little which can be done to fix it. Linux engineers claim that 100 Mbps TOE adapters (which used

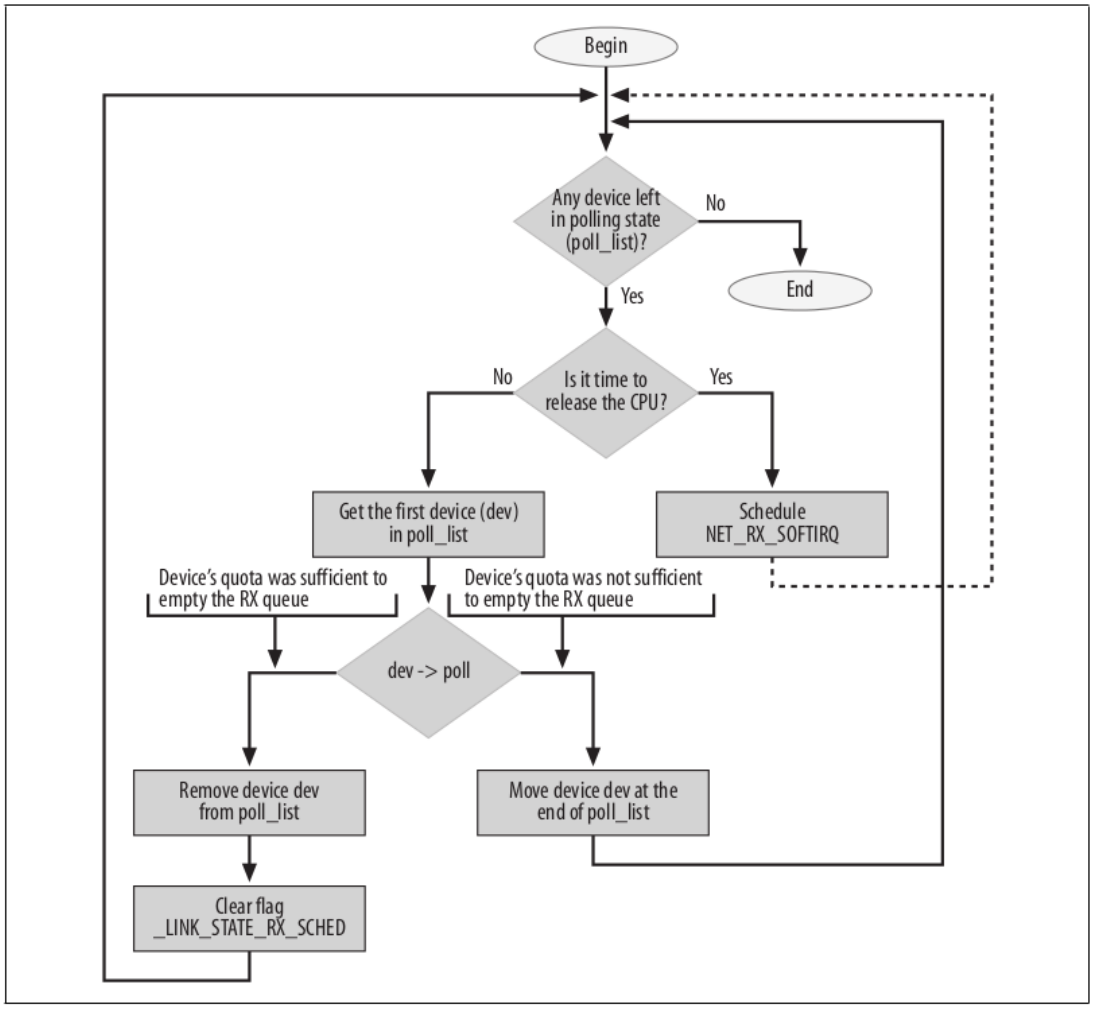


Figure 3.6: Softirq NAPI routine *net_rx_action* (source: [5])

to be the bleeding-edge high speed) are now slower than the Linux networking stack. So any performance advantage from TOE is a temporary thing, but once the TOE's code is merged, it must be supported. As a result of this, TOE support might become a long-term maintenance burden [11].

Although inclusion of TOE support was rejected, there are ways to obtain TOE's performance without necessitating stateful support in the cards [11]. Everything that is worthwhile can be done with stateless offloads. One of the simplest stateless offload technique is computing checksums in hardware. With receive (Rx) checksum offload, IP, TCP and UDP checksums are checked in the hardware of the NIC upon frame reception.

While checksum offload provides some performance improvement, a large portion of per-packet processing overhead remains. Each packet is passed through the entire IP stack, as described in section 3.2. Dealing with each single packet takes a significant amount of the CPU time, particularly on high-speed Ethernet links that can produce millions of packets per second.

Given the importance of per-packet overhead, it makes sense to raise the MTU. However, most connections of interest go across the Internet, and those are all bound by the lowest MTU in the entire path. As noted in section 2.3, the IEEE has determined no support for frames with MTU larger than 1500. Protocol-based mechanisms for MTU discovery exist, but they do not work well on the Internet, because in particular, a lot of firewall setups do not allow them to work [14].

If the kernel can not use a larger MTU, it can pretend that it is using a larger MTU. An optimisation technique to pretend larger MTU is provided by the Large Receive Offload (LRO). LRO merges packets of the same TCP flow together, creating one large super-frame, before it is passed to the higher network layers [14]. Merging multiple packets and processing them as a single packet reduces CPU overhead and thus improves performance [45]. The merging can be done either in the driver or in the hardware. Even LRO emulation in the driver has performance benefits [14].

Since LRO merges everything of the same TCP flow into one large super-frame, the differences in the headers of these packets are lost [14]. If a system is acting as a router, it should not be changing the headers on packets as they pass through, because it brakes the end-to-end principle and can significantly impact performance [45].

A generic solution was introduced by the Generic Receive Offload (GRO) to mitigate the problems of LRO. In GRO, the criteria for which packets can be merged is greatly restricted. The MAC headers must be identical and only a few TCP or IP headers can differ - checksums are necessarily different and the Identification field is allowed to increment [14]. As a result of these restrictions, merged packets can be later resegmented losslessly and therefore the GRO feature can be used by a system acting as a router without braking the end-to-end principle [14]. However, GRO still requires the L4 protocol to have its own segmentation support and it is currently restricted to TCP only [45].

When using GRO, merging packets of the same flow into one large super-frame must be time-limited. In combination with NAPI, there is no need for any special waiting code - the kernel already calls the driver's poll method for new packets occasionally and processes them in batches. Thus, GRO can simply be performed at NAPI poll time without introducing any additional latency [14]. The GRO feature improves network performance and it deprecated LRO in recent kernels [45]. Figure 3.7 shows comparison of ingress frames processing with and without the above described offload mechanisms.

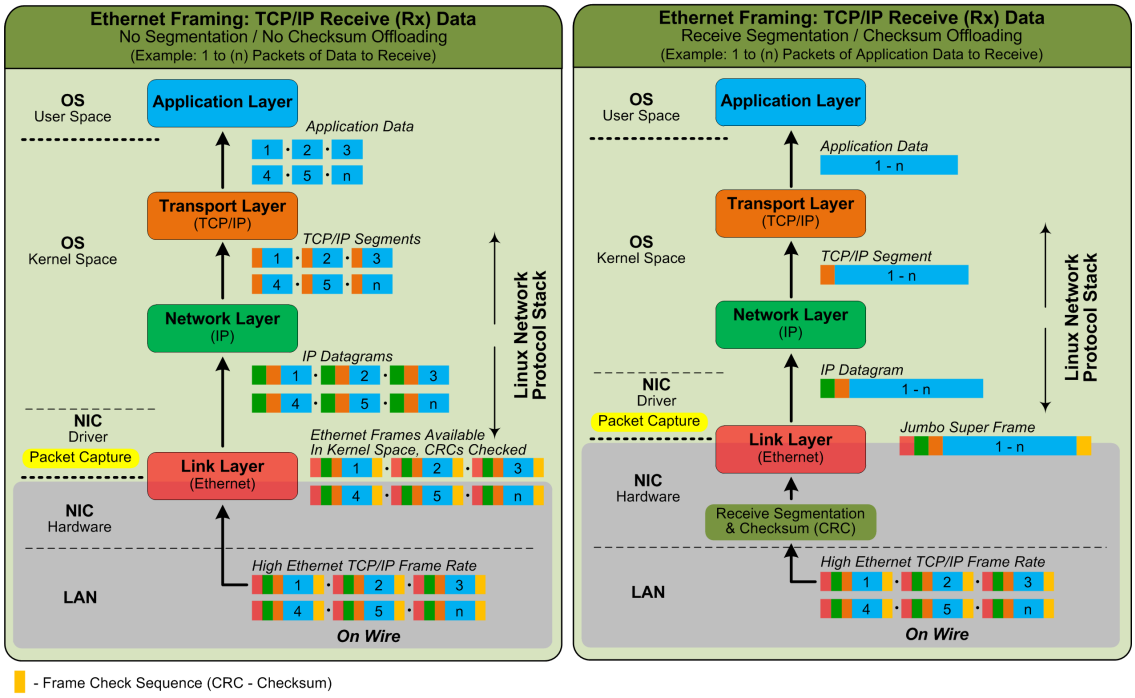


Figure 3.7: Receive offloads (source: [58])

3.5 Egress traffic processing

The previous sections described how the frame reception works and the processing path when the routing subsystem decides to forward them. After the routing decision is made and the *skb* is passed to the *ip_finish_output()* function, it is further passed back to the link layer of the networking stack. This part of the Linux networking also provides interface to the device drivers and handles traffic control [5].

In reference to the egress traffic processing, there are two important functions in this part of the stack - *dev_queue_xmit()* and *dev_hard_start_xmit()*. The *ip_finish_output()* function of the IPv4 stack passes the outgoing *skb* to the *dev_queue_xmit()* function, which determines whether the device is queueless of queueful. If the device is queueless, such as Loopback or Virtual interface, then the *skb* is passed to *dev_hard_start_xmit()* directly without any traffic control mechanism involved. The *dev_hard_start_xmit()* function further prepares the *skb* for transmission and passes it to the transmission function of the device driver, which instructs the device to transmit the frame on the wire [5].

If the device is queueful, such as almost any hardware network adapter, *dev_queue_xmit()* executes the traffic control first [5]. The traffic control implemented in the Linux kernel uses algorithms known as queuing disciplines (often abbreviated as *qdisc*) to arrange the frames in the desired order for transmission [5]. Figure 3.8 shows a brief overview of this part of the stack.

The Linux kernel supports various queuing disciplines, that can be configured by the *tc* utility. The default queuing discipline for every network device is *pfifo_fast* [45]. The *pfifo_fast* is a three-band First In First Out discipline. As long as there are packets waiting in band 0, band 1 will not be processed. Similarly, packets from band 1 are always processed

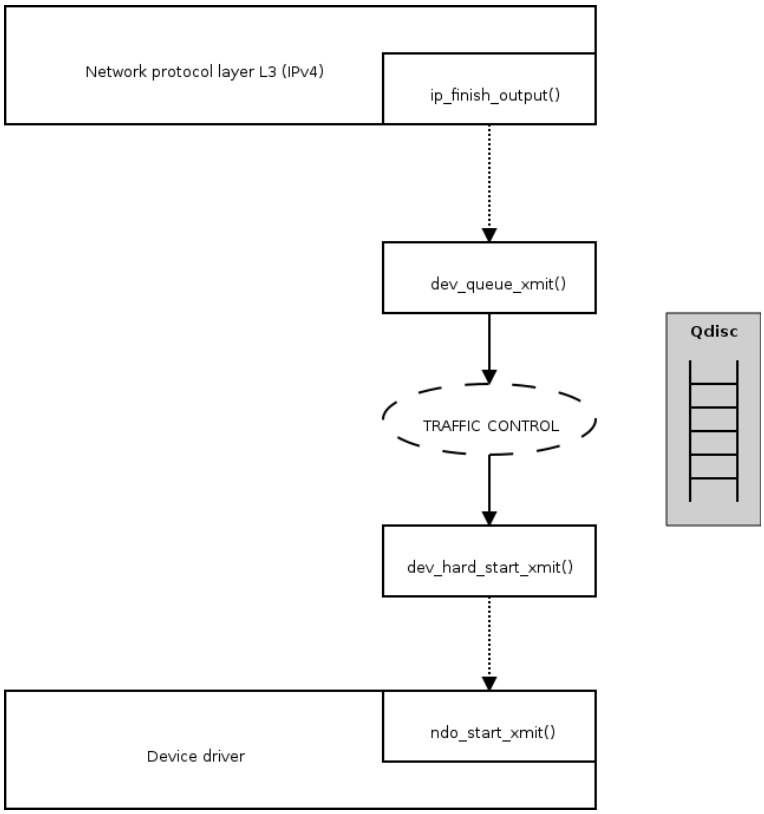


Figure 3.8: Egress packet processing

before packets from band 2. Within each band, a simple FIFO rule apply. The kernel inserts the packets to one of the band according to the Type of Service flag of the packet [46]. A more detailed discussion of the traffic control and its queuing disciplines is outside the scope of this thesis.

After the packet has been selected for transmission by the traffic control, it is passed to the *ndo_start_xmit()* function, which is implemented by the device driver. The packet is inserted to the hardware transmit queue by the driver and transmitted. The hardware transmit queue is implemented as a ring buffer (TX ring) and the DMA controller of the device uses it to fetch the egress frames [22].

The adapter uses interrupts to notify the kernel when the transmission fails or succeeds. Similarly to NET_RX_SOFTIRQ, which performs an interrupt-related work for incoming traffic, the NET_TX_SOFTIRQ softirq handles the interrupt-related work for outgoing traffic. The transmission code uses the softirq to mitigate interrupts in a similar fashion as the reception code. Because different instances of the same softirq handler can run concurrently on different CPUs, networking code is both low latency and scalable [5].

3.5.1 Transmit offloads

Most of the adapters that support receive checksum offload also support its counter-part transmission (Tx) checksum offload. Tx checksum offload calculates TCP/UDP and IP checksums of the packets in the hardware before they are transmitted on the wire.

A counter-part of LRO is TCP Segmentation Offload (TSO). With a TSO-capable adapter, the kernel can prepare much larger packets for outgoing data (e.g. up to 64KB in case of IPv4) and the adapter will re-segment the data into smaller packets according to the MTU [14]. TSO is well supported in Linux - for systems which are engaged mainly in sending of data, it is sufficient to make 10 Gbps rate [14]. TSO reduces the necessary CPU load, bus overhead, and cache impact to send a series of packets, but it still does not require the adapter to actually know anything about specific TCP connections - the kernel still has to deal with the TCP states and ACKs [11].

The TCP Segmentation Offload is designed to work with TCP exclusively. To mitigate this issue, the Generic Segmentation Offload (GSO) was implemented. Performance improves even if the feature is emulated in the driver [14]. Figure 3.9 shows comparison of egress packet processing with and without the above described offload mechanisms.

3.6 Multiqueue adapters and scaling

One of the fundamental data structures in the networking subsystem is the transmit queue associated with each device. As described in section 3.5, the core networking code will call the driver's *ndo_start_xmit()* function to let the driver know that a packet is ready for transmission [13]. The driver feeds that packet into hardware's transmit queue, which results in a data structure which is shown in figure 3.10.

This scheme has worked well for years, but it does not map well to devices which have multiple transmit queues. The multiqueue devices need each transmit queue to be scheduled independently. 10 and 40 Gigabit Ethernet devices with multiple transmit queues are very common [13].

To provide an independent scheduling of a transmit queue, a new *netdev_queue* structure is implemented in the Linux kernel. The *netdev_queue* structure encapsulates all of the information about a single transmit queue, and it is protected by its own lock. Multiqueue

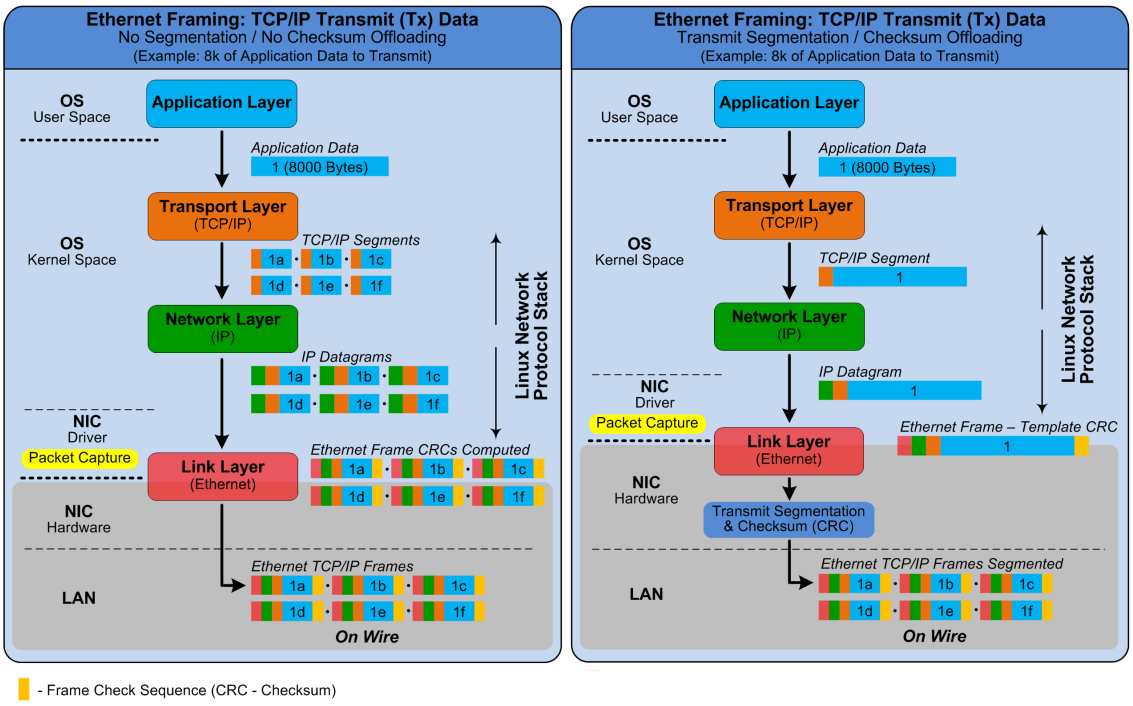


Figure 3.9: Transmit offloads (source: [58])

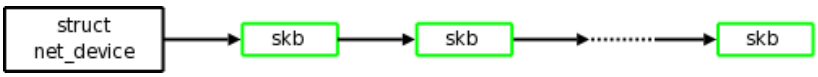


Figure 3.10: Single queue device (source: [13])

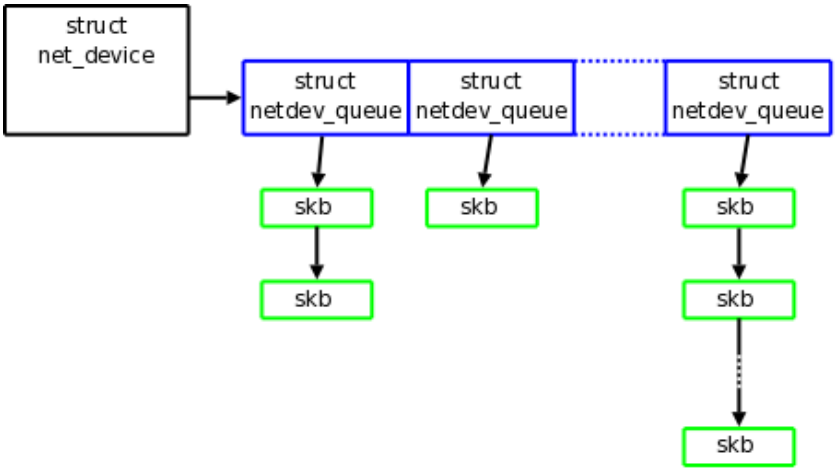


Figure 3.11: Multiqueue device (source: [13])

device drivers set up an array of these structures according to the number of queues. The *mq* (multiqueue) queueing discipline uses the array to attach a specific *qdisc* to each queue. The *mq* discipline is a dummy scheduler, which is used by default for multiqueue devices instead of the regular *pfifo_fast* discipline [19]. Figure 3.11 shows the new data structure.

