Università di Pisa and Scuola Superiore Sant'Anna

Master Degree in Computer Science and Networking

MASTER THESIS

Packet Filtering Module for PFQ Packet Capturing Engine

CANDIDATE

Venkatraman Gopalakrishnan

SUPERVISOR

Stefano Giordano

Gregorio Procissi

Nicola Bonelli

Academic Year 2010/2011

Table of Contents

Table of Contents

Table of Contents

Table of Contents

List of Figures

List of Figures

Abstract

The evolution of commodity hardware is pushing parallelism forward as the key factor that can allow software to attain hardware-class performance while still retaining its advantages. On one side, commodity CPUs are providing more and more cores (the next-generation Intel Xeon E 7500 CPUs will soon make 10 cores processors a commodity product), with a complex cache hierarchy which makes aware data placement crucial to good performance. On the other side, server NIC's are adapting to these new trends by increasing themselves their level of parallelism. While traditional 1Gbps NICs exchanged data with the CPU through a single ring of shared memory buffers, modern 10Gbps cards support multiple queues: multiple cores can therefore receive and transmit packets in parallel. In particular, incoming packets can be de-multiplexed across CPUs based on a hash function (the so-called RSS technology) or on the MAC address (the VMD-q technology, designed for servers hosting multiple virtual machines). The Linux kernel has recently begun to support these new technologies. Though there is lot of network monitoring software's, most of them have not yet been designed with high parallelism in mind. Therefore a novel packet capturing engine, named PFQ was designed, that allows efficient capturing and in-kernel aggregation, as well as connection-aware load balancing. Such an engine is based on a novel lockless queue and allows parallel packet capturing to let the user-space application arbitrarily define its degree of parallelism. Therefore, both legacy applications and natively parallel ones can benefit from such capturing engine. In addition, PFQ outperforms its competitors both in terms of captured packets and CPU consumption. In this thesis, a new packet filtering block is designed,

implemented and added to the existing PFQ capture engine which helps in dropping out unnecessary packets before they are copied into the kernel space thus improves the overall performance of the user space applications considerably. Because network monitors often want only a small subset of network traffic, a dramatic performance gain is realized by filtering out unwanted packets in interrupt context.

Chapter 1

Introduction to traffic monitoring on commodity hardware

Traffic monitoring is performed to collect data that describes the use and performance of the network. Network Traffic Monitoring is needed, not only to fix network problems on time, but also to prevent network failures, to detect inside and outside threats, and make good decisions for network planning. Network traffic monitoring helps us to avoid bandwidth and server performance bottlenecks, discover which applications use up your bandwidth, be proactive and deliver better quality of service to users, reduce costs by buying bandwidth and hardware according to actual load and easily troubleshoot network problems. Generally speaking, it is a necessary practice for the network administrators.

Traffic monitoring on commodity hardware is the process of gathering information out of packets received through one or more network interfaces. From a practical point of view, an application performing traffic monitoring on commodity hardware (say, PCs) must somehow to interface with lower level functionalities and, in particular, with the kernel of the operating system.

Commonly, monitoring applications run in the user space and can access the packets entering the system through the physical interfaces only with the support of kernel modules. The *system calls* provide that interface between any user space application and the kernel (as discussed in section 1.1). A *socket* API is an application programming interface (API), usually provided by the kernel, that allows application to control and use network sockets.

Within the kernel and the application that created a socket, the socket is referred to by a unique integer number called *socket identifier* or *socket number*. The kernel forwards the payload of incoming IP packets to the corresponding application by extracting the socket address information from the IP and transport protocol headers and stripping the headers from the application data.

1.1 User and Kernel Space

Before discussing about socket() library function which internally invokes sys_socket system call, understanding of user space and kernel space is required. Kernel space and user space is the separation of the privileged operating system functions and the restricted user applications. The separation is necessary to prevent user applications from ransacking our computer. Kernel space is strictly reserved for running the kernel, kernel extensions, and device drivers. In contrast, user space is the memory area where all user mode applications work and this memory can be swapped out when necessary. User space programs cannot access system resources directly so access is handled on the program's behalf by the operating system kernel. The user space programs typically make such requests of the operating system through system calls. System calls are requests in a Unix-like operating system by an active process for a service performed by the kernel, such as input/output (I/O) or process creation.

1.2 Linux Socket System Call

Socket is an abstraction that allows user processes to create endpoints for communication. Creating a socket from user space is done by socket() system call.

```
#include <sys/types.h>
#include <sys/socket.h>
int socket(int domain, int type, int protocol);
```

On success, a file descriptor for the new socket is returned which is used for communication from that point [1].