In addition to multiple transmit queues, modern high-speed network adapters support multiple receive queues as well [54]. The principle described above also applies to the receive queues in the Linux kernel. The multiqueue support allows to scale the network load in multiprocessor systems. Such scaling is provided by processing each queue by a different CPU [28]. The NIC distributes packets to the queues by applying a filter that assigns each packet to one of the receive queues. The filter is a hash function (usually Toeplitz hash algorithm) over the network and transport layer headers of the packet and a hash key, which is stored in the NIC's memory. This mechanism is generally known as Receive Side Scaling (RSS) and its goal is to increase performance uniformly [28].

Network adapters that do not support RSS but have multiple receive queues can still benefit from multiprocessor scaling by using Receive Packet Steering (RPS), which is a software implementation of RSS [28]. However, there are some disadvantages of RPS against the hardware-based RSS. The calculation of the hash requires accessing data from the packet header. That access will necessarily involve one or more cache misses on the CPU running the RPS code - that data was just put there by the network interface and thus cannot be in any CPU's cache [15]. Once the packet has been passed over to the CPU which will be doing the real work, that cache miss overhead is likely to be incurred again. Moreover, the targeted CPU is notified by an inter-processor interrupt, which introduces another overhead. If the NIC supports RSS and it is configured to map each hardware receive queue to a single CPU, then RPS is redundant and unnecessary [28].

The network transmission scaling is implemented by the Transmit Packet Steering in the Linux kernel. Transmit Packet Steering (XPS) is a mechanism for intelligently selecting which transmit queue to use when transmitting a packet on a multiqueue device [22]. To accomplish this, there is a configurable mapping from CPU to hardware queues. The goal of the mapping is usually to assign the queues exclusively to a subset of CPUs, which reduces contention on the device queue lock since fewer CPUs contend for the same queue. Contention can be eliminated completely if each CPU has its own transmit queue. Moreover, the cache miss rate on transmit completion is also reduced since a particular CPU is serving

just a subset of transmit queues [28].

Apart from load distribution, the above described mechanism also minimise cache miss rates when configured properly. The most significant is a cache miss of a hardware interrupt handler for the particular queue, queue lock cache miss and packet metadata in its *sk_buff*. RPS and RFS were introduced in kernel version 2.6.35 and XPS in 2.6.38 [28].

To complete the list, there are two additional network scaling mechanisms implemented in the Linux kernel - Receive Flow Steering and Accelerated Receive Flow Steering. Both provide assigning incoming packets to the CPU that the destined user-space application runs on. The difference between them is that the Accelerated Receive Flow Steering is implemented in the NIC's hardware, whereas Receive Flow Steering is a software implementation. Since these two mechanisms provide network scaling to user-space applications, they will not be discussed further in this thesis [28].

Chapter 4

Analysis

Basically, there are three important parts needed to perform network benchmarks at full 40 Gbit speed - a network interface card with 40 Gigabit Ethernet support, a server compatible with the card and a packet generator capable of generating 40 Gbps network traffic. A chosen distribution of the GNU/Linux operating system should contain a stable and recent kernel that supports the 40 Gb Ethernet protocol, the network interface card and the features described in chapter 3.

4.1 Hardware equipment

The network interface card used in the experiments is Mellanox ConnectX-3 EN QSPF dual-port PCI-E 3.0 x8 MCX314A-BCBT [54]. The card was provided by the Faculty of Information Technology, Brno University of Technology. Mellanox ConnectX-3 EN is an adapter that can run 10 Gigabit Ethernet and 40 Gigabit Ethernet. It also supports non-standard 56 Gbps link speed when connected to Mellanox switches. The card is PCI-Express 3.0 x8 compatible with support for previous PCI-Express versions. Mellanox ConnectX-3 EN is a multiqueue NIC with MSI-X support up to 16 receive queues per port featuring Receive Side Scaling with hashing support for both IPv4/IPv6 and TCP/UDP flows [56, 55]. Figure 4.1 shows the block diagram of the network interface card.

The Mellanox NIC requires PCI-Express 3.0 x8 slot to take full advantage of its speed. Brno University of Technology provided a server with the Supermicro X10DRU-i+ motherboard, which features PCI-Express 3.0 slots compatible with the Mellanox Connect-X 3 EN adapter [53]. The server is further equipped with two Intel Xeon E5-2660 v3 processors at 2200MHz with 10 physical cores per CPU and 20 logical cores per CPU when Hyper-Threading is enabled. Each CPU has 20 MB shared L3 cache and PCI Express 3.0 support with up to 40 lanes. The processor features Direct Data I/O technology (also known as Direct Cache Access), which optimises cache access for networking purposes by putting the ingress packtes directly to the CPU cache [17].

There are various software frameworks for high-speed packet generation, such as pktgen[1] or Netmap[2]. Pktgen is an upstream component of the Linux kernel, but at the time of writing it is not capable of generating even full 10 GbE frame rate [43]. However, it can be combined with other frameworks for fast packet processing, such as Intel's Data Plane

[1] https://www.kernel.org/doc/Documentation/networking/pktgen.txt
[2] http://info.iet.unipi.it/~luigi/netmap/

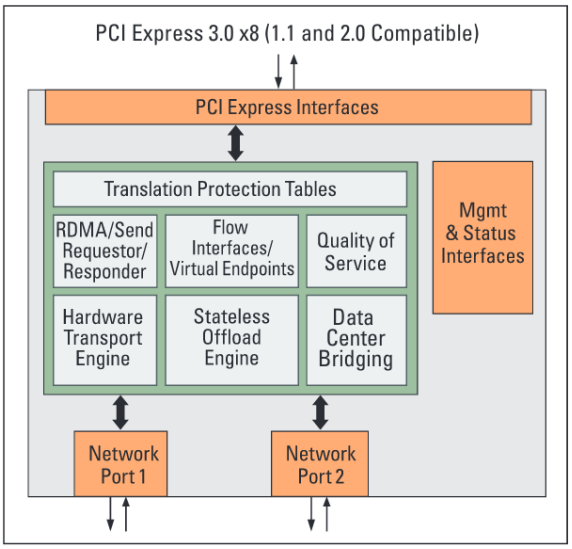


Figure 4.1: Mellanox ConnectX-3 EN block diagram (source: [56])

Development Kit[3]. Netmap is not an upstream component, but it provides patches to the Linux kernel. Netmap claims to generate 14.88 million frames per second, which is a full frame rate of 10 Gigabit Ethernet [43]. However, Netmap was not tested against 40 GbE full frame rate of 59.5 million frames per second, as calculated in section 2.1. Although both frameworks seem promising, their benchmarking and description are outside the scope of this thesis. Moreover, to perform the measurements with a software-based packet generator, another GNU/Linux server and a 40 GbE NIC is needed.

Another solution is to use a hardware-based packet generator such as Spirent [9]. With kind permission of CESNET, the Czech national research and education network operator, the Spirent SPT-3U equipped with a combined 100Gb / 2x40Gb Ethernet module was used to perform the measurements. The Spirent packet generator supports generation of custom Layer 2-7 traffic, custom frame length and various predefined traffic patterns with variable frame length called Internet Mix (iMix). These patterns represent a typical distribution of frame lengths found in the Internet traffic and they can be further customised. Spirent SPT-3U further supports configuration of custom frame rates and bandwidth use [9].

4.2 Software equipment

The Red Hat Enterprise Linux 7 operating system features 40 Gb Ethernet support and the kernel based on upstream version 3.10 [42]. To avoid licensing fees, CentOS 7 can be installed. CentOS 7 provides support for the 40 Gigabit Ethernet protocol and a binary compatible kernel with RHEL 7 [57]. Since the kernel version 3.10 was released in 2013, the latest upstream kernel can be installed to provide an additional comparison. The latest upstream kernel can be either compiled directly from the source code or downloaded from a third party repository. The ELRepo repository contains the *kernel-ml* package with the latest upstream kernel version, which is 4.0.2 at the time of writing [40].

[3] http://dpdk.org/

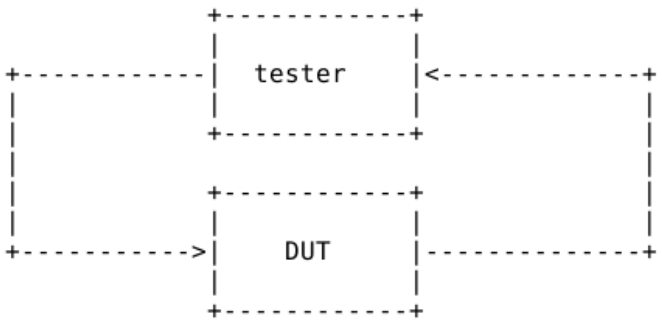


Figure 4.2: RFC2544 test implementation (source: [6])

Mellanox ConnectX-3 EN is supported by the mlx4 driver found in the *drivers/net/ethernet/mellanox/mlx4* directory of the Linux kernel source code. It is the low level driver implementation for the Connect-X adapters designed by Mellanox Technologies. Some Connect-X adapters can operate as an InfiniBand adapter and as an Ethernet NIC. To accommodate the two flavors, the driver is split into modules mlx4_core, mlx4_en and mlx4_ib. The mlx4_core module handles low-level functions like device initialisation and firmware commands processing. The mlx4_en module handles Ethernet specific functions and plugs into the network device layer of the Linux kernel. Similarly, the mlx4_ib module handles InfiniBand specific functions [55]. By default, the driver uses adaptive interrupt moderation for the receive path, which adjusts the interrupt moderation to the traffic pattern [55].

Spirent TestCenter Application version 4.46 is provided to use the Spirent hardware packet generator. The application allows to create virtual devices connected to selected ports. The devices can use a full-featured IPv4 or IPv6 stack, including ARP/ND, ICMP, TCP/UDP etc [10].

4.3 Benchmarking methodology

Procedures described by RFC 2544 can be used to measure routing performance of the Linux kernel. RFC 2544 specifies the benchmarking methodology for network interconnect devices. The ideal way to implement the series of tests described in RFC 2544 is to use a tester with both transmitting and receiving ports. Connections are made from the sending ports of the tester to the receiving ports of the device under test (DUT) and from the sending ports of the DUT back to the tester [6]. Figure 4.2 shows the test implementation.

Since the tester both sends the test traffic and receives it back, after the traffic has been forwarded by the DUT, the tester can easily determine if all of the transmitted packets were received [6]. The Spirent TestCenter Application provides statistics about transmitted and received frames, which can be used for this purpose.

4.3.1 Traffic generation

The RFC 2544 specifies the following frame sizes to be used on Ethernet: 64, 128, 256, 512, 1024, 1280 and 1518 [6]. However, at least 66 B frame size must be used in case of transmitting UDP over IPv6 - the size of L2 header is 14, the size of CRC is 4, the size of IPv6 header is 40 and the size of UDP header is 8.

In addition to the specified frame sizes, a custom frame size distribution can be defined

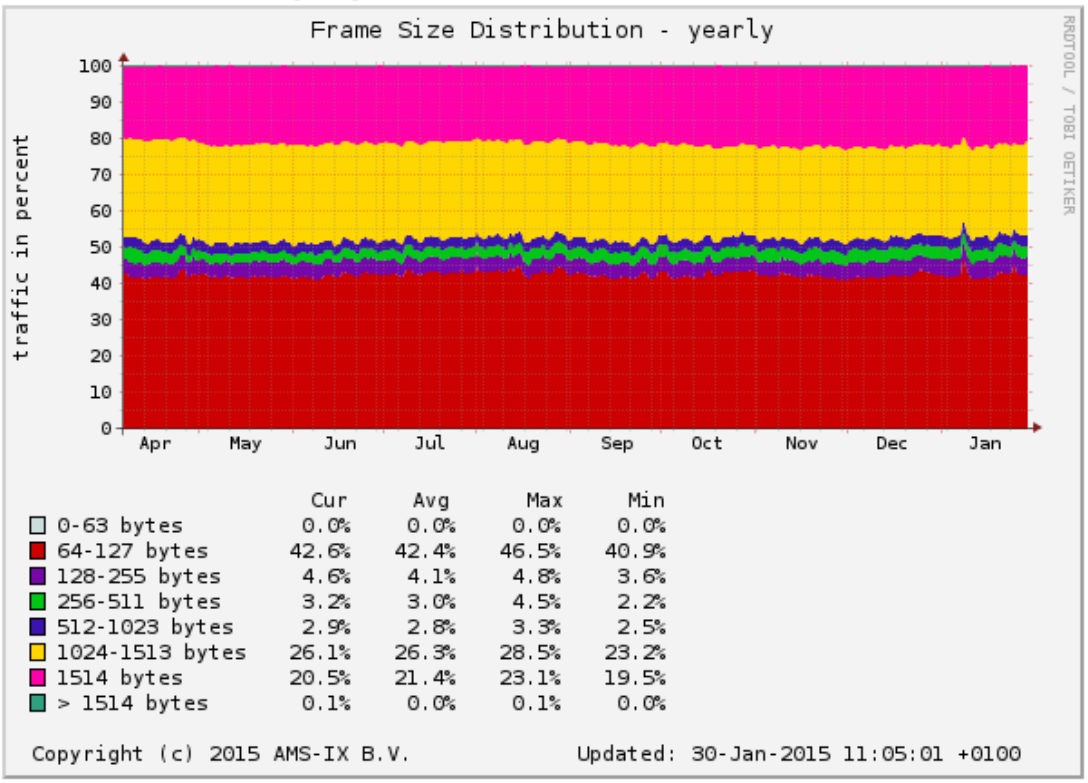


Figure 4.3: Yearly frame size distribution at AMS-IX (source: [31])

for the purpose of a real internet traffic simulation. The Amsterdam Internet Exchange (AMS-IX) provides statistics of the frame size distribution in the Internet traffic [31]. Figure 4.3 shows yearly frame size distribution provided by AMS-IX. This distribution can be configured in the Spirent TestCenter Application, however, to use the same iMix for both IPv4 and IPv6, the minimum frame size must be increased to 66 as described above. To avoid an unfair packet scheduling by the server, all packets should be assigned the same Type of Service flag.

Unfortunately, the provided Spirent TestCenter Application does not contain license to configure a device participating in TCP streaming. This constraint could be workarounded by sending TCP packets with no flags set, however such configuration is also not possible. The generated TCP packets always contain TCP SYN flag, which bypasses the Generic Receive Offload described in subsection 3.4.2. Therefore, the TCP packet processing is the same as in case of UDP.

The Spirent TestCenter Application allows to configure exact frame rate or bandwidth use. The measurements should distinguish at least between 50 000 frames per second or 1% of bandwidth use. Each measurement takes 60 seconds and it is repeated 3 times. If the kernel is able to forward all frames in at least one of the 3 measurements, the measurement is successful. This is to determine whether the kernel is able to forward such amount of traffic. Some network unrelated tasks performed by the kernel may cause the inability to forward all frames, such as gathering statistics, memory management, etc.

4.3.2 Statistics collection

The Spirent TestCenter Application is able to display counters of transmitted and received frames on each interface. The counters can be used to determine whether all of the transmitted packets on one interface were received on the other interface and hence successfully forwarded by the server. Unfortunately, the provided Spirent TestCenter Application contains no licence to perform the RFC 2544 throughput test automatically, so the measurements must be configured manually in the Spirent TestCenter application. The manual configuration consists of configuring the transmit rate, observing the packet counters and comparing their values. If the server forwards packets without a single loss, the transmit rate can be increased and the test repeated.

The proc filesystem provides access to several statistics as well. The network statistics exported via proc filesystem can be found in the /proc/net directory. The files in this directory are read-only and cannot be manipulated using the *sysctl* utility. There are 2 important files for the measurements - the *fib_trie* and *fib_triestat*. The *fib_trie* file exports the kernel's Forwarding Information Base overview. The file describes both the Main and Local FIBs, as described in section 3.3. The *fib_triestat* file exports metadata of the FIB trie structures, such as average depth, maximum depth, number of leaves, etc [22].

The CPU utilisation can be observed using the *perf* utility, which can be found in the tools/perf directory of the Linux kernel source code. Although perf is included in the Linux kernel source code, it must be installed separately on most GNU/Linux distributions, including CentOS 7. To obtain additional statistics, such as PCI-Express utilisation or memory utilisation, the Intel Performance Counter Monitor (PCM)[4] can be used. The PCM package is not included in the official CentOS 7 repository, but it can be compiled directly from the source code. Non-maskable interrupt watchdog must be disabled in order to run Intel PCM. The non-maskable interrupt watchdog can be disabled by writing „0“ to the /proc/sys/kernel/nmi_watchdog file.

4.4 Software settings

The CentOS 7 operating system features various components that influence forwarding performance. The performance of the CentOS 7 distribution kernel is measured in this thesis. Further measurements also take the latest upstream kernel 4.0.2 into account [22].

To measure a bare routing performance of the Linux kernel, the Netfilter and SELinux components should be disabled. If disabling the Netfilter is not appropriate, the *iptables* utility must be used to configure the Netfilter to allow IP forwarding, because the default rules do not allow packet forwarding in CentOS 7. Additionally, SELinux should be disabled to prevent performance decrease. SELinux in CentOS 7 uses Enforcing policy by default. An influence of both the components on forwarding performance can be measured.