Parameter Description:

(1) The first parameter, domain, is also sometimes referred to as *family*. The domain argument specifies a communication domain; this selects the protocol family which will be used for communication. Most common domains are AF_INET for IPv4 protocol, AF_INET6 for IPv6 protocol and PF_PACKET for low level packet interface which will be seen in detail in the following section.

(2) The second parameter, *type*, specifies the communication semantics. Most commonly used types are SOCK_STREAM which provides sequenced, reliable, two-way, connection-based byte streams, SOCK_DGRAM which supports datagrams (connectionless, unreliable messages of a fixed maximum length) and SOCK_RAW which provides raw network protocol access. Some socket types may not be implemented by all protocol families. So understanding of which protocol goes with which type is important.

(3) The *protocol* parameter specifies a particular protocol to be used with the socket. Normally only a single protocol exists to support a particular socket type within a given protocol family, in which case protocol can be specified as 0. However, it is possible that multiple protocols may exist, in which case the particular protocol must be specified.

1.3 Software on Linux: the default PF_PACKET socket

As mentioned earlier, PF_PACKET is one specific communication domain for low level packet interface on device level which is of our interest here. Packet sockets are used to receive or send raw packets at the device driver (OSI Layer 2) level. They allow the user to implement protocol modules in user space on top of the physical layer. This family allows an application to send and receive packets dealing directly with the network card driver, thus avoiding the usual protocol stack-handling (e.g., IP/TCP or IP/UDP processing). That is, any packet sent through the socket will be directly passed to the Ethernet interface, and any packet received through the interface will be passed to the application [2].

```
#include <sys/socket.h>
 #include <netpacket/packet.h>
 #include <net/ethernet.h>     /* the L2 protocols */
 packet_socket = socket(PF_PACKET, int socket_type, int protocol);
```

The socket_type is either SOCK_RAW for raw packets including the link Level header or SOCK_DGRAM for cooked packets with the link level header removed. The link level header information is available in a common format in the sockaddr_ll structure. *Protocol* field is the IEEE 802.3 protocol number in network order. See the <linux/if_ether.h> include file for a list of allowed protocols. When protocol is set to htons(ETH_P_ALL) then all protocols are received. All incoming packets of that protocol type will be passed to the packet socket before they are passed to the protocols

implemented in the kernel. Only processes with effective uid 0 or the CAP_NET_RAW capability are allowed to open packet sockets.

SOCK_RAW packets are passed to and from the device driver without any changes in the packet data. When receiving a packet, the addresses are passed form/to the user space application through the sockaddr_ll address structure. When transmitting a packet, the user supplied buffer should contain the physical layer header. The packet is then queued unmodified to the network driver of the interface defined by the network interface specified in the sockaddr_ll data structure. Some device drivers always add other headers.

SOCK_DGRAM operates on a slightly higher level. The physical header is removed before the packet is passed to the user. Packets sent through a SOCK_DGRAM packet socket get a suitable physical layer header based on the information in the sockaddr_ll destination address before they are queued.

By default all packets of the specified protocol type are passed to a packet socket. To get packets only from a specific interface, bind it to the interface of interest by specifying an address in a struct sockaddr_ll. Only the sll_protocol and the sll_ifindex address fields are used for purposes of binding.

1.4 Configuring Commodity Hardware as a Monitoring tool with PF_PACKET

The PF_PACKET family allows an application to retrieve data packets as they are received at the network card level, but still does not allow it to read packets that are not addressed to its host. As we have seen before, this

is due to the network card discarding all the packets that do not contain its own MAC address—an operation mode called non-promiscuous, which basically means that each network card is minding its own business and reading only the frames directed to it. There are three exceptions to this rule: a frame whose destination MAC address is the special broadcast address (FF: FF: FF: FF: FF: FF) will be picked up by any card; a frame whose destination MAC address is a multicast address will be picked up by cards that have multicast reception enabled, and have subscribed to a particular multicast group and a card that has been set in promiscuous mode will pick up all the packets it sees. To set a network card to promiscuous mode, all we have to do is issue a particular ioctl() call. Since this is a potentially security-threatening operation, the call is only allowed for the root user [3].