The Linux kernel features dynamic CPU frequency scaling. The CPUFreq governors are policies that decide what frequency should be used. The *performance* governor should be used during the measurements. The *performance* governor sets the CPU statically to the highest frequency available. The default governor is *powersave* in CentOS 7, which sets the CPU statically to the lowest frequency [7].

To change the Linux kernel compile-time configuration, the kernel must be recompiled. The CentOS 7 kernel provides a fair amount of features that could break existing setups

[4] https://software.intel.com/en-us/articles/intel-performance-counter-monitor-a-better-way-to-measure-cpu-utilization

when disabled. The default kernel compile-time configuration does not have to provide the best routing performance, however, it is usually used in most scenarios and hence its benchmark results are of interests for most people.

4.4.1 CentOS 7 kernel compile-time configuration

When the Linux kernel is compiled with support for symmetric multiprocessing with the CONFIG_SMP option and runs on a multiprocessor system, the code for receiving and transmitting packets takes full advantage of that. Every modern GNU/Linux distribution compiled for the AMD64 architecture has the option enabled, including the CentOS 7 [5]. Advanced networking features enabled in compile-time does not have to be used in run-time. For example, the MULTIPLE_IP_TABLES support is enabled in the CentOS 7 distribution kernel, however, since the measurements presented in this thesis use no policy routing, the simple FIB lookup principle described in section 3.3 is still performed.

The CentOS 7 Linux kernel configuration has the CONFIG_NO_HZ_FULL option enabled. That is, the system uses adaptive ticks and there are no regular interrupts from the timer which would cause additional delays during the packet processing [24].

Apart from compile-time options, the Linux kernel configuration can be changed in run-time. The *proc* and *sys* filesystems provide access to the kernel variables that influence packet processing. Tuning of these variables can provide a significant performance improvement when configured properly.

4.4.2 Procfs settings

The variables exported via procfs are accessible as files under /proc in CentOS 7. The variables in the /proc/sys directory can be manipulated by the *sysctl* utility as well. The files in the /proc/sys/net directory are of interest for the experiments. This directory includes the following subdirectories:

- /proc/sys/net/ipv4 - contains variables influencing the IPv4 protocol settings
- /proc/sys/net/ipv6 - contains variables influencing the IPv6 protocol settings
- /proc/sys/net/netfilter - contains variables influencing the netfilter settings, not discussed in this thesis
- /proc/sys/net/unix - contains variables influencing communication over unix sockets, not discussed in this thesis
- /proc/sys/net/core - contains variables influencing low-level networking settings, including parameters of NAPI, Low Latency Sockets, etc.

The most important setting for the routing performance measurements of the Linux kernel is IPv4 forwarding. It can be enabled by writing „1“ to the /proc/sys/net/ipv4/ip_forward file. This variable is special - its change resets all IPv4 configuration parameters to their default state [23]. The /proc/sys/net/ipv4/conf/*ifname*/forwarding file can be used to further selectively enable or disable forwarding on a particular interface. Historically, some of the files in /proc/sys/net/ipv4 also influence settings of L4 protocols, such as memory limits, TCP Timestamping, Selective ACKs, etc. Although these files are located in the *ipv4* subdirectory, the L4 settings are independent on the underlying protocol. Most of

the Layer 4 settings are auto-tuned by the kernel itself and their description is outside the scope of this thesis [45].

Files in the *ipv6* directory influence the IPv6 protocol settings only. The IPv6 protocol is disabled in CentOS 7 on all interfaces by default. To enable the IPv6 protocol on all interfaces, the variable accessible via /proc/sys/net/ipv6/conf/all/disable_ipv6 must be changed to „0“. Similarly, the /proc/sys/net/ipv6/conf/all/forwarding variable must changed to „1“ to enable IPv6 forwarding on all interfaces. Both settings can be changed on per-interface basis as well.

The /proc/sys/net/ipv4/route.max_size sets the maximum number of IPv4 routes allowed in the kernel. This is 2 147 483 647 by default in CentOS 7, which is enough for a full BGP table, which contains approx. 538 000 prefixes at the time of writing [29]. The /proc/sys/net/ipv6/route.max_size sets the maximum number of IPv6 routes allowed in the kernel. This is 4096 by default in CentOS 7, which must be raised for the measurements involving IPv6 BGP routes. The number of IPv6 prefixes announced in the Internet is approx. 22 000 at the time of writing [29].

The source IPv4 address validation is enabled by default in CentOS 7. This feature is called Reverse path filtering (*rp_filter*) in the Linux kernel and it prevents IP spoofing. However, it introduces additional processing and thus it should be disabled during the experiments. The *rp_filter* can be disabled on a particular interface by writing „0“ to /proc/sys/net/ipv4/conf/*ifname*/rp_filter [23]. The *rp_filter* for IPv6 is implemented in the netfilter subsystem of the Linux kernel and thus it can be configured by ip6tables [22].

Files in core directory provide access to low-level variables of the networking code. There are two parameters that influence NAPI processing. The /proc/sys/net/core/dev_weight file sets the maximum number of packets that a single device can feed to the kernel in its *poll()* function. The default value is 64 in CentOS 7. This value can be increased to allow the device to feed more packets at once. However, most of the drivers provide their own limit which cannot be overwritten unless the code of the driver is changed. This is the case of the mlx4 driver as well, which imposes the limit to 64 packets [22]. The /proc/sys/net/core/netdev_budget file sets the maximum number of packets taken from all interfaces by a single *net_rx_action()* run. The interfaces which are registered to polling are probed in a round-robin manner, as described in subsection 3.4.1. To allow the kernel to spend more time on packet processing, the *netdev_budget* value can be increased.

The proc filesystem further provides access to the IRQ settings and statistics. The /proc/interrupts file exports a table of all registered interrupts and their respective counters for each CPU. Each registered interrupt has its own IRQ number. On a multiprocessor system the interrupt can be served by any of the present CPU if the physical bus supports it. This is the case of the PCI-Express MSI-X feature, which allows to deliver an interrupt to a specified CPU, as described in section 2.2. The /proc/interrupts file even allows to assign IRQ to a CPU which is not directly connected to the PCI-Express, however this would lead to additional overhead due to inter-processor interrupts and QPI communication. Routing performance of such configuration can be measured.

Section 3.6 described how a network adapter with multiple queues allows to scale the network traffic processing on a multi-processor system. Each queue has assigned its own separate IRQ, thus it can be served by a particular CPU if configured properly. The targeted CPU is defined using a mask written to the /proc/irq/*NUMBER*/smp_affinity file, or using a list of CPUs written to the /proc/irq/*NUMBER*/smp_affinity_list file. The /proc/irq/default_smp_affinity file specifies a default mask of CPUs for newly registered interrupts [37].

The Linux kernel does not set the IRQ mapping automatically. Instead, the user must configure it manually. The *irqbalance* daemon reads the content of the /proc/interrupts file and assigns the IRQ mappings according to the load. The *irqbalance* daemon is part of the CentOS 7 and it is enabled by default. While the *irqbalance* can introduce some performance advantage, it does not take scaling or CPU placements (NUMA) into account [25].

The CentOS 7 operating system further features the *tuned* utility in its default installation. The *tuned* allows user to switch between user definable tuning profiles. Several predefined profiles are already included, such as *network-throughput* or *network-latency*. However, none of the profiles influences the low-level packet processing parameters described in chapter 3. Instead, the *tuned* focuses on L4 protocol parameters such as socket memory options [21].

4.4.3 Sysfs settings

Apart from the *proc* filesystem, the Linux kernel provides another virtual filesystem found under /sys in CentOS 7. The *sys* filesystem exports information about loaded modules, including the parameters if a module takes any. The /sys/modules/mxl4_core/parameters directory contains parameters used by the mlx4_core module. The *msi_x* parameter is set to 1 by default, which means attempt to use MSI-X. The /sys/modules/mxl4_en/parameters directory contains parameters used by the mlx4_en module. The *udp_rss* parameter is set to 1 by default, which enables RSS for incoming UDP traffic. There are more parameters taken by both modules, but they are not discussed in this thesis. Further description of the parameters can be found in the *drivers/net/ethernet/mellanox/mlx4* directory of the Linux kernel source code [22].

Each network interface is represented by a symlink in the /sys/class/net/ directory. The symlinks point to the corresponding network device, which is represented as a directory in *sysfs*. The scaling mechanisms described in section 3.6 can be set using the files exported in /sys/class/net/*ifname*/queues. Each rx-*xx* subdirectory represents a single hardware receive queue. The rx-*xx*/rps_cpus file can be used to set a mask of CPUs serving interrupt requests from a particular hardware queue. Since the Mellanox ConnectX-3 NIC supports RSS, the RPS feature is disabled by default - each rps_cpus is set to 0. Similary, the XPS feature can be configured via tx-*xx*/xps_cpus. The XPS configuration should be always checked to reflect the IRQ affinity mappings configured via proc.

4.4.4 Ethtool settings

Ethtool is a standard Linux utility for manipulating network drivers and hardware, particularly for wired Ethernet devices. Supported offload features can be displayed using ethtool --show-offload *ifname*. Listing 4.1 shows the output of ethtool --show-offload eth0, where eth0 is the Mellanox ConnectX-3 adapter.

```
rx-checksumming: on
tx-checksumming: on
  tx-checksum-ipv4: on
  tx-checksum-ip-generic: off [fixed]
  tx-checksum-ipv6: on
  tx-checksum-fcoe-crc: off [fixed]
  tx-checksum-sctp: off [fixed]
scatter-gather: on
  tx-scatter-gather: on
  tx-scatter-gather-fraglist: off [fixed]
```

```
tcp-segmentation-offload: on
  tx-tcp-segmentation: on
  tx-tcp-ecn-segmentation: off [fixed]
  tx-tcp6-segmentation: on
udp-fragmentation-offload: off [fixed]
generic-segmentation-offload: on
generic-receive-offload: on
large-receive-offload: off [fixed]
rx-vlan-offload: on [fixed]
tx-vlan-offload: on [fixed]
ntuple-filters: off [fixed]
receive-hashing: on
highdma: on [fixed]
rx-vlan-filter: on [fixed]
vlan-challenged: off [fixed]
tx-lockless: off [fixed]
netns-local: off [fixed]
tx-gso-robust: off [fixed]
tx-fcoe-segmentation: off [fixed]
tx-gre-segmentation: off [fixed]
tx-ipip-segmentation: off [fixed]
tx-sit-segmentation: off [fixed]
tx-udp_tnl-segmentation: off [fixed]
tx-mpls-segmentation: off [fixed]
fcoe-mtu: off [fixed]
tx-nocache-copy: on
loopback: off
rx-fcs: off [fixed]
rx-all: off [fixed]
tx-vlan-stag-hw-insert: off [fixed]
rx-vlan-stag-hw-parse: off [fixed]
rx-vlan-stag-filter: off [fixed]
```

Listing 4.1: Output of ethtool --show-offload for Mellanox ConnectX3 adapter

Selected offload features can be enabled by ethtool --offload *devname feature* on, or disabled by ethtool --offload *devname feature* off. The features listed as [fixed] cannot be changed on this particular NIC. The rest of the features can be changed, but the feature name differs when executing ethtool --offload. For example, the rx-checksumming feature is turned on by ethtool --offload eth0 *rx* on. Similarly, the scatter-gather feature is turned on by ethtool --offload eth0 *sg* on. For more information about using ethtool, the man page of ethool(8) should be consulted.

Listing 4.1 shows that all supported offload features not marked as [fixed] are enabled, except the *loopback* option. However, the option is not used when processing network traffic between two hosts [22]. This is the default configuration in the CentOS 7 operating system with the Mellanox ConnectX-3 EN adapter.

Chapter 5

Setup

The basic setup consists of plugging the Mellanox ConnectX-3 NIC to the PCI-Express slot of the server's motherboard, cabling the server with the Spirent packet generator, installing the CentOS 7 GNU/Linux distribution, assigning IP adresseses on all interconnected interfaces and allowing IP forwarding. The CentOS 7 routing performance with the default configuration can be measured afterwards.

Further setup includes setting the *performance* scaling governor for better CPU utilisation, disabling SELinux, disabling Netfilter and Reverse path filter, disabling the *irqbalance* daemon and assigning IRQ affinity manually. These steps are focused on maximum performance in frames per second statistic and on maximum CPU utilisation.

5.1 Hardware and networking

Figure 5.1 shows the block diagram of the Supermicro motherboard. The Intel Xeon E5-2660 v3 processors were plugged into the CPU sockets. The Mellanox ConnectX-3 EN adapter was plugged into the PCIE 3.0 x8 Upper slot, which is part of the *WIO* block. The PCI-Express links are directly connected to the CPU 1 only. Communication with the other CPU is performed over the QPI links. Thus, an interrupt request targeted to CPU 0 involves a necessary execution on CPU 1 as well.

The server was put to the same rack as the Spirent hardware generator. A pair of 2 metres long 40GBASE-SR4 multimode fiber cables with QSPF connectors was used to connect Spirent with the Mellanox ConnectX-3 EN adapter.

IPv4 addresses from 192.0.2.0/24 (TEST-NET-1) block were assigned. IPv6 addresses from 2001:db8::/32 range were assigned. Addresses within these blocks should not appear on the public Internet and therefore they do not conflict with the prefixes from the Internet BGP table [30] [3]. Figure 5.2 shows the addressing scheme used for the measurements.

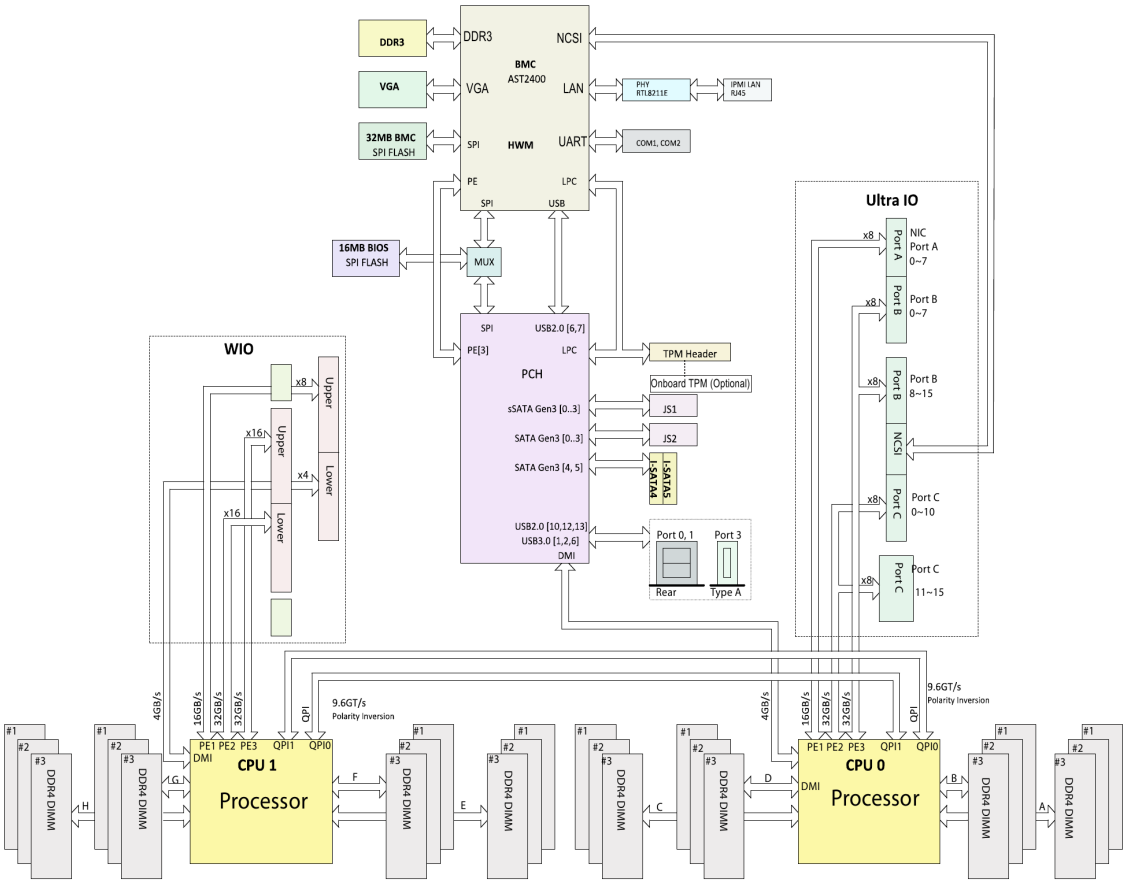


Figure 5.1: Supermicro motherboard's block diagram

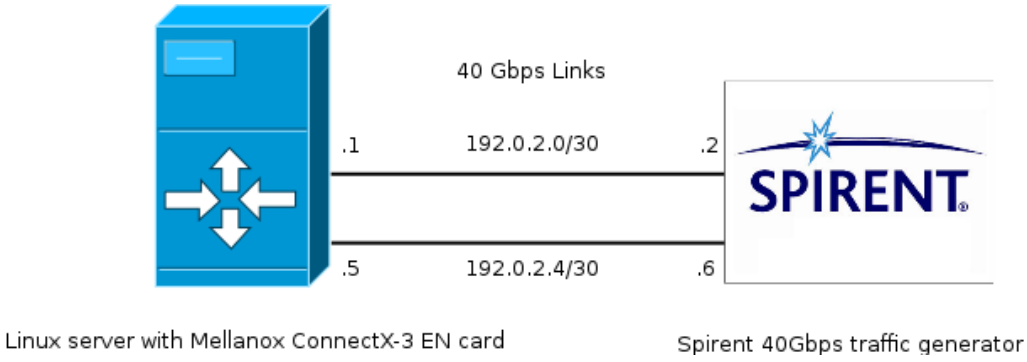


Figure 5.2: Addressing scheme

5.2 Software and firmware

Base CentOS 7 was installed on the server. The operating system features Linux kernel based on version 3.10 - the installed version is 3.10.0-123.20.1.el7.x86_64. The operating system was updated with all updates available as of 1st May 2015. The upstream kernel version 4.0.2 was additionally installed from the ELRepo repository [40].

The Linux kernel detects the Mellanox ConnectX-3 EN card automatically and loads the mlx4_core and mlx4_en module. The mlx4_core module prints the detected PCI-Express link parameters to the kernel's message buffer. The buffer can be viewed using the *dmesg* utility and its partial output is shown bellow:

```
mlx4_core 0000:06:00.0: PCIe link speed is 8.0GT/s, device supports 8.0GT/s
mlx4_core 0000:06:00.0: PCIe link width is x8, device supports x8
```

The mlx4_core module further registers interrupts and prints the assigned IRQ numbers for each queue to the kernel's message buffer:

```
mlx4_core 0000:06:00.0: irq 61 for MSI/MSI-X
mlx4_core 0000:06:00.0: irq 62 for MSI/MSI-X
...
mlx4_core 0000:06:00.0: irq 90 for MSI/MSI-X
```

The driver uses either MSI or MSI-X feature of the PCI-Express bus, as described in section 2.2. The MSI-X feature is used automatically if the system supports it, otherwise the adapter uses MSI. The *lspci -vv* command can be used to check whether MSI-X is used. Listing 5.1 shows a partial output of lspci for the Mellanox ConnectX-3 EN adapter. The MSI-X capability is followed by an Enable flag which is followed with either „+“ (enabled) or „-“ (disabled). Listing 5.1 shows that the system supports MSI-X and the adapter is configured to use it.

```
06:00.0 Ethernet controller: Mellanox Technologies MT27500 Family [ConnectX
    -3]
    ...
    Capabilities: [9c] MSI-X: Enable+ Count=128 Masked-
        ...
        LnkCap: Port #8, Speed 8GT/s, Width x8, ASPM L0s, Exit Latency L0s
            unlimited, L1 unlimited
        ...
```

Listing 5.1: Partial output of lspci -vv for Mellanox ConnectX-3 EN

Random Length	IP Total Length	Default Ethernet	Weight	Percentage
☐	1500	1518	214	21,4%
☑	1006 - 1499	1024 - 1517	263	26,3%
☑	494 - 1005	512 - 1023	28	2,8%
☑	238 - 493	256 - 511	30	3%
☑	110 - 237	128 - 255	41	4,1%
☑	48 - 109	66 - 127	424	42,4%

Figure 5.3: AMS-IX iMix

Apart from the NIC driver, the Mellanox ConnectX-3 adapter uses its own proprietary firmware. The firmware was updated to version 2.32.5100, which is the latest version available as of 10th January 2015. The firmware is not a part of the Linux kernel and its update procedure is described in appendix B.

5.3 Spirent configuration

Custom iMix, named AMS-IX, was configured according to the AMS-IX statistics described in subsection 4.3.1. Figure 5.3 shows the configuration window for custom iMix definition from the Spirent TestCenter Application. One device was configured on each interface and it was assigned IPv4 and IPv6 addresses, as described in section 5.1. The *Respond to ping* option was enabled to test the connection.

Two traffic patterns were configured for each IP version - one generates a single L4 flow and the other generates 32 L4 flows. The single flow traffic pattern uses IP addresses according the scheme described in section 5.1 and UDP source and destination port 1024. Each flow within the traffic pattern with 32 L4 flows uses different UDP source and destination ports within range from 1024 up to 1055. The source and destination ports are the same for each flow. The scheme is used for both IPv4 and IPv6 traffic patterns. The purpose of the scheme is to use Receive Side Scaling to distribute the traffic to all CPUs uniformly.

To test the routing performance of the Linux kernel with full BGP table, another traffic pattern was configured. The pattern uses randomly generated IP destination address for each packet to avoid having the previous lookup result cached. All traffic patterns can be configured to generate fixed frame size or custom AMS-IX iMix.

5.4 Server configuration

The general system configuration used by default in all measurements consists of disabling Netfilter, disabling SELinux, changing the scaling governor, disabling rp_filter and configuring the IP addresses according to the scheme described in section 5.1. Additionally, the system should run no unused services - CentOS 7 runs Postfix, Avahi daemon, Polkitd, Tuned, Crond, NetworkManger, dbus, crond, rsyslogd and auditd by default. It is recommended to disable all of these services for the performance reasons. Note that dbus may still be activated if required by any of the user-space application the system runs. Beware that disabling NetworkManager may cause no network connectivity. If the system uses DHCP to obtain an IP address, the *ifcfg* network scripts or the */etc/rc.local* script can be

used for this purpose. Execution of *dhclient* can be added to the */etc/rc.local* file and the file must be made executable by running *chmod +x /etc/rc.local*. The *dhclient* daemon should be killed after it obtained an IP address to avoid sending DHCP Request packets on other interfaces. The services can be disable using *systemctl* command, as the following listing shows.

```
systemctl disable postfix avahi-daemon tuned crond polkit NetworkManager
    auditd dbus cron rsyslog
reboot
```

The *ps uax* command can be further used to list running processes on the system.

The measurements were executed under the following conditions:
Netfilter was disabled completely:

```
systemctl stop firewalld
systemctl disable firewalld  # do not execute firewalld on boot
```

SELinux was disabled by changing the SELINUX variable to *disabled* in /etc/sysconfig/selinux (default is *Enforcing*). The system must reboot to apply:

```
vi /etc/sysconfig/selinux
  SELINUX=disabled
reboot
```

The scaling governor was changed for each CPU to *performance* (default is *powersave*):

```
echo performance | tee /sys/devices/system/cpu/cpu[0-9]*/cpufreq/
    scaling_governor
```

The rp_filter was disabled on all interfaces (it should be disabled at least on the forwarding interfaces):

```
echo 0 | tee /proc/sys/net/ipv4/conf/*/rp_filter
```

It is suggested to disable the rp_filter at boot time by putting the following lines to the */etc/sysctl.conf* file:

```
net.ipv4.conf.all.rp_filter=0
net.ipv4.conf.default.rp_filter=0
net.ipv4.conf.lo.rp_filter=0
net.ipv4.conf.enp129s0.rp_filter=0        # forwarding interface 1
net.ipv4.conf.enp129s0d1.rp_filter=0      # forwarding interface 2
```

IPv6 was enabled on all interfaces (it must be enabled at least on the forwarding interfaces):

```
echo 0 > /proc/sys/net/ipv6/conf/all/disable_ipv6
```

IPv4 forwarding was enabled:

```
echo 1 > /proc/sys/net/ipv4/ip_forward
```

IPv6 forwarding was enabled on all interfaces (it must be enabled at least on the forwarding interfaces):

```
echo 1 > /proc/sys/net/ipv6/conf/all/forwarding
```

IPv4 neighbours were set:

```
ip neigh add 192.0.2.2 lladdr 00:10:94:00:00:01 dev enp6s0d1
ip neigh add 192.0.2.6 lladdr 00:10:94:00:00:02 dev enp6s0
```

IPv6 neighbours were set:

```
ip -6 neigh add 2001:db8:1::2 lladdr 00:10:94:00:00:03 dev enp6s0d1
ip -6 neigh add 2001:db8:2::6 lladdr 00:10:94:00:00:04 dev enp6s0
```

IPv4 addresses were assigned:

```
ip addr add 192.0.2.1/30 broadcast 192.0.2.3 dev enp6s0d1
ip addr add 192.0.2.5/30 broadcast 192.0.2.7 dev enp6s0
```

IPv6 addresses were assigned:

```
ip -6 addr add 2001:db8:1::1/64 dev enp6s0d1
ip -6 addr add 2001:db8:2::5/64 dev enp6s0
```

The routing performance of the upstream Linux kernel version 4.0.2 was further measured. The ELRepo was added to available repositories and the kernel was installed. The instructions to add the ELRepo repository are provided by the elrepo.org site.[1] Afterwards, the *kernel-ml* package was installed:

```
yum --enablerepo=elrepo-kernel install kernel-ml
```

The kernel can be set as default in the bootloader configuration. The following command prints all available kernels on the system:

```
grep "submenu\|^\menuentry" /boot/grub2/grub.cfg | cut -d "'" -f2
  CentOS Linux, with Linux 4.0.2-1.el7.elrepo.x86_64
  CentOS Linux, with Linux 3.10.0-123.13.1.el7.x86_64
  CentOS Linux, with Linux 0-rescue-f8351e2baaac42a285a6443a1f777333
```

The 4.0.2 kernel can be set as default by changing the configuration of grub2:

```
grub2-set-default 'CentOS Linux, with Linux 4.0.2-1.el7.elrepo.x86_64'
```

The routes announced in public BGP were imported to the kernel's FIB to perform additional measurements. Appendix A describes the step-by-step instructions on how to obtain the BGP table and import it to the FIB. Any additional system settings for a particular measurement are described next to the result.

[1] http://elrepo.org/

Chapter 6

Measurements

The results presented in this thesis are based on the setup and basic settings described in chapter 5. The CentOS 7 operating system was installed on the server with two Intel Xeon E5-2660 v3 CPUs and Mellanox ConnectX-3 EN 40 Gbps Ethernet adapter, as described in section 4.1. Each measurement put the server under 60 seconds of constant unidirectional traffic load, as described in section 4.3. The bandwidth use is configured in frames per second with a unit of margin 50 000. If the bandwidth use reaches 39.40 Gbps (98.50%) then the link is considered as saturated. The measurements with 32 flows use different UDP source and destination ports for each flow, as described in section 5.3.

6.1 CentOS 7 distribution kernel 3.10.0-123

The CentOS 7 distribution kernel version 3.10.0-123.20.1.el7.x86_64 was used in the measurements presented in this section.

6.1.1 Measurement 1 - default configuration - single IPv4 flow

The first measurement shows the routing performance with IP addresses assigned, IP forwarding enabled and Netfilter rules flushed.

The IP addresses were assigned as described in section 5.4. The IP forwarding was enabled by echoing „1“ to the /proc/sys/net/ipv4/ip_forward file and the Netfilter rules were flushed using *iptables -F*, because the default rules do not allow forwarding.

Frame size	% of link	bandwidth	frame rate
64	0.59%	0.24 Gb/s	350 000
594	4.30%	1.72 Gb/s	350 000
1518	10.77%	4.31 Gb/s	350 000
AMS-IX	5.33%	2.13 Gb/s	350 000

6.1.2 Measurement 2 - default configuration - 32 IPv4 flows

Since the Linux kernel scaling mechanisms described in section 3.6 are based on processing each flow by a different CPU, the routing performance of the default configuration was tested against 32 IPv4 flows.

Frame size	% of link	bandwidth	frame rate
64	2.69%	1.08 Gb/s	1 550 000
594	19.65%	7.86 Gb/s	1 600 000
1518	49.22%	19.69 Gb/s	1 650 000
AMS-IX	24.34%	9.74 Gb/s	1 600 000

As expected, the scaling mechanisms help to increase the routing performance of the Linux kernel. The scaling mechanisms perform better when forwarding larger frames. This may be caused by differences in memory management allocations, since the memory management is common for all CPUs present in the system.

6.1.3 Measurement 3 - single IPv4 flow

Unlike the previous measurements, the following measurements use the complete setup described in section 5.4 to increase the throughput.

Frame size	% of link	bandwidth	frame rate
64	1.68%	0.67 Gb/s	1 000 000
594	12.28%	4.91 Gb/s	1 000 000
1518	30.76%	12.30 Gb/s	1 000 000
AMS-IX	15.22%	6.09 Gb/s	1 000 000

The Linux kernel is able to forward 1 million IPv4 packets per second on a single core. The following partial output of the /proc/interrupts file shows the interrupt mapping:

```
        ...  CPU8   CPU9   CPU10  CPU11  CPU12   ...
178:    ...     0      0  255725      0      0   ...  enp129s0-0
179:    ...     0      0       0      0      0   ...  enp129s0-1
180:    ...     0      0       0      0      0   ...  enp129s0-2
181:    ...     0      0       0      0      0   ...  enp129s0-3
182:    ...     0      0       0      0      0   ...  enp129s0-4
183:    ...     0      0       0      0      0   ...  enp129s0-5
184:    ...     0      0       0      0      0   ...  enp129s0-6
185:    ...     0      0       0      0      0   ...  enp129s0-7
186:    ...     0      0       0      0      0   ...  enp129s0d1-0
187:    ...     0      0       0      0      0   ...  enp129s0d1-1
188:    ...     0      0       0      0      0   ...  enp129s0d1-2
189:    ...     0      0       0      0      0   ...  enp129s0d1-3
190:    ...     0      0       0      0      0   ...  enp129s0d1-4
191:    ...     0      0  313670      0      0   ...  enp129s0d1-5
192:    ...     0      0       0      0      0   ...  enp129s0d1-6
193:    ...     0      0       0      0      0   ...  enp129s0d1-7
```

All packets are assigned to a single queue, in this case the queue number 5 (*enp129s0d1* is the receiving interface). The smp_affinity files show that the Linux kernel automatically assigned each interrupt to the CPUs 10-19 and 30-39.

```
cat /proc/irq/178/smp_affinity_list
  10-19,30-39
cat /proc/irq/179/smp_affinity_list
  10-19,30-39
...
cat /proc/irq/193/smp_affinity_list
  10-19,30-39
```

The *lscpu* utility or the Intel Performance Counter Monitor can be used to verify that the CPUs 10-19 and 30-39 are logical cores of the CPU in Socket 1. The CPU in this socket is directly connected to the PCI-Express link, as shown in figure 5.1.

The *perf* utility can be used to list the functions utilising the CPU 10.

```
perf top -C 10
  11.42%  [kernel]  [k] fib_table_lookup
   9.62%  [kernel]  [k] _raw_spin_lock
   6.65%  [kernel]  [k] mlx4_en_xmit
   4.84%  [kernel]  [k] memcpy
   4.03%  [kernel]  [k] mlx4_en_complete_rx_desc
   3.61%  [kernel]  [k] check_leaf.isra.7
   3.52%  [kernel]  [k] mlx4_en_free_tx_desc.isra.22
   3.42%  [kernel]  [k] mlx4_en_process_rx_cq
   3.16%  [kernel]  [k] mlx4_en_poll_tx_cq
   2.89%  [kernel]  [k] put_compound_page
   2.72%  [kernel]  [k] dev_queue_xmit
```

The kernel spends most of the time on FIB lookup, which is performed by the *fib_table_lookup* function described in section 3.3, and on locking.

The following listing shows the cache utilisation on all CPUs present in the system.

```
L3MISS: L3 cache misses
L2MISS: L2 cache misses (including other core's L2 cache *hits*)
L3HIT : L3 cache hit ratio (0.00-1.00)
L2HIT : L2 cache hit ratio (0.00-1.00)
L3CLK : ratio of CPU cycles lost due to L3 cache misses (0.00-1.00), in some
     cases could be >1.0 due to a higher memory latency
L2CLK : ratio of CPU cycles lost due to missing L2 cache but still hitting
    L3 cache (0.00-1.00)
L3OCC : L3 occupancy (in KBytes)

Core (SKT) | L3MISS | L2MISS | L3HIT | L2HIT | L3CLK | L2CLK |  L3OCC

  0    0     4052       23 K   0.83    0.26    0.19    0.18      120
  1    0       52       1542   0.97    0.14    0.03    0.17       80
  2    0       37       1458   0.97    0.14    0.02    0.17      200
  3    0       42       1489   0.97    0.14    0.02    0.13        0
  4    0       36       1460   0.98    0.15    0.02    0.16        0
  5    0      223       1695   0.87    0.22    0.00    0.00       80
  6    0        4        542   0.99    0.13    0.00    0.12       40
  7    0        4        539   0.99    0.12    0.00    0.12        0
  8    0        4        533   0.99    0.12    0.00    0.11        0
  9    0       20        589   0.97    0.10    0.02    0.11       40
 10    1     1179       2608 K  1.00    0.88    0.00    0.04      240
 11    1      129        586   0.78    0.11    0.16    0.13        0
 12    1        7        531   0.99    0.12    0.01    0.16        0
 13    1      446       2223   0.80    0.14    0.06    0.05       80
 14    1        7        531   0.99    0.14    0.01    0.12        0
 15    1        7        535   0.99    0.13    0.01    0.13        0
 16    1        6        534   0.99    0.13    0.01    0.12        0
 17    1       16        675   0.98    0.17    0.01    0.12        0
 18    1       10        534   0.98    0.13    0.01    0.12        0
 19    1       24        607   0.96    0.11    0.02    0.10        0
 20    0      175       1090   0.84    0.07    0.08    0.14        0
 21    0        9        598   0.98    0.13    0.01    0.18        0
```

```
 22     0        80       974        0.92    0.14    0.01    0.03        0
 23     0        10       583        0.98    0.13    0.01    0.10        0
 24     0        15       831        0.98    0.12    0.01    0.17        0
 25     0        41       582        0.93    0.13    0.03    0.11        0
 26     0         4       513        0.99    0.13    0.00    0.13        0
 27     0         5       534        0.99    0.13    0.01    0.14       40
 28     0         5       534        0.99    0.12    0.01    0.12       40
 29     0        34       659        0.95    0.12    0.03    0.12        0
 30     1       102       603        0.83    0.12    0.16    0.17       40
 31     1        12       627        0.98    0.34    0.01    0.15      120
 32     1        37       544        0.93    0.12    0.05    0.16        0
 33     1        11       584        0.98    0.13    0.01    0.16       40
 34     1         7       528        0.99    0.14    0.01    0.17       40
 35     1         8       541        0.99    0.16    0.01    0.17        0
 36     1         8       541        0.99    0.12    0.01    0.16        0
 37     1        11       541        0.98    0.13    0.02    0.17       40
 38     1        11       553        0.98    0.12    0.01    0.16        0
 39     1      2105      4296        0.51    0.65    0.13    0.03      480
---------------------------------------------------------------------------
SKT     0      4852       40 K       0.88    0.21    0.03    0.05      640
SKT     1      4143      2624 K      1.00    0.87    0.00    0.04     1080
---------------------------------------------------------------------------
TOTAL   *      8995      2664 K      1.00    0.87    0.00    0.04      N/A
```

The CPU 10 is performing the work related to TCP/IP processing. Most of the cache misses are L2 miss, but there are also L3 misses. The number of L3 misses is greatly reduced by the Intel Data Direct I/O technology, as described in section 4.1. The L2 hit ratio is 88%, while L3 hit ration is almost 100%. Other CPUs are in idle state, except for CPU 0, which is running the Intel PCM and prints the output.

6.1.4 Measurement 4 - 32 independent IPv4 flows

This measurement can be compared with Measurement 2, except that the *irqbalance* daemon is disabled. Therefore, the IRQ mapping is left untouched in its default state.