When executed as a root user with PC connected to LAN, you will be able to see all the packets flowing on the cable, even if they are not sent to your host. This is because your network card is working in promiscuous mode. This has laid the environment setup for network monitoring. If the number of nodes in the LAN grows, then the resulting traffic of the network also increases. Then follows the problem. The sniffer will start losing packets, since the PC will not be able to process them quickly enough.

```
struct ifreq ethreq;
int sock;
  if ( (sock=socket(PF_PACKET, SOCK_RAW, hons(ETH_P_IP)))<0) {
        perror("socket");
        exit(1);
  }
```

```
 /* Set the network card in promiscuos mode */
strncpy(ethreq.ifr_name,"eth0",IFNAMSIZ);
if (ioctl(sock,SIOCGIFFLAGS,&ethreq)==-1) {
      perror("ioctl");
      close(sock);
      exit(1);
}
ethreq.ifr_flags |=IFF_PROMISC;
if (ioctl(sock,SIOCSIFFLAGS,&ethreq)==-1) {
     perror("ioctl");
     close(sock);
     exit(1);
}
```

The solution to this problem is to filter out non interesting packets, and process out information only on those you are interested in. One idea would be to insert an “if statement” in the sniffer's source; this would help polish the output of the sniffer, but it would not be very efficient in terms of performance. The kernel would still pull up all the packets flowing on the network, thus wasting processing time, and the sniffer would still examine each packet header to decide whether to process out the related data or not.

The optimal solution to this problem is to put the filter as early as possible in the packet-processing chain [Figure 1]. The Linux kernel allows us to put a filter, called an LPF, directly inside the PF_PACKET protocol-processing routines, which are run shortly after the network card reception interrupt has been served. The filter decides which packets shall be relayed to the application and which ones should be discarded.

In order to be as flexible as possible, and not to limit the programmer to a set of predefined conditions, the packet-filtering engine is actually implemented as a state machine running a user-defined program. The program is written in a specific pseudo-machine code language called BPF (for Berkeley packet filter). BPF actually looks like a real assembly language with a couple of registers and a few instructions to load and store values, perform arithmetic operations and conditionally branch.

The filter code is run on each packet to be examined, and the memory space into which the BPF processor operates are the bytes containing the packet data. The result of the filter is an integer number that specifies how many bytes of the packet (if any) the socket should pass to the application level. This is a further advantage, since often you are interested in just the first few bytes of a packet, and you can spare processing time by avoiding copying the excess ones.

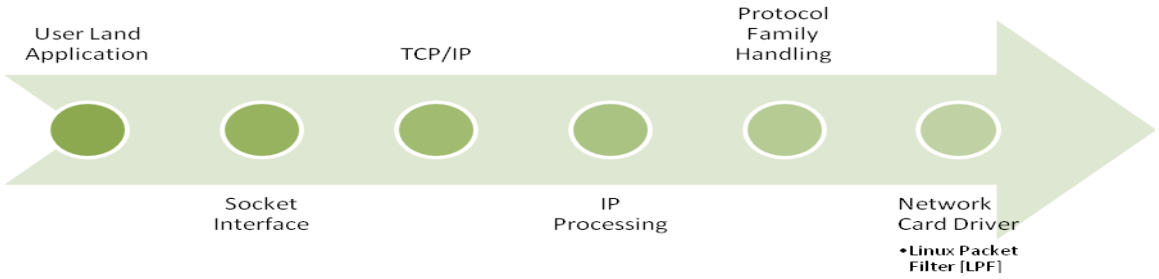

Figure 1: Packet Processing Chain

1.5 Pcap Library

Pcap (packet capture) consists of an application programming interface (API) for capturing network traffic. All packets on the network, even those destined for other hosts, are accessible through this mechanism. Unix-like systems implement Pcap in the Libpcap library; Windows uses a port of Libpcap known as WinPcap [5].

Libpcap is a system-independent interface for user-level packet capture. Libpcap provides a portable framework for low-level network monitoring. In Linux, Libpcap uses PF_PACKET sockets.

Applications such as Intrusion detection systems, Security Monitoring tools, packet analyzers, Network Debuggers, Network Statistics Collection (example) tcpdump, snort, wireshark, Nmap uses BPF, in the form of libpcap, to capture and filter packets. Even if the BPF language is pretty simple and easy to learn, most of us would probably be more comfortable with filters written in human-readable expressions. Libpcap helps us to avoid learning BPF language.

The Libpcap library is an OS-independent wrapper for the BPF engine [6]. When used on Linux machines, BPF functions are carried out by the Linux packet filter (LPF). One of the most useful functions provided by the libpcap is pcap_compile(), which takes a string containing a logic expression as input and outputs the BPF filter code. Tcpdump uses this function to translate the command-line expression passed by the user into a working BPF filter. The filter code is however not always optimized, since it is generated for a generic BPF machine and not tailored to the specific architecture that runs the filter engine.