Frame size	% of link	bandwidth	frame rate
64	1.26%	0.50 Gb/s	750 000
594	12.28%	4.91 Gb/s	800 000
1518	24.61%	9.84 Gb/s	800 000
AMS-IX	15.22%	6.09 Gb/s	800 000

When forwarding IP packets from multiple IPv4 flows on a single CPU, the routing performance of the Linux kernel drops by 20% against forwarding a single IPv4 flow. The following listing shows that the packets are uniformly distributed among all hardware queues. However, the interrupts are not distributed among CPUs.

```
        ...  CPU8   CPU9    CPU10  CPU11  CPU12  ...
178:    ...     0      0   474701      0      0  ...  enp129s0-0
179:    ...     0      0        0      0      0  ...  enp129s0-1
180:    ...     0      0        0      0      0  ...  enp129s0-2
181:    ...     0      0        0      0      0  ...  enp129s0-3
182:    ...     0      0        0      0      0  ...  enp129s0-4
183:    ...     0      0        0      0      0  ...  enp129s0-5
184:    ...     0      0        0      0      0  ...  enp129s0-6
185:    ...     0      0        0      0      0  ...  enp129s0-7
186:    ...     0      0   317322      0      0  ...  enp129s0d1-0
```

```
187:   ...     0     0   317648     0     0   ...   enp129s0d1-1
188:   ...     0     0   317231     0     0   ...   enp129s0d1-2
189:   ...     0     0   317384     0     0   ...   enp129s0d1-3
190:   ...     0     0   317114     0     0   ...   enp129s0d1-4
191:   ...     0     0   317291     0     0   ...   enp129s0d1-5
192:   ...     0     0   317190     0     0   ...   enp129s0d1-6
193:   ...     0     0   317964     0     0   ...   enp129s0d1-7
```

Each receive queue triggers roughly the same number of interrupts as in the previous measurement, but overall the NIC triggers much more interrupts. The kernel spends more time on running the interrupt service routine code. Apart from servicing interrupts, the kernel must fetch the packets from different ingress queues, which in turn may need additional locking.
The following listing shows partial output of *perf*:

```
perf top -C 10
  12.07%  [kernel]  [k] _raw_spin_lock
   8.68%  [kernel]  [k] fib_table_lookup
   5.01%  [kernel]  [k] mlx4_en_xmit
   4.63%  [kernel]  [k] mlx4_en_process_rx_cq
   3.64%  [kernel]  [k] __netif_receive_skb_core
   3.49%  [kernel]  [k] memcpy
   3.08%  [kernel]  [k] irq_entries_start
   2.68%  [kernel]  [k] mlx4_eq_int
   2.33%  [kernel]  [k] mlx4_en_poll_tx_cq
   2.24%  [kernel]  [k] ip_route_input_noref
```

The kernel spends most of the time on locking and FIB table lookup.

6.1.5 Measurement 5 - single IPv4 flow over QPI

In this measurement, the kernel is instructed to use the CPU 9 for processing the RX interrupts. The logical CPU 9 resides in Socket 0, which is not directly connected to the PCI-Express link. The softirq and forwarding code runs on CPU 9. To achieve this, the QPI links between CPUs must be used, as described in section 5.1. The following command maps the interrupts to CPU 9.

```
for i in `seq 178 193` ; do echo 9 > /proc/irq/$i/smp_affinity_list ; done
```

The results are presented by the table bellow.

Frame size	% of link	bandwidth	frame rate
64	1.09%	0.44 Gb/s	650 000
594	7.98%	3.19 Gb/s	650 000
1518	19.99%	8.00 Gb/s	650 000
AMS-IX	9.89%	3.96 Gb/s	650 000

The routing performance drops by 35% when it is performed by a CPU not directly connected to the PCI-Express link with the NIC. This is a significant performance drop.
Moreover, the following listing shows that the CPU 10 must be also involved, as it communicates with the CPU 9 over the QPI.

```
L3MISS: L3 cache misses
L2MISS: L2 cache misses (including other core's L2 cache *hits*)
L3HIT : L3 cache hit ratio (0.00-1.00)
L2HIT : L2 cache hit ratio (0.00-1.00)
L3CLK : ratio of CPU cycles lost due to L3 cache misses (0.00-1.00), in some
     cases could be >1.0 due to a higher memory latency
L2CLK : ratio of CPU cycles lost due to missing L2 cache but still hitting
    L3 cache (0.00-1.00)
L3OCC : L3 occupancy (in KBytes)
```

Core	(SKT)	L3MISS	L2MISS	L3HIT	L2HIT	L3CLK	L2CLK	L3OCC
0	0	6209	24 K	0.75	0.32	0.19	0.12	480
1	0	13	546	0.98	0.12	0.01	0.14	640
2	0	154	4746	0.97	0.16	0.03	0.18	40
3	0	16	1452	0.99	0.14	0.01	0.15	0
4	0	68	1164	0.94	0.25	0.00	0.00	80
5	0	10	537	0.98	0.13	0.01	0.14	160
6	0	5	538	0.99	0.13	0.01	0.14	0
7	0	10	543	0.98	0.12	0.01	0.13	0
8	0	10	548	0.98	0.11	0.01	0.11	0
9	0	2002 K	2798 K	0.28	0.74	0.20	0.02	2360
10	1	790	12 K	0.94	0.14	0.08	0.24	4640
11	1	29	985	0.97	0.12	0.02	0.15	0
12	1	22	569	0.96	0.11	0.02	0.09	0
13	1	325	2112	0.85	0.15	0.05	0.06	40
14	1	238	1328	0.82	0.16	0.05	0.05	120
15	1	28	526	0.95	0.16	0.02	0.07	40
16	1	32	572	0.94	0.13	0.02	0.07	0
17	1	27	563	0.95	0.11	0.02	0.08	0
18	1	20	563	0.96	0.12	0.01	0.09	40
19	1	30	606	0.95	0.11	0.02	0.09	40
20	0	158	1017	0.84	0.07	0.08	0.13	40
21	0	10	553	0.98	0.13	0.02	0.20	0
22	0	10	739	0.99	0.10	0.01	0.16	40
23	0	7	574	0.99	0.13	0.01	0.18	0
24	0	10	578	0.98	0.14	0.01	0.16	0
25	0	4	535	0.99	0.14	0.01	0.16	0
26	0	11	551	0.98	0.12	0.01	0.16	0
27	0	11	550	0.98	0.13	0.01	0.13	0
28	0	10	560	0.98	0.13	0.01	0.14	0
29	0	97	5729	0.98	0.07	0.00	0.05	0
30	1	185	911	0.80	0.08	0.10	0.12	0
31	1	19	572	0.97	0.11	0.02	0.13	0
32	1	15	556	0.97	0.13	0.02	0.13	80
33	1	23	591	0.96	0.13	0.02	0.12	40
34	1	21	555	0.96	0.12	0.01	0.09	0
35	1	21	539	0.96	0.15	0.02	0.09	120
36	1	23	555	0.96	0.13	0.02	0.09	0
37	1	63	684	0.91	0.18	0.01	0.02	0
38	1	16	555	0.97	0.14	0.01	0.10	0
39	1	4826	7108	0.32	0.52	0.24	0.02	0
SKT	0	2009 K	2844 K	0.29	0.73	0.20	0.02	3840
SKT	1	6753	32 K	0.79	0.26	0.10	0.08	5160
TOTAL	*	2015 K	2877 K	0.30	0.73	0.19	0.02	N/A

CPU 9 performs the actual forwarding, while CPU 10 is busy with the QPI communication overhead. As a result of this, performing the actual forwarding by a CPU connected over the QPI decreases the routing performance significantly.

The use of QPI links is shown by the listing bellow.

```
Intel(r) QPI data traffic estimation in bytes (data traffic coming to CPU/
    socket through QPI links):

                QPI0      QPI1
--------------------------------------------------------------------------------
 SKT    0        90 M      90 M
 SKT    1        70 M      71 M
--------------------------------------------------------------------------------
Total QPI incoming data traffic:  323 M
QPI data traffic/Memory controller traffic: 0.39
```

The following listings shows the output of the *perf* utility for CPU 9 and CPU 10.

```
perf top -C 9
  21.25%  [kernel]  [k] _raw_spin_lock
  11.61%  [kernel]  [k] memcpy
   6.69%  [kernel]  [k] fib_table_lookup
   6.60%  [kernel]  [k] skb_gro_reset_offset
   4.55%  [kernel]  [k] udp_gro_receive
   3.96%  [kernel]  [k] mlx4_en_xmit
   3.44%  [kernel]  [k] mlx4_en_process_rx_cq
   2.93%  [kernel]  [k] mlx4_en_poll_tx_cq
```

```
perf top -C 10
   9.77%  [kernel]  [k] find_busiest_group
   3.47%  [kernel]  [k] cpumask_next_and
   3.01%  [kernel]  [k] _raw_spin_lock
   2.93%  [kernel]  [k] ktime_get
   2.59%  [kernel]  [k] mlx4_en_DUMP_ETH_STATS
   2.58%  [kernel]  [k] run_timer_softirq
   2.36%  [kernel]  [k] idle_cpu
   2.22%  [kernel]  [k] __schedule
```

Apart from locking and the actual FIB lookup, the CPU 9 is also busy with *memcpy* which may be caused by the overhead of the QPI communication.

6.1.6 Measurement 6 - 32 IPv4 flows with irqbalance daemon

The measurement includes the *irqbalance* daemon enabled. The *irqbalance* daemon is responsible for dynamically assigning the interrupts to CPUs using the files found under /proc/irq/*NUMBER*/smp_affinity, as described in subsection 4.4.2.

Frame size	% of link	bandwidth	frame rate
64	8.99%	3.60 Gb/s	5 350 000
594	68.15%	27.26 Gb/s	5 550 000
1518	98.50%	39.40 Gb/s	3 202 210
AMS-IX	88.25%	35.30 Gb/s	5 800 000

The scaling mechanisms of the Linux kernel take advantage of interrupt assignment done by the *irqbalance* daemon. The server is able to route almost 36 Gbps of the simulated

AMS-IX internet traffic. The measurement further confirms that the scaling mechanisms are sensitive to the frame size. The measurement featuring 1518 octet frames was configured to use 98.5% of the link bandwidth.
The following listing shows that the *irqbalance* daemon assigned IRQs to CPUs 11-18, 30 and 33-39. The CPUs 12-19 are serving RX interrupts, while the CPUs 30-39 are serving TX interrupts (*en29d1* represents the receiving interface).

	CPU12	CPU13	CPU14	CPU15	CPU16	CPU17	CPU18	CPU19	CPU30	CPU33	CPU34	CPU35	CPU36	CPU37	CPU38	CPU39	
178:	0	0	0	0	0	0	0	0	0	292448	0	0	0	0	0	0	en29−0
179:	0	0	0	0	0	0	0	0	0	0	292978	0	0	0	0	0	en29−1
180:	0	0	0	0	0	0	0	0	0	0	0	292698	0	0	0	0	en29−2
181:	0	0	0	0	0	0	0	0	0	0	0	0	286435	0	0	0	en29−3
182:	0	0	0	0	0	0	0	0	0	0	0	0	0	282449	0	0	en29−4
183:	0	0	0	0	0	0	0	0	0	0	0	0	0	0	288839	0	en29−5
184:	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	327901	en29−6
185:	0	0	0	0	0	0	0	0	325935	0	0	0	0	0	0	0	en29−7
186:	53145	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	en29d1−0
187:	0	53090	0	0	0	0	0	0	0	0	0	0	0	0	0	0	en29d1−1
188:	0	0	39978	0	0	0	0	0	0	0	0	0	0	0	0	0	en29d1−2
189:	0	0	0	40484	0	0	0	0	0	0	0	0	0	0	0	0	en29d1−3
190:	0	0	0	0	40072	0	0	0	0	0	0	0	0	0	0	0	en29d1−4
191:	0	0	0	0	0	39982	0	0	0	0	0	0	0	0	0	0	en29d1−5
192:	0	0	0	0	0	0	40488	0	0	0	0	0	0	0	0	0	en29d1−6
193:	0	0	0	0	0	0	0	43262	0	0	0	0	0	0	0	0	en29d1−6

The listing bellow shows the cache use.

```
L3MISS: L3 cache misses
L2MISS: L2 cache misses (including other core's L2 cache *hits*)
L3HIT : L3 cache hit ratio (0.00-1.00)
L2HIT : L2 cache hit ratio (0.00-1.00)
L3CLK : ratio of CPU cycles lost due to L3 cache misses (0.00-1.00), in some
     cases could be >1.0 due to a higher memory latency
L2CLK : ratio of CPU cycles lost due to missing L2 cache but still hitting L
    3 cache (0.00-1.00)

 Core (SKT) | L3MISS | L2MISS | L3HIT | L2HIT | L3CLK | L2CLK |  L3OCC

   0    0     1076       11 K    0.91   0.17   0.01   0.01       120
   1    0      574     4139      0.86   0.12   0.10   0.14        40
   2    0      241     1421      0.83   0.18   0.12   0.12         0
   3    0      658       11 K    0.94   0.09   0.07   0.24         0
   4    0       19      580      0.97   0.14   0.02   0.12         0
   5    0       78      363      0.79   0.25   0.03   0.03         0
   6    0       65      710      0.91   0.15   0.04   0.10        40
   7    0       19      544      0.97   0.12   0.02   0.10        40
   8    0       13      648      0.98   0.11   0.01   0.08         0
   9    0       32      582      0.95   0.11   0.02   0.08         0
  10    1     1007     3598      0.72   0.06   0.72   0.37        40
  11    1     4452     3802 K    1.00   0.45   0.00   0.10        40
  12    1     4932     3770 K    1.00   0.42   0.00   0.10       240
  13    1     6273     4131 K    1.00   0.51   0.00   0.09        40
  14    1     5086     4211 K    1.00   0.52   0.00   0.09        80
  15    1     4762     4211 K    1.00   0.50   0.00   0.10        80
  16    1     4453     4111 K    1.00   0.54   0.00   0.09        80
  17    1     4680     4124 K    1.00   0.56   0.00   0.09       160
  18    1     4737     4275 K    1.00   0.48   0.00   0.10        80
  19    1      105      573      0.82   0.18   0.18   0.17         0
  20    0      170      979      0.83   0.06   0.07   0.10        40
  21    0       16      692      0.98   0.10   0.01   0.12         0
  22    0       12      570      0.98   0.12   0.01   0.13         0
  23    0       15      839      0.98   0.10   0.01   0.13         0
  24    0       13      532      0.98   0.16   0.02   0.14         0
  25    0       22      441      0.95   0.21   0.01   0.06         0
  26    0       14      540      0.97   0.14   0.02   0.13         0
  27    0        9      516      0.98   0.15   0.01   0.13         0
  28    0      268     2659      0.90   0.09   0.10   0.18        40
  29    0      239      916      0.74   0.15   0.10   0.06         0
  30    1     1238     2816 K    1.00   0.60   0.00   0.27       680
  31    1       54      560      0.90   0.25   0.09   0.19        80
  32    1     3416     9666      0.65   0.20   0.72   0.27        40
  33    1       14 K   3887 K    1.00   0.64   0.00   0.27       640
  34    1       16 K   3969 K    1.00   0.63   0.00   0.27       960
  35    1       15 K   3881 K    1.00   0.64   0.00   0.27       960
  36    1       14 K   3878 K    1.00   0.63   0.00   0.27       880
  37    1       14 K   3920 K    1.00   0.63   0.00   0.27       440
  38    1       17 K   4014 K    1.00   0.63   0.00   0.27       600
  39    1     2106     2853 K    1.00   0.60   0.00   0.28      1240
-------------------------------------------------------------------------
SKT     0     3553       41 K    0.91   0.13   0.01   0.03       320
SKT     1      141 K     61 M    1.00   0.57   0.00   0.14      7360
-------------------------------------------------------------------------
TOTAL   *      144 K     61 M    1.00   0.57   0.00   0.14       N/A
```

The measurement featuring 1518 octet frames is the first measurement saturating the 40 Gbps Ethernet connection. Intel PCM can be used to monitor the PCI-Express utilisation:

```
Skt  | PCIe Rd (B)  | PCIe Wr (B)
 0          5270 K          86 K
 1            11 G        5422 M
---------------------------------
 *            11 G        5422 M
```

The PCI-Express link could be saturated when forwarding bidirectional traffic - the PCI-Express 3.0 x8 throughput is 7 876.8 MB/s as calculated in section 2.2. Note, that there seems to be a bug in Intel PCM related to displaying the PCIe Read bandwidth - it always shows double the expected value (11 Gigabytes does not make sense).

6.1.7 Measurement 7 - 32 IPv4 flows with manual IRQ affinity mappings

While *irqbalance* mapped the interrupts intelligently, it is always worth checking the mappings. The dynamic mappings made by the *irqbalance* daemon can change during the run-time, which may lead to unpredictable performance drops.
The following listing shows the interrupt mapping scheme used during this measurement. Unlike the mapping assigned by the *irqbalance* daemon, this mapping targets both RX and TX interrupts to 8 CPUs only. Additionally, Transmission Packet Steering (XPS) mechanism was configured to maps each exclusively to a single CPUs, as described in section 3.6.