Depending on the operating system, libpcap implements a virtual device from which captured packets are read from userspace applications. Despite different platforms provide the very same API; the libpcap performance varies significantly according to the platform being used. On low traffic conditions there is no big difference among the various platforms as all the packets are captured, whereas at high speed the situation changes significantly.

The following table reported in a study of Luca Deri in [7], shows the outcome of some tests performed using a traffic generator on a fast host (Dual 1.8 GHz Athlon, 3Com 3c59x Ethernet card) that sends packets to a mid-range PC (VIA C3 533 MHz2, Intel 100Mbit Ethernet card) connected over a 100 Mbit Ethernet switch (Cisco Catalyst 3548 XL) that is used to count the real number of packets sent/received by the hosts [7].

It is found that

(1) At 100 Mbit using a low-end PC, the simplest packet capture application is not able to capture everything (i.e. there is packet loss).

(2) Linux, a very popular OS used for running network appliances, performs very poorly with respect to other OSs used in the same test.

Traffic Capture Application	**Linux 2.4.x**	**FreeBSD 4.8**	**Windows 2k**
Standard Libpcap	0.2%	34%	68%
Mmap Libpcap	1%		

Kernel Module	4%		

Table 1: Percentage of Captured Packets (generated by tcpreplay)

(3) *Libpcap-mmap* [libpcap-mmap], a special version of libpcap, which is now the current implementation, exploits the mmap() system call for passing packets to user space, does improve the performance, but not significantly.

This means that Linux spends most of its time moving packets from the network card to the kernel and very little from kernel to userspace. The reason is because of the occurrence of *interrupt livelock*. Device drivers instrument network cards to generate an interrupt whenever the card needs attention (e.g. for informing the operating system that there is an incoming packet to handle, or when a packet transmission acknowledgment is received or when transmission error occurs). In case of high traffic rate, the operating system spends most of its time handling interrupts leaving little time for other tasks. A solution to this problem is something called *device polling.*

1.6 Understanding of NAPI

NAPI ("New API") is a modification to the device driver packet processing framework, which is designed to improve the performance of high-speed networking.

The only hardware requirement is that an interface is able to own DMA hardware [4]. However, in order to accommodate devices not capable of DMA, the old interface is still available for drivers. A new API is added to the driver interface.

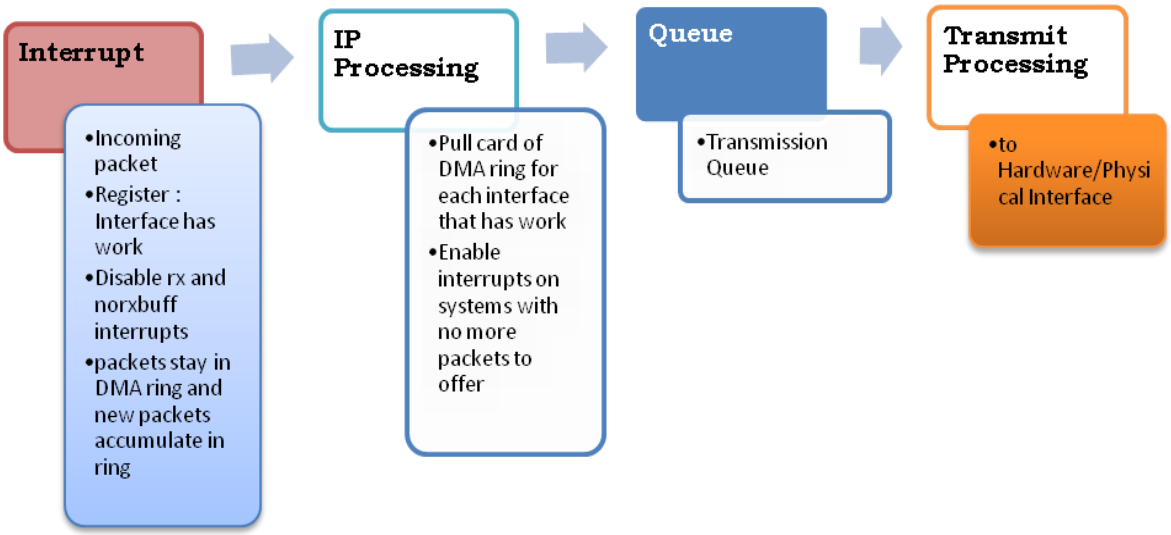

Figure 2: The NAPI Context

NAPI works through:

(A) **Interrupt mitigation**

High-speed networking can create thousands of interrupts per second, all of which tell the system something it already knew: it has lots of packets to process. NAPI allows drivers to run with (some) interrupts disabled during times of high traffic, with a corresponding decrease in system load. Polling is an alternative to interrupt-based processing. The kernel can periodically check for the arrival of incoming network packets without being interrupted, which eliminates the overhead of interrupt processing. Establishing an optimal polling frequency is important, however. Too frequent polling wastes CPU resources by repeatedly checking for incoming packets that have not yet arrived. On the other hand, polling

too infrequently introduces latency by reducing system reactivity to incoming packets, and it may result in the loss of packets if the incoming packet buffer fills up before being processed. Given the same workload (i.e., number of frames per second), the load on the CPU is lower with NAPI. This is especially true at high workloads.

(B) **Packet throttling**

When the system is overwhelmed and must drop packets, it's better if those packets are disposed of before much effort goes into processing them. NAPI-compliant drivers can often cause packets to be dropped in the network adaptor itself, before the kernel sees them at all.

(C) More careful packet treatment, with special care taken to avoid reordering packets. Out-of-order packets can be a significant performance bottleneck.

(D) Balance between latency and throughput

(E) Is independent from any hardware specifics

1.7 Disadvantages of NAPI

(1) In some cases, NAPI may introduce additional software IRQ latency.

(2) On some devices, changing the IRQ mask may be a slow operation, or require additional locking. This overhead may negate any performance benefits observed with NAPI

1.8 Linux Kernel and the new NAPI context

Linux Kernel and the new NAPI context packet processing mainly differ in the reception and the way of handling of the packets received. The following steps are followed when a packet is received:

(a) Packets are first received by the card. They are put in the rx_ring using DMA for recent cards. The size of the ring is hardware dependent. Older cards, which do not support DMA, use the PIO scheme: it is the host CPU which transfers the data from the card into the host memory;

(b) The card interrupts the CPU, which then jumps to the driver ISR(Interrupt Service Routine) code. Here arise some differences between the old subsystem and NAPI.

For the older subsystem, the interrupt handler calls the netif_rx() kernel procedure. netif_rx() enqueues the received packet in the interrupted CPU's backlog queue and schedules a softirq, responsible for further processing of the packet (e.g.TCP/IP processing). The backlog size can be specified in /proc/sys/net/core/netdev_max_backlog. When it is full, it enters the throttle state and waits for being totally empty to reenter a normal state and allow again an enqueue by calling netif_rx(). If the backlog is in the throttle state, netif_rx drops the packet. Backlog stats are available in /proc/net/softnet_stats: one line per CPU, the first two columns are

packets and drops counts. The third is the number of times the backlog entered the throttle state.

NAPI drivers act differently: the interrupt handler calls netif_rx_schedule(). Instead of putting the packets in the backlog, it puts a reference to the device in a queue attached to the interrupted CPU. A softirq is schedules too, just like in the previous case. To insure backward compatibility, the backlog is considered as a device in NAPI, which can be enqueued just as an another card, to handle all the incoming packets. netif_rx() is rewritten to enqueue the backlog into the poll_list of the CPU after having enqueued the packet in the backlog;

(c) When the softirq is scheduled, it executes net_rx_action(). Since the previous step differs between the older network subsystem and NAPI, this one does too. For older versions, net_rx_action pulls all the packets in the backlog and calls for each of them the ip_rcv() procedure or another one depending on the type of the packet: arp, bootp, etc. For NAPI, the CPU polls the devices present in his poll_list to get all the received packets from their rx_ring or from the backlog. The poll method of the backlog or of any device calls, for each received packet, netif_receive_skb() which roughly calls ip_rcv().

1.9 Packet Capture Performance (Polling Vs No polling)

It can be noticed from the graph that, as long as the system has enough CPU cycles to handle all the traffic, there is not much difference between the different setups [7]. However for non-polling systems there is a maximum full-capture speed after which the system spends most of the available cycles to handle interrupts leaving little time to other tasks, hence the packet loss.

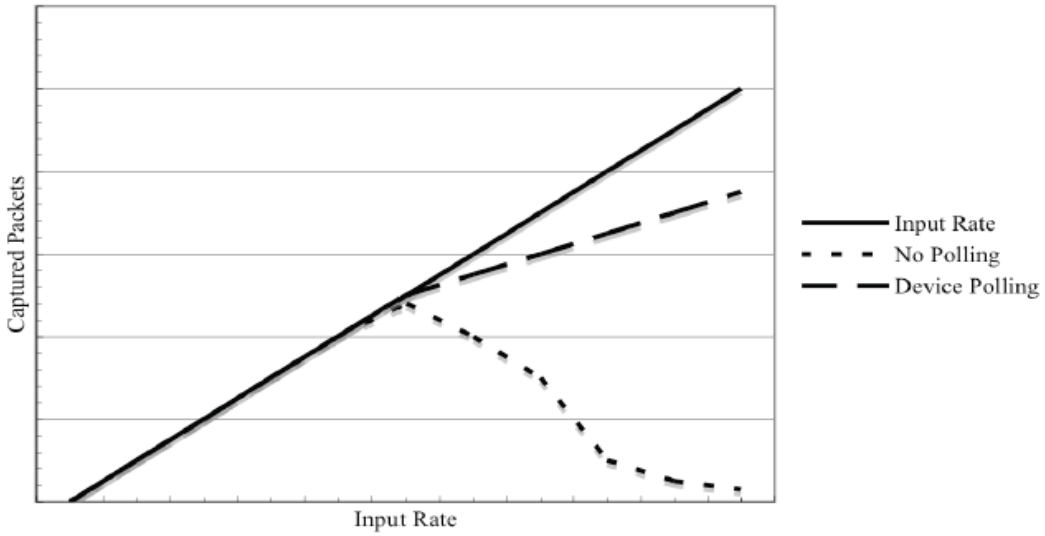

Figure 3: Packet Capture Performance

Source: L.Deri, Improving Passive Packet Capture: Beyond Device Polling

1.10 Beyond Device Polling

Device Polling is not the ultimate solutions. The packet capturing engines has to be designed with further considerations

(1) Design a solution for improving packet capture performance that is general and not locked to a specific driver or operating system architecture

(2) Device polling proved to be very effective; hence (if available) it should be exploited to improve the overall performance.

(3) For performance reasons, it is necessary to avoid passing incoming packets to the kernel that will pass then to userspace. Instead a straight path from the adapter to the user space needs to be identified in order to avoid the kernel overhead.

1.11 PF_RING

PF_RING is a type of network socket that improves the packet capture speed. PF_RING can be used with vanilla kernels (i.e. no kernel patch required); PF_RING is device driver independent, Kernel-based packet capture and sampling modules, which can be used for effective content inspection, so that only packets matching the payload filter are passed [8]. It also provides an ability to work in transparent mode in which the packets are forwarded to upper layers and the applications will work as usual. PF_RING polls packets from NICs by means of Linux NAPI. This means that NAPI copies packets from the NIC to the PF_RING circular buffer, and then the user space application reads packets from ring.

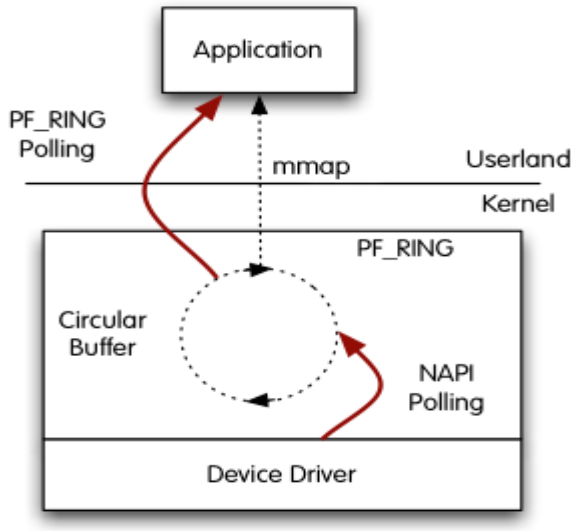

Figure 4: Vanilla PF_RING

Source: http://www.ntop.org/products/pf_ring/

PF_RING has three operational modes. Transparent mode 0 is the standard NAPI polling. Transparent mode 1, in which PF_RING-aware driver

copies the packets into PF_RING, while the same packet is passed to Linux kernel and Transparent mode 2, in which PF_RING aware driver copies the packets into PF_RING. Note that transparent mode 1 and 2 are meaningless on non PF_RING-aware drivers.

Advantage in this scenario is that PF_RING can distribute incoming packets to multiple rings (hence multiple applications) simultaneously. In this scenario, the drawback is that there are two poller's, both the application and NAPI and this results in CPU cycles used for this polling resulting in performance degradation.