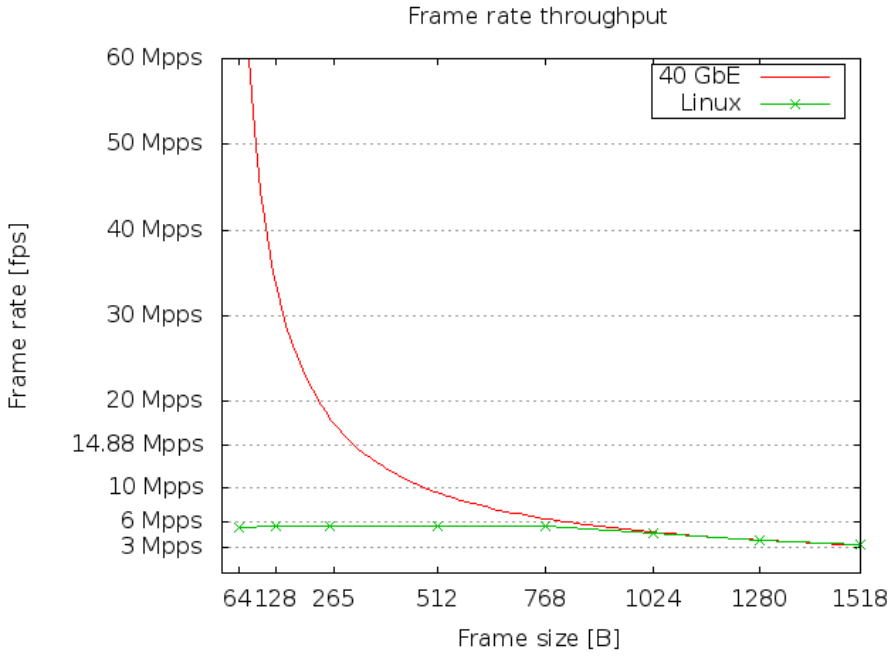
```
echo 1 > /proc/irq/default_smp_affinity      # mask for new registered irqs
echo 0 | tee /proc/irq/*/smp_affinity_list   # assign all IRQs to CPU 0

echo 18 > /proc/irq/177/smp_affinity_list    # assign mlx4-async IRQ to CPU 18

echo 10 > /proc/irq/178/smp_affinity_list    # enp129s0-0 IRQ to CPU 10
echo 11 > /proc/irq/179/smp_affinity_list
echo 12 > /proc/irq/180/smp_affinity_list
echo 13 > /proc/irq/181/smp_affinity_list
echo 14 > /proc/irq/182/smp_affinity_list
echo 15 > /proc/irq/183/smp_affinity_list
echo 16 > /proc/irq/184/smp_affinity_list
echo 17 > /proc/irq/185/smp_affinity_list    # enp192s0-7 IRQ to CPU 17

echo 10 > /proc/irq/186/smp_affinity_list    # enp192s0d1-0 IRQ to CPU 10
echo 11 > /proc/irq/187/smp_affinity_list
echo 12 > /proc/irq/188/smp_affinity_list
echo 13 > /proc/irq/189/smp_affinity_list
echo 14 > /proc/irq/190/smp_affinity_list
echo 15 > /proc/irq/191/smp_affinity_list
echo 16 > /proc/irq/192/smp_affinity_list
echo 17 > /proc/irq/193/smp_affinity_list    # enp192s0d1-7 IRQ to CPU 17

# clear XPS on both interfaces
echo "0" | tee /sys/class/net/enp192s0/queues/tx-*/xps_cpus
echo "0" | tee /sys/class/net/enp192s0d1/queues/tx-*/xps_cpus

# use the IRQ mask to assign XPS
cat /proc/irq/178/smp_affinity > /sys/class/net/enp129s0/queues/tx-0/xps_cpus
cat /proc/irq/179/smp_affinity > /sys/class/net/enp129s0/queues/tx-1/xps_cpus
cat /proc/irq/180/smp_affinity > /sys/class/net/enp129s0/queues/tx-2/xps_cpus
cat /proc/irq/181/smp_affinity > /sys/class/net/enp129s0/queues/tx-3/xps_cpus
```

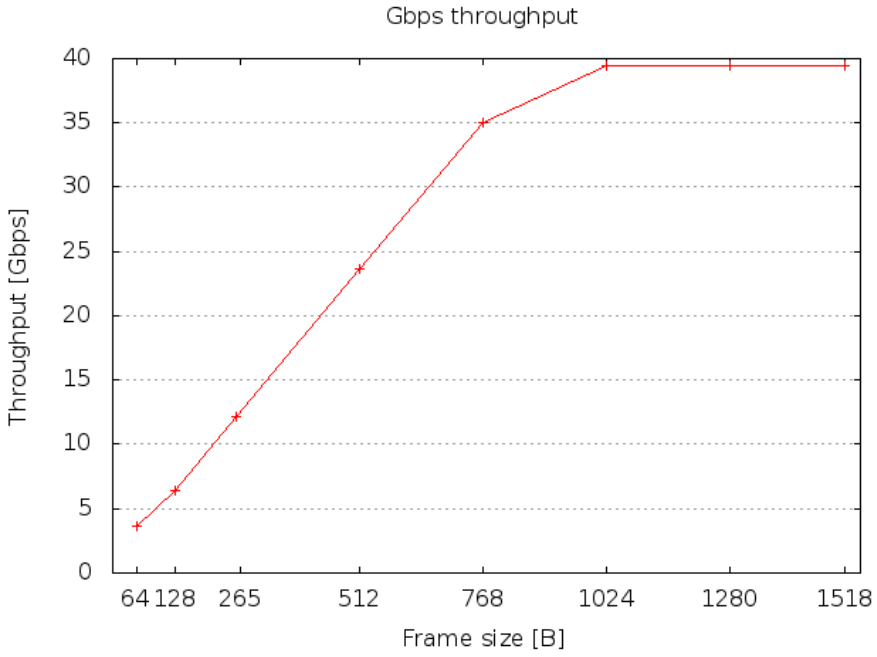
```
cat /proc/irq/182/smp_affinity > /sys/class/net/enp129s0/queues/tx-4/xps_cpus
cat /proc/irq/183/smp_affinity > /sys/class/net/enp129s0/queues/tx-5/xps_cpus
cat /proc/irq/184/smp_affinity > /sys/class/net/enp129s0/queues/tx-6/xps_cpus
cat /proc/irq/185/smp_affinity > /sys/class/net/enp129s0/queues/tx-7/xps_cpus

cat /proc/irq/186/smp_affinity > /sys/class/net/enp129s0d1/queues/tx-0/xps_
    cpus
cat /proc/irq/187/smp_affinity > /sys/class/net/enp129s0d1/queues/tx-1/xps_
    cpus
cat /proc/irq/188/smp_affinity > /sys/class/net/enp129s0d1/queues/tx-2/xps_
    cpus
cat /proc/irq/189/smp_affinity > /sys/class/net/enp129s0d1/queues/tx-3/xps_
    cpus
cat /proc/irq/190/smp_affinity > /sys/class/net/enp129s0d1/queues/tx-4/xps_
    cpus
cat /proc/irq/191/smp_affinity > /sys/class/net/enp129s0d1/queues/tx-5/xps_
    cpus
cat /proc/irq/192/smp_affinity > /sys/class/net/enp129s0d1/queues/tx-6/xps_
    cpus
cat /proc/irq/193/smp_affinity > /sys/class/net/enp129s0d1/queues/tx-7/xps_
    cpus
```

The *mlx4* driver uses combined interrupts for RX and TX, therefore each mapped CPU serves RX and TX interrupts for the same packets. Such mapping should lead to a better cache utilisation than in the previous measurement.

Frame size	% of link	bandwidth	frame rate
64	8.99%	3.60 Gb/s	5 350 000
594	68.15%	27.26 Gb/s	5 550 000
1518	99.60%	39.40 Gb/s	3 202 210
AMS-IX	88.25%	35.30 Gb/s	5 800 000

The throughput performance with manual IRQ mappings is equal to the mappings set by the *irqbalance* daemon. The measurement was also configured to use 128, 256, 512, 768, 1024 and 1280 byte sized frames and the following graph presents the results.





The system is able to perform forwarding at nearly line rate speed with frames 1024 B and larger. The following listing shows the actual interrupt mappings obtained from the /proc/interrupts file.

	CPU10	CPU11	CPU12	CPU13	CPU14	CPU15	CPU16	CPU17	CPU18	
178:	298717	0	0	0	0	0	0	0	0	enp129s0-0
179:	0	294264	0	0	0	0	0	0	0	enp129s0-1
180:	0	0	291502	0	0	0	0	0	0	enp129s0-2
181:	0	0	0	296514	0	0	0	0	0	enp129s0-3
182:	0	0	0	0	302588	0	0	0	0	enp129s0-4
183:	0	0	0	0	0	294017	0	0	0	enp129s0-5
184:	0	0	0	0	0	0	294314	0	0	enp129s0-6
185:	0	0	0	0	0	0	0	302088	0	enp129s0-7
186:	402863	0	0	0	0	0	0	0	0	enp129s0d1-0
187:	0	411668	0	0	0	0	0	0	0	enp129s0d1-1
188:	0	0	416164	0	0	0	0	0	0	enp129s0d1-2
189:	0	0	0	422023	0	0	0	0	0	enp129s0d1-3
190:	0	0	0	0	421805	0	0	0	0	enp129s0d1-4
191:	0	0	0	0	0	415759	0	0	0	enp129s0d1-5
192:	0	0	0	0	0	0	402918	0	0	enp129s0d1-6
193:	0	0	0	0	0	0	0	413299	0	enp129s0d1-7

The RX and TX interrupts are spread across 8 CPUs. The advantage of this mapping against the mapping done by the *irqbalance* daemon is that it requires half the CPUs and the IRQ serving should cause fewer cache misses as well. The listing bellow shows the cache statistics.

```
L3MISS: L3 cache misses
L2MISS: L2 cache misses (including other core's L2 cache *hits*)
L3HIT : L3 cache hit ratio (0.00-1.00)
L2HIT : L2 cache hit ratio (0.00-1.00)
L3CLK : ratio of CPU cycles lost due to L3 cache misses (0.00-1.00)
L2CLK : ratio of CPU cycles lost due to missing L2 cache (0.00-1.00)
L3OCC : L3 occupancy (in KBytes)

 Core (SKT) | L3MISS | L2MISS | L3HIT | L2HIT | L3CLK | L2CLK |  L3OCC
   0     0      5799       33 K   0.82   0.24   0.19   0.18       0
   1     0       122       2488   0.95   0.15   0.03   0.14       0
   2     0       374       3928   0.90   0.19   0.00   0.00       0
   3     0         8        532   0.98   0.15   0.01   0.08      80
   4     0         6        519   0.99   0.16   0.00   0.08       0
   5     0        13        540   0.98   0.13   0.01   0.07       0
   6     0        12        552   0.98   0.14   0.01   0.07       0
   7     0        12        560   0.98   0.14   0.01   0.08     120
   8     0         9        547   0.98   0.13   0.01   0.08       0
   9     0        25        593   0.96   0.13   0.02   0.08       0
  10     1      9527       6014 K  1.00   0.51   0.00   0.14    1080
  11     1      9633       6096 K  1.00   0.52   0.00   0.14    1280
  12     1      9440       6257 K  1.00   0.53   0.00   0.15    1280
  13     1      9349       6069 K  1.00   0.54   0.00   0.14     960
  14     1      9147       6007 K  1.00   0.50   0.00   0.14    1000
  15     1      9270       6043 K  1.00   0.52   0.00   0.15    1160
  16     1      9382       6230 K  1.00   0.52   0.00   0.15    1560
  17     1      9249       6055 K  1.00   0.56   0.00   0.14     920
  18     1      1395       8002   0.83   0.15   0.29   0.29      40
  19     1       198       1129   0.82   0.12   0.13   0.14       0
  20     0       171       1042   0.84   0.08   0.07   0.11      40
  21     0        35       1176   0.97   0.16   0.02   0.17       0
  22     0        24        715   0.97   0.13   0.01   0.11      40
  23     0        18        537   0.97   0.15   0.01   0.09       0
  24     0         5        536   0.99   0.16   0.00   0.09       0
  25     0        11        533   0.98   0.14   0.01   0.08       0
  26     0        15        541   0.97   0.14   0.01   0.08       0
  27     0        11        538   0.98   0.16   0.01   0.09       0
  28     0        14        547   0.97   0.15   0.01   0.09       0
  29     0        46        672   0.93   0.12   0.03   0.10       0
  30     1       557       1200   0.54   0.34   0.03   0.01       0
  31     1       181        612   0.70   0.21   0.26   0.14      40
  32     1       192        604   0.68   0.23   0.25   0.12       0
  33     1       246        634   0.61   0.22   0.31   0.12       0
  34     1       202        622   0.68   0.22   0.28   0.15       0
  35     1       144        599   0.76   0.25   0.20   0.15       0
  36     1       127        620   0.80   0.22   0.17   0.15       0
  37     1       142        619   0.77   0.20   0.21   0.16      40
  38     1       132        685   0.81   0.11   0.13   0.15       0
  39     1       308        921   0.22   0.43   0.38   0.02     360
-------------------------------------------------------------------
 SKT     0      6730        50 K   0.87   0.21   0.03   0.04     280
 SKT     1        85 K      48 M   1.00   0.52   0.00   0.14    9720
-------------------------------------------------------------------
 TOTAL   *        92 K      48 M   1.00   0.52   0.00   0.14     N/A
```

As expected, the total cache miss count is lower with the manual IRQ mappings.
The manual mappings are the best IRQ affinity settings in terms of number of CPUs used, cache use and predictability.

6.1.8 Measurement 8 - single IPv6 flow

The measurement serves as a comparison between IPv4 and IPv6 processing performance. The measurement can be directly compared to Measurement 3.

Frame size	% of link	bandwidth	frame rate
78	1.76%	0.71 Gb/s	900 000
594	11.05%	4.42 Gb/s	900 000
1518	27.68%	11.07 Gb/s	900 000
AMS-IX	13.69%	5.48 Gb/s	900 000

The IPv6 processing performance is 10% lower than the IPv4 processing performance when forwarding traffic on a single core.
The following listing shows the output of the *perf* utility:

```
perf top -C 10
  11.75%  [kernel]  [k] ip6t_do_table
  10.37%  [kernel]  [k] _raw_spin_lock
   8.43%  [kernel]  [k] fib6_lookup
   4.93%  [kernel]  [k] ip6_forward
   3.66%  [kernel]  [k] fib6_get_table
   3.48%  [kernel]  [k] ip6_rcv_finish
   3.40%  [kernel]  [k] build_skb
   3.26%  [kernel]  [k] __netif_receive_skb_core
   3.01%  [kernel]  [k] mlx4_en_complete_rx_desc
   3.00%  [kernel]  [k] _raw_read_unlock_bh
   2.65%  [kernel]  [k] mlx4_en_process_rx_cq
   2.62%  [kernel]  [k] dst_release
```

As in the case of IPv4, the CPU spends most of the time on the actual lookup and locking.

6.1.9 Measurement 9 - 32 IPv6 flows with manual IRQ affinity mappings

The measurement can be directly compared to Measurement 7.

Frame size	% of link	bandwidth	frame rate
78	5.88%	2.35 Gb/s	3 000 000
594	36.84%	14.74 Gb/s	3 550 000
1518	98.50%	39.40 Gb/s	3 202 210
AMS-IX	54.77%	21.91 Gb/s	3 600 000

The measurement featuring 1518 octet frames was configured to use 98.50% of the link bandwidth. The result shows a significant performance drop when routing multiple IPv6 flows.
This performance drop can be investigated by changing the number of UDP flows to 8 and observing the interrupt distribution. The following listing shows the partial output of the /proc/interrupts file when routing 8 flows.

```
           ...  CPU12  CPU13  CPU14  CPU15  CPU16  CPU17  CPU18  ...
178:  ...      0      0      0      0      0      0      0  ...  enp129s0d1-0
179:  ...      0      0      0      0      0      0      0  ...  enp129s0d1-1
```

```
180:  ...      0      0      0      0      0      0      0  ...  enp129s0d1-2
181:  ...      0  32513      0      0      0      0      0  ...  enp129s0d1-3
182:  ...      0      0      0      0      0      0      0  ...  enp129s0d1-4
183:  ...      0      0      0      0      0      0      0  ...  enp129s0d1-5
184:  ...      0      0      0      0      0      0      0  ...  enp129s0d1-6
185:  ...      0      0      0      0      0  33543      0  ...  enp129s0d1-7
186:  ...      0      0      0      0      0      0      0  ...  enp129s0-0
187:  ...      0      0      0      0      0      0      0  ...  enp129s0-1
188:  ...      0      0      0      0      0      0      0  ...  enp129s0-2
189:  ...      0  19384      0      0      0      0      0  ...  enp129s0-3
190:  ...      0      0      0      0      0      0      0  ...  enp129s0-4
191:  ...      0      0      0      0      0      0      0  ...  enp129s0-5
192:  ...      0      0      0      0      0      0      0  ...  enp129s0-6
193:  ...      0      0      0      0      0  19576      0  ...  enp129s0-7
```

In contrast to IPv4, the RSS does not distribute IPv6 packets uniformly. As a result of this, the routing performance of IPv6 protocol is lower than in case of IPv4. The traffic is also not distributed uniformly when routing 32 flows, however, this issue cannot be detected from reading the /proc/interrupts file due to interrupt mitigation mechanism used by NAPI, as described in subsection 3.4.1.

6.2 Upstream mainline kernel 4.0.2

The following measurements use the upstream Linux kernel downloaded form elrepo, as described in 4.2. This is the latest upstream kernel version as of 6th May 2015.

6.2.1 1 IPv4 flow

Frame size	% of link	bandwidth	frame rate
AMS-IX	15.22%	6.09 Gb/s	1 000 000

Routing performance of the Linux kernel version 4.0.2 on a single core is the same as the CentOS 7 distribution kernel 3.10.0-123. The following listing shows the output of the *perf* tool.

```
perf top -C 10
  11.25%  [kernel]               [k] _raw_spin_lock
   9.69%  [kernel]               [k] fib_table_lookup
   5.58%  [kernel]               [k] mlx4_en_xmit
   4.87%  [kernel]               [k] __memcpy
   4.65%  [kernel]               [k] mlx4_en_process_rx_cq
   2.98%  [kernel]               [k] put_compound_page
   2.73%  [kernel]               [k] __netif_receive_skb_core
   2.64%  [kernel]               [k] mlx4_en_poll_tx_cq
   2.43%  [kernel]               [k] ip_route_input_noref
   2.15%  [kernel]               [k] ip_rcv
```

As in case of the CentOS 7 distribution kernel, the FIB table lookup and locking take most of the CPU's time.

6.2.2 32 IPv4 flows

The measurement uses manual IRQ affinity mappings and it can be directly compared to Measurement 7.

Frame size	% of link	bandwidth	frame rate
AMS-IX	86.73%	34.70 Gb/s	5 700 000

The upstream kernel 4.0.2 performs approx. 2% slower than the CentOS 7 distribution kernel.

6.2.3 32 IPv4 flows on 16 queues

The measurement uses the *ethtool* utility to configure the Mellanox ConnectX-3 EN adapter to use 16 receive queues per each port. The network adapter supports up to 16 receive queues per port with RSS, as described in section 4.1.

```
ethtool -L enp129s0 rx 16
ethtool -L enp129s0d1 rx 16
```

The following table shows the result.

Frame size	% of link	bandwidth	frame rate
AMS-IX	81.40%	32.56 Gb/s	5 350 000

Although the networking code runs on 16 logical CPUs simultaneously, the performance is lower than in case of using 8 queues only. This may be caused by the overhead of the Hyper-Threading technology.

6.2.4 Single IPv6 flow

The measurement can be directly compared to Measurement 8.

Frame size	% of link	bandwidth	frame rate
AMS-IX	13.69%	5.48 Gb/s	900 000

The single flow routing performance is the same as in case of the CentOS 7 distribution kernel. The following listing shows the output of the *perf* tool.

```
perf top -C 10
   9.43%  [kernel]  [k] _raw_spin_lock
   6.29%  [kernel]  [k] fib6_lookup_1
   5.49%  [kernel]  [k] mlx4_en_process_rx_cq
   4.12%  [kernel]  [k] mlx4_en_xmit
   3.62%  [kernel]  [k] memcpy
   3.09%  [kernel]  [k] mlx4_en_poll_tx_cq
   3.06%  [kernel]  [k] ip6_pol_route.isra.44
   2.84%  [kernel]  [k] irq_entries_start
   2.22%  [kernel]  [k] dst_release
   2.10%  [kernel]  [k] __netif_receive_skb_core
   2.08%  [kernel]  [k] __local_bh_enable_ip
   1.81%  [kernel]  [k] build_skb
   1.71%  [kernel]  [k] ip6_forward
   1.69%  [kernel]  [k] ipv6_rcv
```

6.2.5 32 IPv6 flows

The measurement can be directly compared to Measurement 9.

Frame size	% of link	bandwidth	frame rate
AMS-IX	54.77%	21.91 Gb/s	3 600 000

As in Measurement 9, the low performance is caused by the RSS distribution done by the network adapter.

6.3 Settings influence

There are various system settings, which can impact the routing performance of the GNU/Linux-based system. The influence of some the most interesting settings is demonstrated by the following measurements.