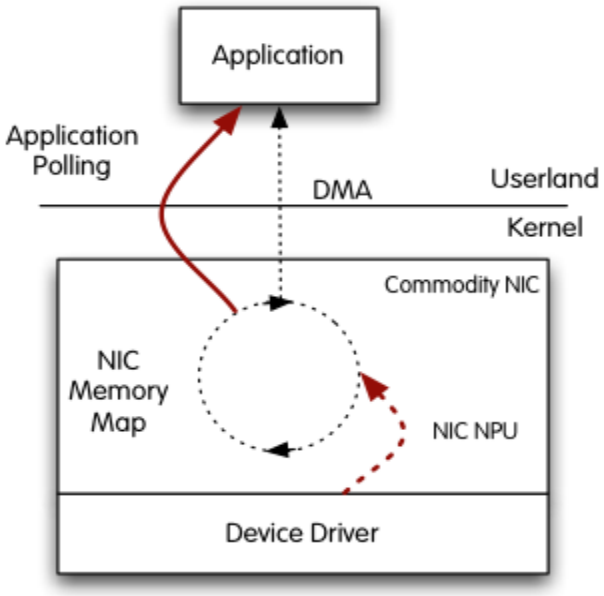

Figure 5: PF_RING with DNA driver

Source: http://www.ntop.org/products/pf_ring/

PF_RING DNA (Direct NIC Access) is a recent mechanism added to some specific device drivers that map NIC memory and registers to user space so

that packet copy from the NIC to the DMA ring is done by the NIC NPU (Network Process Unit) and not by NAPI [8]. This results in better performance as CPU cycles are used uniquely for consuming packets and not for moving them off the adapter.

The drawback is that only one application at time can open the DMA ring (note that modern NICs can have multiple RX/TX queues thus you can start simultaneously one application per queue), or in other words that applications in user space need to talk each other in order to distribute packets.

The main limitations are that, as NAPI polling does not take place in DNA mode, packet filtering is not supported as well as multi-core architecture is not completely exploited.

Chapter 2

Multi-core Architecture and 10G NICs with Multiple Hardware Queue support (MSI-X)

2.1 Introduction

Network speeds have been increasing at an incredible rate, doubling every 3-12 months. This is even faster than one version of "Moore's Law" which states that processing power doubles every 18-24 months; an observation made in 1965 which is still remarkably accurate. Moreover, although disk sizes have been increasing dramatically, disk and bus bandwidth have been increasing almost linearly; much more slowly than the exponential growth in other areas. In short, the disparity between network, CPU, and disk speeds will continue to increase problematically.

2.2 Today's Multi-core Architecture

Today's COTS hardware offers features and performance that just a few years ago were only provided by expensive special purpose hardware. The desktop machines are becoming advanced multi-core or even multi-processor parallel architectures capable of concurrently execute multiple threads at the same time. Modern network adapters feature several independent transmission (TX) and reception (RX) queues, each mapped on a separate core. Initially designed for facilitating the implementation of virtual machine supervisors, network queues can also be used to accelerate network traffic tasks, such as routing by processing incoming packets into concurrent threads of execution.

Although operating systems were adapted a long time ago to support multi-processing, kernel network layers have not yet taken advantage of this new technology. The result is that packet capture, the cornerstone of every network monitoring application is not able to capitalize of these breakthrough network technologies for traffic analysis, thus dramatically limiting its scope of application.

2.3 Load Balancing Technologies

2.3.1 Capture accelerators

Capture accelerators based on FPGA, implement filtering mechanisms at the network layer by means of rule sets (usually limited to 32 or 64) similar to BPF. Filtering runs at wire-speed. As the rule set is not meant to be changed at runtime, its scope of application is drastically limited [9]. Often traffic filtering is used to mark packets and balance them across DMA engines. Capture accelerators supporting multiple ring buffers implement in firmware the logic for balancing the traffic according to traffic rules, so that different processes or threads receive and analyze a portion of the traffic.

2.3.2 Receive Side Scaling

Intel's Receive-side scaling (RSS) is a network driver technology that enables the efficient distribution of network receive processing across multiple CPUs in multiprocessor systems. RSS enabled network adapters include the logic for balancing incoming traffic across multiple RX queues [9]. This balancing policy is implemented in hardware and thus RSS is not as flexible as rule based systems. However, this simple policy is effective in

practice also for network monitoring applications, as most of them though not bi-directional are flow-oriented.