6.3.1 Disabled Hyper-Threading

The measurement features routing of a single IPv4 flow with the Hyper-Threading technology disabled.

Frame size	% of link	bandwidth	frame rate
64	1.93%	0.77 Gb/s	1 150 000
594	14.12%	5.65 Gb/s	1 150 000
1518	35.37%	14.15 Gb/s	1 150 000
AMS-IX	17.50%	7.00 Gb/s	1 150 000

A single core routing performance increases by 15% with disabled Hyper-Threading.
Routing of 32 IPv4 flows with manual IRQ affinity mappings was tested to investigate how is the routing performance influenced when the networking code runs on multiple cores with Hyper-Threading disabled.

Frame size	% of link	bandwidth	frame rate
64	9.07%	3.63 Gb/s	5 400 000
594	68.77%	27.51 Gb/s	5 600 000
1518	98.50%	39.40 Gb/s	3 202 210
AMS-IX	89.77%	35.91 Gb/s	5 900 000

Disabled Hyper-Threading provides only 2% performance increase when routing the AMS-IX traffic on multiple cores. The logical cores which are not utilised do not decrease performance significantly, which means that Hyper-Threading is highly optimised on Intel Xeon E5-2660 v3.

6.3.2 Netdev_budget

The measurement features increased *netdev_budget* NAPI parameter from its default value of 300 to 3 000. The parameter is described in section 3.4. The *netdev_budget* can be configured using *procfs*.

```
echo 3000 > /proc/sys/net/core/netdev_budget
```

The following table shows the results.

Frame size	% of link	bandwidth	frame rate
AMS-IX	88.25%	35.30 Gb/s	5 800 000

Increasing the *netdev_budget* has no influence on routing performance, which means that the raise softirq mechanism is highly optimised.

6.3.3 Reverse path filter

The measurement features Reverse path filter enabled, which is described in subsection 4.4.2. The rp_filter can be enabled via *procfs*.

```
echo 1 | tee /proc/sys/net/ipv4/conf/*/rp_filter
```

The following table shows the results.

Frame size	% of link	bandwidth	frame rate
AMS-IX	55.54%	22.21 Gb/s	3 650 000

The rp_filter introduces a significant performance drop of about 37%. The following listing shows the output of the *perf* utility.

```
perf top -C 10
  39.88%  [kernel]  [k] fib_table_lookup
   8.35%  [kernel]  [k] check_leaf.isra.7
   6.43%  [kernel]  [k] _raw_spin_lock
   2.94%  [kernel]  [k] mlx4_en_xmit
   2.40%  [kernel]  [k] memcpy
   2.32%  [kernel]  [k] __netif_receive_skb_core
   2.18%  [kernel]  [k] mlx4_en_process_rx_cq
   2.14%  [kernel]  [k] fib_validate_source
   1.40%  [kernel]  [k] ip_route_input_noref
```

The *fib_table_lookup* is performed twice for each packet and it is therefore taking much more of the CPU time. There is also new *fib_validate_source* function, which calls the actual *fib_table_lookup*.

In bidirectional routing with Reverse path filter enabled, it may be worth changing the default RSS hash key to a symmetric one. The RSS hash key is a taken as input by the Toeplitz hash function, as described in section 3.6. A symmetric RSS key would lead to processing both directions of the same flow on the same CPU, therefore the result of the *fib_table_lookup* could be fetched from the CPU's cache [61].

6.3.4 SELinux

The measurement features enabled SELinux in Enforcing mode, which can be set in the */etc/sysconfig/selinux* file.

Frame size	% of link	bandwidth	frame rate
AMS-IX	80.64%	32.26 Gb/s	5 300 000

SELinux negatively impacts the routing performance by approx. 8.6%.

6.3.5 Firewall

The measurement features enabled Netfilter and 10 000 filtering rules inserted. The following commands were used to setup the measurement.

```
systemctl start firewalld
iptables -F
```

```
for i in `seq 10001 20000`; do iptables -A INPUT -p udp --dport $i -j DROP;
    done
```

The *firewalld* loads Netfilter modules automatically.

```
lsmod | grep nf
nf_conntrack_ipv6       18738  5
nf_defrag_ipv6          34651  1 nf_conntrack_ipv6
nf_nat_ipv6             13279  1 ip6table_nat
nf_conntrack_ipv4       14862  4
nf_defrag_ipv4          12729  1 nf_conntrack_ipv4
nf_nat_ipv4             13263  1 iptable_nat
nf_nat                  21798  4 nf_nat_ipv4,nf_nat_ipv6,ip6table_nat,
    iptable_nat
nf_conntrack           101024  8 nf_nat,nf_nat_ipv4,nf_nat_ipv6,xt_conntrack,
    ip6table_nat,iptable_nat,nf_conntrack_ipv4,nf_conntrack_ipv6
```

The following table shows the results.

Frame size	% of link	bandwidth	frame rate
AMS-IX	88.25%	35.30 Gb/s	5 800 000

The table above shows that 10 000 filtering rules have no impact on throughput. The following listing shows the output of the *perf* utility.

```
perf top -C 10
  16.37%  [kernel]  [k] fib_table_lookup
  11.48%  [kernel]  [k] _raw_spin_lock
   6.20%  [kernel]  [k] mlx4_en_process_rx_cq
   5.33%  [kernel]  [k] memcpy
   3.86%  [kernel]  [k] kmem_cache_alloc
   2.90%  [kernel]  [k] check_leaf.isra.7
   2.89%  [kernel]  [k] ipt_do_table
   2.57%  [kernel]  [k] mlx4_en_xmit
   2.42%  [kernel]  [k] mlx4_en_alloc_frags
```

The *ipt_do_table* function is responsible for the actual filtering.

6.3.6 NAT

The measurement features a simple Network Address Translation using Netfilter's *MASQUERADE*, which was configured using *iptables*.

```
iptables -t nat -A POSTROUTING -o enp129s0 -j MASQUERADE

lsmod | grep nf
  nf_conntrack_ipv4       14862  1
  nf_defrag_ipv4          12729  1 nf_conntrack_ipv4
  nf_nat_ipv4             13263  1 iptable_nat
  nf_nat                  21798  3 ipt_MASQUERADE,nf_nat_ipv4,iptable_nat
  nf_conntrack           101024  5 ipt_MASQUERADE,nf_nat,nf_nat_ipv4,
      iptable_nat,nf_conntrack_ipv4
```

The following table shows the results.

Frame size	% of link	bandwidth	frame rate
AMS-IX	65.43%	26.17 Gb/s	4 300 000

The performance decreased by approx. 25% when performing Network Address Translation using *MASQUERADE*. Additionally, Netfilter provides other types of Network Address Translation such as *SNAT* or *SAME*, however, their description and benchmarking are outside the scope of this thesis. The following listing shows the output of the *perf* tool.

```
perf top -C 10
  12.71%  [kernel]  [k] fib_table_lookup
  10.84%  [kernel]  [k] _raw_spin_lock
   6.05%  [kernel]  [k] memcpy
   5.13%  [kernel]  [k] nf_xfrm_me_harder
   4.19%  [kernel]  [k] mlx4_en_process_rx_cq
   2.71%  [kernel]  [k] mlx4_en_xmit
   2.43%  [kernel]  [k] ip_route_input_noref
   2.14%  [kernel]  [k] nf_iterate
   2.07%  [kernel]  [k] check_leaf.isra.7
   2.02%  [kernel]  [k] __netif_receive_skb_core
```

The *nf_xfrm_me_harder* function is responsible for the actual address translation.

6.4 BGP routes

The following measurement is performed with the Internet routes obtained from RIPE's BGP data. The data contains 538 738 routes at the time of writing. The complete setup of loading the routes into the kernel's FIB is described in appendix A. The FIB statistics are exported via the */proc/net/fib_triestat* file, as described in subsection 4.3.2. The following listing shows the content of the file after loading the Internet routes to the kernel's FIB.

```
Basic info: size of leaf: 40 bytes, size of tnode: 40 bytes.
Main:
  Aver depth:      2.43
  Max depth:       8
  Leaves:          503308
  Prefixes:        538739
  Internal nodes: 114429
    1: 58727  2: 26171  3: 14805  4: 7315  5: 4240  6: 2103  7: 1065  8: 2
        17: 1
  Pointers: 995794
Null ptrs: 378058
Total size: 61373  kB

Counters:
---------
gets = 14129134
backtracks = 1913823
semantic match passed = 15973885
semantic match miss = 0
null node hit= 1360956
skipped node resize = 0

Local:
  Aver depth:      3.08
  Max depth:       4
  Leaves:          12
  Prefixes:        13
  Internal nodes: 7
    1: 6  3: 1
  Pointers: 20
```

```
Null ptrs: 2
Total size: 2  kB

Counters:
---------

gets = 13959019
backtracks = 15201204
semantic match passed = 103193
semantic match miss = 0
null node hit= 8238092
skipped node resize = 0
```

Each generated packet is assigned a random destination IP address, as described in section 5.3. Such configuration avoids having the previous lookup result in the CPU's cache.

Frame size	% of link	bandwidth	frame rate
AMS-IX	77.60%	30.95 Gb/s	5 100 000

As it was expected, the routing performance decreases when performing lookups for many destination addresses. The performance drop is approx. 12%. The following listing shows the cache utilisation.

```
L3MISS: L3 cache misses
L2MISS: L2 cache misses (including other core's L2 cache *hits*)
L3HIT : L3 cache hit ratio (0.00-1.00)
L2HIT : L2 cache hit ratio (0.00-1.00)
L3CLK : ratio of CPU cycles lost due to L3 cache misses (0.00-1.00)
L2CLK : ratio of CPU cycles lost due to missing L2 cache (0.00-1.00)
L3OCC : L3 occupancy (in KBytes)

 Core (SKT) | L3MISS | L2MISS | L3HIT | L2HIT | L3CLK | L2CLK |  L3OCC
   0    0     4048      16 K    0.75    0.40    0.18    0.11     360
   1    0      574      3257    0.82    0.17    0.00    0.00     160
   2    0      515      4977    0.90    0.13    0.06    0.10      40
   3    0        5       539    0.99    0.11    0.00    0.12      40
   4    0        5       526    0.99    0.13    0.00    0.11       0
   5    0        5       532    0.99    0.12    0.00    0.10       0
   6    0        4       548    0.99    0.11    0.00    0.09       0
   7    0        4       541    0.99    0.12    0.00    0.09       0
   8    0        6       548    0.99    0.11    0.00    0.08      40
   9    0       21       550    0.96    0.13    0.01    0.07      40
  10    1     1410 K    9962 K  0.86    0.47    0.09    0.14    2480
  11    1     1400 K    9901 K  0.86    0.46    0.09    0.14    2160
  12    1     1396 K    9872 K  0.86    0.46    0.09    0.14    2200
  13    1     1396 K    9897 K  0.86    0.47    0.09    0.14    2000
  14    1     1402 K    9872 K  0.86    0.46    0.09    0.14    2720
  15    1     1395 K    9861 K  0.86    0.49    0.09    0.14    2160
  16    1     1409 K    9827 K  0.86    0.48    0.09    0.14    2200
  17    1     1402 K    9867 K  0.86    0.46    0.09    0.14    2240
  18    1      398       827    0.52    0.07    0.43    0.11       0
  19    1      200       891    0.78    0.07    0.08    0.09       0
  20    0      213       675    0.68    0.09    0.11    0.06       0
  21    0        7       656    0.99    0.09    0.01    0.13       0
  22    0        6       747    0.99    0.07    0.00    0.12      40
  23    0       13       548    0.98    0.12    0.01    0.12       0
  24    0        4       543    0.99    0.14    0.00    0.11      40
  25    0        5       529    0.99    0.13    0.00    0.10      80
  26    0       25       761    0.97    0.09    0.02    0.13       0
  27    0        5       707    0.99    0.09    0.00    0.12       0
```

28	0	5	716	0.99	0.09	0.00	0.11	0
29	0	45	643	0.93	0.11	0.03	0.09	0
30	1	362	704	0.49	0.22	0.33	0.06	40
31	1	206	643	0.68	0.21	0.24	0.12	0
32	1	187	644	0.71	0.18	0.22	0.12	0
33	1	188	635	0.70	0.38	0.24	0.13	0
34	1	193	619	0.69	0.20	0.27	0.14	0
35	1	208	614	0.66	0.11	0.34	0.15	40
36	1	184	639	0.71	0.21	0.26	0.15	0
37	1	194	642	0.70	0.19	0.26	0.13	0
38	1	181	616	0.71	0.09	0.23	0.13	0
39	1	22 K	29 K	0.24	0.16	0.85	0.06	200
SKT	0	5515	35 K	0.84	0.28	0.02	0.03	840
SKT	1	11 M	79 M	0.86	0.47	0.09	0.14	18440
TOTAL	*	11 M	79 M	0.86	0.47	0.09	0.14	N/A

The FIB table lookup introduces about 1.4 million L3 cache misses per second on each logical CPU, which is the source of the performance drop.

6.5 Summary

The Linux kernel is able to forward up to 5.9 million frames per second with the system settings used in the measurements, while the default configuration limits the performance to 1.6 million frames per second. The most significant performance drop is caused by the Reverse path filtering feature, which requires to perform the FIB lookup twice for each packet. Disabling the Hyper-Threading technology provides a negligible performance improvement. The FIB lookup with imported prefixes from the BGP table does not introduce a large performance drop thanks to the CPU's 20 MB L3 cache. Figure 6.1 shows the presented results. Further performance improvement may require Linux kernel recompilation.

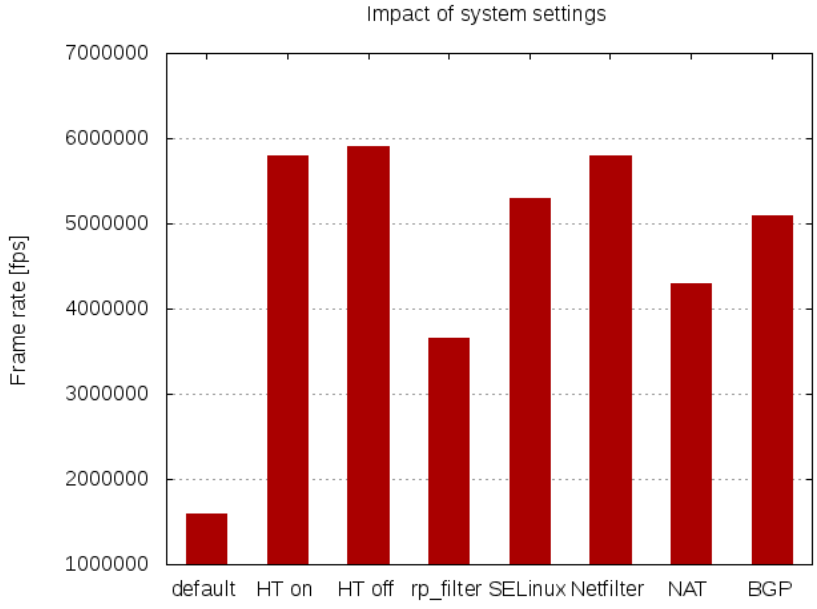


Figure 6.1: Measurements overview

Chapter 7

Conclusion

The thesis provides description of the principles behind packet processing in the Linux kernel, as well as a general description of the 40 Gigabit Ethernet protocol and its performance limitation. An advanced knowledge of the principles described in the thesis is required to perform the correct system settings for maximum routing performance with the GNU/Linux operating system.

The CentOS 7 operating system was installed to perform the measurements. The system features Linux kernel based on version 3.10. Additionally, upstream kernel 4.0.2 was installed. The Linux kernel routing performance was measured under different scenarios, including simulation of a real internet traffic, based on the data provided by the Amsterdam Internet Exchange. A custom frame-size distribution was configured for this purpose and used in the experiments.

40Gbit software router with GNU/Linux may provide a reasonable alternative to proprietary hardware-based routers. There are several settings of the default installation which negatively impact routing performance. The thesis describes how to adjust the settings for the purpose of maximum throughput.

The overall routing performance of the Linux kernel is sufficient for routing 35.91 Gbps of the simulated Internet traffic on a general purpose Intel Xeon CPU, that is 5.9 million frames per second. The CPU features 8 physical cores and the Hyper-Threading technology (16 logical cores). The Mellanox ConnectX-3 EN adapter is able to scale the packet processing up to 16 cores, however, the measurements show that utilising 8 cores performs slightly better. Since the packet processing is highly optimised in terms of cache hit ratio, this may be a constraint of the Hyper-Threading technology. The measurement involving the Internet BGP routes shows, that the Linux kernel is sufficient for routing 30.95 Gbps of the simulated Internet traffic, that is 5.1 million frames per second.

The Linux kernel uses advantage of its scaling mechanisms to perform well in network processing, however, a single-core packet processing must be improved to achieve better results. The Linux kernel is able to route approx. 1 150 000 frames per second using a single physical core. This is also the limitation of a single network flow processing, since the scaling mechanisms implemented in the Linux kernel and network adapters are based on processing each flow by a different CPU.

The thesis can be further extended through a comparison against hardware-based routers. The future work may compare latency, throughput or power consumption of both systems. Additionally, various user-space frameworks focused on improving the GNU/Linux network performance may be evaluated, such as Data Plane Development Kit. Further work extending the thesis may include comparison of the frameworks with the presented results.

Bibliography

[1] 10 Gigabit Ethernet Alliance. TCP/IP offload Engine TOE. http://www.10gea.org/whitepapers/tcpip-offload-engine-toe/, 2010 [cit. 2014-09-01].

[2] Ethernet Alliance. Ethernet Jumbo Frames. http://www.ethernetalliance.org/wp-content/uploads/2011/10/EA-Ethernet-Jumbo-Frames-v0-1.pdf, 2009 [cit. 2014-09-02].

[3] J. Arkko, M. Cotton, and L. Vegoda. Ipv4 address blocks reserved for documentation. http://www.rfc-editor.org/rfc/rfc5737.txt, 2010 [cit. 2015-02-02].

[4] E. Banks. 40GbE Over A Single MMF Pair? With QSFP-40G-SR-BD, You Can. http://ethancbanks.com/2013/11/09/40gbe-over-a-single-mmf-pair-with-qsfp-40g-sr-bd-you-can/, 2013 [cit. 2014-09-23].

[5] C. Benvenuti. *Understanding Linux Network Internals*. O'Reilly Media, 2005. ISBN-978-0-596-00255-8.

[6] S. Bradner and J. McQuaid. Benchmarking methodology for network interconnect devices. http://www.rfc-editor.org/rfc/rfc2544.txt, 1999 [cit. 2015-02-08].

[7] D. Brodowski. Linux cpufreq gvoernors - information for users and developers. https://www.kernel.org/doc/Documentation/cpu-freq/governors.txt, 2014 [cit. 2015-02-10].

[8] G. Chanda. The Market Need for 40 Gigabit Ethernet. http://www.cisco.com/c/en/us/products/collateral/switches/catalyst-6500-series-switches/white_paper_c11-696667.pdf, 2012 [cit. 2014-09-24].

[9] Spirent Communications. Spirent spt-3u. http://www.spirent.com/Ethernet_Testing/Platforms/3U_Chassis, 2015 [cit. 2015-01-08].

[10] Spirent Communications. Spirent testcenter. http://www.spirent.com/Ethernet_Testing/Software/TestCenter, 2015 [cit. 2015-03-03].

[11] J. Corbet. Linux and TCP offload engines. http://lwn.net/Articles/148697/, 2005 [cit. 2014-09-03].

[12] J. Corbet. Reworking NAPI. http://lwn.net/Articles/214457/, 2006 [cit. 2014-10-01].

[13] J. Corbet. Multiqueue networking. http://lwn.net/Articles/289137/, 2008 [cit. 2015-01-05].

[14] J. Corbet. JLS2009: Generic receive offload. https://lwn.net/Articles/358910/, 2009 [cit. 2014-09-01].

[15] J. Corbet. Receive packet steering. http://lwn.net/Articles/362339/, 2009 [cit. 2015-01-06].

[16] J. Corbet. Low-latency Ethernet device polling. http://lwn.net/Articles/551284/, 2013 [cit. 2014-09-03].

[17] Intel Corporation. Intel xeon processor e5-2660 v3. http://ark.intel.com/products/64584/Intel-Xeon-Processor-E5-2660-20M-Cache-2_20-GHz-8_00-GTs-Intel-QPI, 2014 [cit. 2015-01-08].

[18] J. Cummings and E. Tamir. Open Source Kernel Enhancements for Low Latency Sockets using Busy Poll. http://www.intel.com/content/dam/www/public/us/en/documents/white-papers/open-source-kernel-enhancements-paper.pdf, 2013 [cit. 2014-10-08].

[19] A. Duyck and P. Waskiewicz. HOWTO for multiqueue network device support. https://www.kernel.org/doc/Documentation/networking/multiqueue.txt, 2014 [cit. 2014-12-01].

[20] N. Edwards. Theoretical vs. Actual Bandwidth: PCI Express and Thunderbolt. http://www.tested.com/tech/457440-theoretical-vs-actual-bandwidth-pci-express-and-thunderbolt/, 2013 [cit. 2014-09-25].

[21] Jan Kaluza et al. tuned. https://fedorahosted.org/tuned/, 2014 [cit. 2014-12-30].

[22] L. Torvalds et al. Linux kernel source code version 3.10.61. http://www.kernel.org/, 2014.

[23] L. Torvalds et al. /proc/sys/net/ipv4/* Variables. https://www.kernel.org/doc/Documentation/networking/ip-sysctl.txt, 2014 [cit. 2015-03-03].

[24] L. Torvalds et al. NO_HZ: Reducing Scheduling-Clock Ticks. https://www.kernel.org/doc/Documentation/timers/NO_HZ.txt, 2014 [cit. 2015-04-04].

[25] P. Holasek et al. irqbalance source code version 1.0.8. https://github.com/Irqbalance/irqbalance/tree/v1.0.8, 2014 [cit. 2014-12-30].

[26] Linux Foundation. Napi. http://www.linuxfoundation.org/collaborate/workgroups/networking/napi„ 2009 [cit. 2014-08-30].

[27] Linux Foundation. TOE. http://www.linuxfoundation.org/collaborate/workgroups/networking/toe, 2009 [cit. 2014-10-08].

[28] T. Herbert and W. de Bruijn. Scaling in the Linux Networking Stack. https://www.kernel.org/doc/Documentation/networking/scaling.txt, 2014 [cit. 2014-12-01].

[29] G. Huston. Bgp analysis reports. http://bgp.potaroo.net/index-bgp.html, 2015 [cit. 2015-03-20].

[30] G. Huston, A. Lord, and P. Smith. Ipv6 address prefix reserved for documentation. http://tools.ietf.org/html/rfc3849, 2004 [cit. 2015-02-02].

[31] AMS IX. Frame size distribution. https://ams-ix.net/technical/statistics/sflow-stats/frame-size-distribution, 2015 [cit. 2015-01-30].

[32] P. Kish. 10GBASE-T Today or 40GBASE-T Tomorrow. http://www.belden.com/blog/datacenters/10GBASE-T-Today-or-40GBASE-T-Tomorrow.cfm, 2014 [cit. 2014-09-23].

[33] O. Lichtner. Networking Subsystem Configuration Interface, 2014. http://www.fit.vutbr.cz/study/DP/DP.php.cs?id=16789.

[34] R. Love. *Linux Kernel Development Second Edition*. Sams Publishing, 2005. ISBN-0-672-32720-1.

[35] Xillybus Ltd. Down to the TLP: How PCI express devices talk (Part I). http://xillybus.com/tutorials/pci-express-tlp-pcie-primer-tutorial-guide-1, 2014 [cit. 2014-09-03].

[36] D. Miller. Removing The Linux Routing Cache. http://vger.kernel.org/~davem/columbia2012.pdf, 2012 [cit. 2014-09-04].

[37] I. Molnar and M. Krasnyansky. Smp irq affinity. https://www.kernel.org/doc/Documentation/IRQ-affinity.txt, 2014 [cit. 2015-03-14].

[38] T. Nguyen and M. Wilcox. The MSI Driver Guide HOWTO. https://www.kernel.org/doc/Documentation/PCI/MSI-HOWTO.txt, 2008 [cit. 2015-01-28].

[39] PCI-SIG. PCI Express Base Specification Revision 3.0. http://komposter.com.ua/documents/PCI_Express_Base_Specification_Revision_3.0.pdf, 2010 [cit. 2014-09-25].

[40] The ELRepo Project. Elrepo.org kernel-ml. http://elrepo.org/tiki/kernel-ml, 2014 [cit. 2015-03-29].

[41] Red Hat Inc. Red Hat Enterprise Linux 6.6 Beta Now Available for Testing. https://www.redhat.com/archives/rhelv6-announce/2014-August/msg00000.html, 2014 [cit. 2014-09-24].

[42] Red Hat Inc. Red hat enterprise linux 7.0 release notes. https://access.redhat.com/documentation/en-US/Red_Hat_Enterprise_Linux/7/html/7.0_Release_Notes/index.html, 2014 [cit. 2014-09-24].

[43] L. Rizzo. The netmap project. http://info.iet.unipi.it/~luigi/netmap/, 2012 [cit. 2014-07-15].

[44] R. Rosen. Linux Kernel Networking. http://www.haifux.org/lectures/172/netLec.pdf, 2007 [cit. 2014-09-02].

[45] R. Rosen. *Linux Kernel Networking - Implementation and Theory*. Apress, 2014. ISBN-978-1-4302-6197-1.

[46] S. Seeth and M. Venkatesulu. *TCP/IP Architecture, Design, and Implementation in Linux*. John Wiley and Sons, 2008. ISBN-978-0470-14773-3.

[47] S. Shankland. Linux development by the numbers: Big and getting bigger. http://www.cnet.com/news/linux-development-by-the-numbers-big-and-getting-bigger/, 2013 [cit. 2014-09-01].

[48] IEEE Computer Society. IEEE Std 802.3ae-2002. http://standards.ieee.org/getieee802/download/802.3ae-2002.pdf, 2002 [cit. 2014-09-16].

[49] IEEE Computer Society. IEEE Std 802.3an-2006. http://standards.ieee.org/getieee802/download/802.3an-2006.pdf, 2006 [cit. 2014-09-16].

[50] IEEE Computer Society. IEEE Std 802.3ba-2010. http://standards.ieee.org/getieee802/download/802.3ba-2010.pdf, 2010 [cit. 2014-09-03].

[51] IEEE Computer Society. IEEE P802.3bq 40GBASE-T Task Force. http://www.ieee802.org/3/bq/, 2013 [cit. 2014-09-23].

[52] IEEE Computer Society. IEEE P802.3bm 40 Gb/s and 100 Gb/s Fiber Optic Task Force. http://www.ieee802.org/3/bm/, 2014 [cit. 2014-09-23].

[53] Supermicro. X10DRU-i+ User's manual. https://www.supermicro.com/products/motherboard/Xeon/C600/X10DRU-i_.cfm, 2014 [cit. 2015-01-07].

[54] Mellanox Technologies. ConnectX-3 EN - Ethernet Adpater Cards Product Brief. http://www.mellanox.com/related-docs/prod_adapter_cards/PB_ConnectX3_EN_Card.pdf, 2014 [cit. 2014-09-03].

[55] Mellanox Technologies. MLNX_EN for Linux User Manual Rev 2.2-1.0.1. http://www.mellanox.com/related-docs/prod_software/Mellanox_EN_for_Linux_User_Manual_v2_2-1_0_1.pdf, 2014 [cit. 2014-09-03].

[56] Mellanox Technologies. ConnectX-3 EN. http://www.mellanox.com/related-docs/prod_silicon/ConnectX3_EN_Silicon.pdf, 2015 [cit. 2015-05-13].

[57] The CentOS Project. Centos 7.0.1406 release notes. http://wiki.centos.org/Manuals/ReleaseNotes/CentOS7, 2014 [cit. 2015-01-12].

[58] Network Security Toolkit. LAN Ethernet Maximum Rates, Generation, Capturing and Monitoring. http://wiki.networksecuritytoolkit.org/nstwiki/index.php/LAN_Ethernet_Maximum_Rates,_Generation,_Capturing_%26_Monitoring, 2011 [cit. 2015-02-03].

[59] Treyl. Anatomy of an Ethernet Frame. https://communities.netapp.com/blogs/ethernetstorageguy/2009/09/12/anatomy-of-an-ethernet-frame, 2009 [cit. 2014-09-23].

[60] H. Welte and P. Ayuso. The netfilter.org project. http://www.netfilter.org/, 2014 [cit. 2014-09-12].

[61] S. Woo and K. Park. Scalable tcp session monitoring with symmetric receive-side scaling. http://www.ndsl.kaist.edu/~kyoungsoo/papers/TR-symRSS.pdf, 2015 [cit. 2015-05-08].

Appendix A

Populating kernel's FIB with BGP routes

RIPE Network Coordination Centre provides BGP data on its website.[1] The BGP measurement presented in this thesis uses the data of RIPE NCC Amsterdam from 9th December 2014. The actual data can be obtained from the RIPE NCC's site.[2]

The BGP data is in MRT Routing Information Export Format, which is described in RFC 6396. The data can be obtained from the Quagga routing daemon using *dump bgp routes* command. RIPE NCC provides bgpdump utility to parse this file to a human-readable text file. To build bgpdump on CentOS 7 issue the following commands:

```
yum install bzip2-devel zlib-devel     # dependencies
wget http://www.ris.ripe.net/source/bgpdump/libbgpdump-1.4.99.13.tgz
tar xf libbgpdump-1.4.99.13.tgz
cd libbgpdump-1.4.99.13
./configure
make
make example      # needed prior make install
make install
```

Note that make example is required prior to make install. Afterwards, bgpdump command should be available (installed to /usr/local/bin/bgpdump). The Makefile also supports make rpm target to generate rpm package (a wrong date in specfile needs to be corrected).

The bgpdump utility can parse the downloaded latest-bview.gz file directly (even without extracting first). The following command extracts individual destination networks from the BGP data file and writes them to *destinations* file.

```
bgpdump latest-bview.gz | grep 'PREFIX' | grep '\.' | sed 's/PREFIX: \(.*\)
    /\1/' | uniq > destinations
```

The number of routes corresponds to the number of prefixes announced on the Internet. Note that the first destination entry is default route 0.0.0.0/0 and the file also contains subnets such as 10.0.0.0/8 or 192.168.0.0/16, which you may want to remove from the file prior to the next step. Basically, you want to remove all the local routes from the generated file. The local routes can be displayed using the following command.

```
ip route show table local | grep '^local'
```

[1] http://www.ripe.net/data-tools/stats/ris/ris-raw-data

[2] http://data.ris.ripe.net/rrc00/latest-bview.gz

If you are connected remotely, such as through SSH, you must also remove routes to your local PC, which would break the existing connection when misconfigured. In case of the provided server's connection, the *destinations* file must not contain 195.113.0.0/16 prefix The *destinations* file was obtained by running the following command:

```
bgpdump latest-bview.gz | grep 'PREFIX' | grep '\.' | sed 's/PREFIX: \(.*\)
    /\1/' | grep -v '195.113.0.0/16' uniq > destinations
```

The *destinations* file can be used to insert the router to the kernel's FIB table by ip route add command. To insert subnets from the *destinations* file to the kernel's FIB, the following C program was written. The neighbours of the local router, which are used to forward the packets, are specified in the *gateways* array. This array needs to be adjusted prior to executing. The destination subnets are then inserted to kernel's FIB via these neighbours. The program takes a single argument, path to the *destinations* file generated as described above. Note the *execlp()* function call on line 57, which performs the route insertions.

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <unistd.h>
4  #include <errno.h>
5  #include <sys/wait.h>
6
7  #if !defined(ARRAY_SIZE)
8      #define ARRAY_SIZE(x) (sizeof((x)) / sizeof((x)[0]))
9  #endif
10
11 /* Define your gateway IP addresses */
12 char *gateways[] = { "192.0.2.6" }; //, "193.160.39.1" }; // , etc. };
13
14 int main(int argc, char *argv[])
15 {
16   char *gw;
17   char *buf;
18   size_t size = 64;
19   int i = 0;
20
21   if (argc != 2)
22   {
23     fprintf(stderr, "Usage: %s dstfile\n", argv[0]);
24     return EXIT_FAILURE;
25   }
26
27   FILE *f = fopen(argv[1], "r");
28   if (f == NULL)
29   {
30     fprintf(stderr, "%s %d\n", argv[1], __LINE__);
31     return EXIT_FAILURE;
32   }
33
34   buf = malloc(64);
35   if (buf == NULL)
36   {
37     fprintf(stderr, "%s %d\n", argv[0], __LINE__);
38     fclose(f);
39     return EXIT_FAILURE;
40   }
41
```

```
42    ssize_t len;
43    while ((len = getline(&buf, &size, f)) > 5) // get destination subnets
44    {
45      buf[len-1] = '\0';
46      gw = gateways[i % ARRAY_SIZE(gateways)];
47      pid_t pid = fork();
48      if (pid == -1)
49      {
50        fprintf(stderr, "%s %d\n", argv[0], __LINE__);
51        fclose(f);
52        free(buf);
53        return EXIT_FAILURE;
54      }
55      else if (pid == 0) // child
56      {
57        printf("ip route add %s via %s\n", buf, gw);
58        execlp("ip", "ip", "route", "add", buf, "via", gw, NULL);
59      }
60      else // parent
61      {
62        wait(NULL); // wait for child
63      }
64      i++; // move to next gw
65    }
66    fclose(f);
67    free(buf);
68    return EXIT_SUCCESS;
69  }
```

The scripts and programs shown above can be found on the CD attached to this thesis.

Appendix B

Updating the Mellanox ConnectX-3 EN firmware

The firmware can be updated using the Mellanox Firmware Tools (MFT), which can be downloaded from the Mellanox management tools site.[1] Installation of Mellanox Firmware Tools is described in MTF User Manual available at the Mellanox documentation site.[2]

The installation consists of unpacking the downloaded MFT archive and running the *install.sh* script. This will install kernel-mft and mft packages. The kernel-mft package contains required low-level kernel drivers, whereas the mft package contains utilities which use these drivers. To start the Mellanox Software Tools service, which also loads the kernel drivers, run *mst start*. Afterwards, the utilities from the mft package can be used, including the mlxfwmanager for updating the firmware. The mlxfwmanager displays various information about the adapter, including the firmware version.

The newest firmware for the Mellanox ConnectX-3 adapters can be downloaded from the Mellanox firmware site.[3] Unzip the firmware and use the *mlxfwmanager -u* command in the same directory where the firmware was unzipped. This will update the firmware. After the update is complete, reboot is required to take effect.

Listing B.1 shows the output of the firmware update procedure using the *mlxfwmanager -u* command. The current firmware version is 2.31.5050, whereas the newer firmware placed in the current working directory is of version 2.32.5100. The mlxfwmanager_pci utility can be used to query the adapter information without starting the mst service.

[1] http://www.mellanox.com/page/management_tools
[2] http://www.mellanox.com/related-docs/MFT/MFT_user_manual.pdf
[3] http://www.mellanox.com/page/firmware_table_ConnectX3EN

```
[root@server]# mlxfwmanager -u
Querying Mellanox devices firmware ...

Device #1:
----------

  Device Type:      ConnectX3
  Part Number:      MCX314A-BCB_Ax
  Description:      ConnectX-3 EN network interface card; 40GigE; dual-port
     QSFP; PCIe3.0 x8 8GT/s; RoHS R6
  PSID:             MT_1090110023
  PCI Device Name:  /dev/mst/mt4099_pci_cr0
  Port1 MAC:        f452145e6c70
  Port2 MAC:        f452145e6c71
  Versions:         Current        Available
     FW             2.31.5050      2.32.5100
     PXE            3.4.0225       3.4.0306

  Status:           Update required

---------
Found 1 device(s) requiring firmware update...

Perform FW update? [y/N]: y
Device #1: Updating FW ... Done

Restart needed for updates to take effect.
```

Listing B.1: Firmware update procedure

Appendix C

General steps for maximum routing performance

- the networking code does not benefit from Hyper-Threading, disable it
- determine how many RX queues does the NIC support by „ethtool --show-channels *ifname*“ and use all of them using „ethtool --set-channels *ifname* rx *num*“, beware that RSS usually limits addressing IRQs to power of 2 CPUs
- disable *irqbalance* daemon and assign each IRQ exclusively to a single CPU via /proc/irq/*NUMBER*/smp_affinity_list or /proc/irq/*NUMBER*/smp_affinity
- set the number of TX queues the same as RX queues by „ethtool --set-channels *ifname* tx *num*“ and assign exclusive mapping of each queue to a single CPU
- disable all daemons such as avahi, dbus, NetworkManager, etc.
- check for disabled offloads by „ethtool --show-offload *ifname*“ and enable all of them
- disable SELinux in /etc/sysconfig/selinux (and reboot to apply)
- disable Netfilter if appropriate
- set *performance* scaling governor for all CPUs
- disable *rp_filter* on all forwarding interfaces by writing „0“ to /proc/sys/net/ipv4/conf/*ifname*/rp_filter, beware that this might be required to set at boot time using the /etc/sysctl.conf file (use *perf top* to check)
- avoid using DHCP and ARP/ND protocols, use static addressing and neighboring configuration instead