2.4 Challenges in exploiting the Parallel Architectures

Primary concern in designing an optimized monitoring application lies in overcoming the challenges introduced by the parallel architecture under consideration.

2.4.1 General Challenges

(1) Packet capture applications are memory bound, but memory bandwidth does not seem to increase as fast as the number of core available.

(2) Balancing the traffic among different processing units is challenging, as it is not possible to predict the nature of the incoming traffic.

(3) Exploiting the parallelism with general-purpose operating systems is even more difficult as they have not been designed for accelerating packet capture.

(4) The operating system scheduler is completely unaware of the workload and in some cases it does not have the knowledge to relocate threads on the right core/processors.

(5) Balancing the workload among processors is not straightforward, as the workload depends on the incoming traffic, which cannot be predicted.

(6) Preserving the cache locality is a prerequisite in order to achieve a good scalability on modern parallel architectures and to overcome the bandwidth limitations.

2.5 Pitfalls in Monitoring Application Architecture

Though the commodity CPUs are providing more and more cores and the modern NICs are supporting multiple hardware queues that allow cores to fetch packets concurrently (in particular, this technology is known as *Receive Side Scaling*, henceforward RSS), the current network monitoring and security software is not yet able to completely leverage the potential which is brought on by the hardware evolution: even if progress is actually being made (multiple queue support has been included in the latest releases of the Linux kernel), much of current monitoring software has been designed in the pre multi-core era.

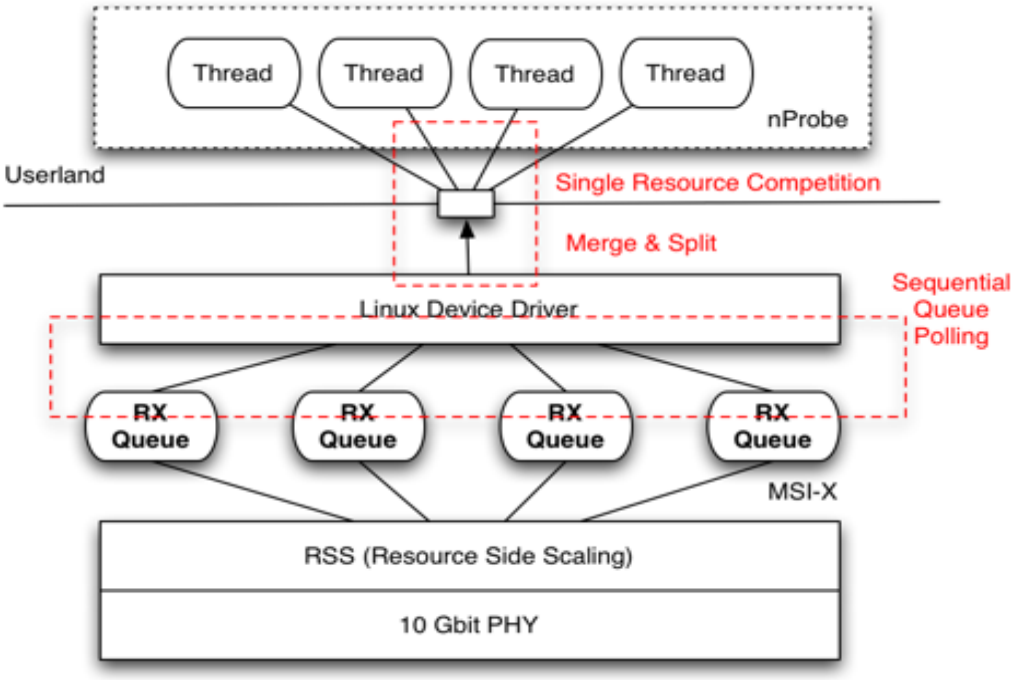

Figure 6: Design Limitation in existing Networking Monitoring Architecture

Source: http://www.ntop.org/products/pf_ring/tnapi/

Though modern network adapters are trying to improve network performance by splitting a single RX queue into several queues, each mapped to a processor core and also to balance the load, both in terms of packets and interrupts, across all cores hence to improve the overall performance, the device drivers are unable to preserve this design up to the application: they merge all queues into one as it used to happen with legacy adapters featuring only one queue [10]. This limitation is a major performance bottleneck, because even if a userland application uses several threads to consume packets, they all have to compete for receiving packets from the same socket resulting in *Single Resource Competition.*

Competition is costly as semaphores or similar techniques have to be used in order to serialize this work instead of carrying it out in parallel, as happens at the kernel level. In multi-core systems, this problem is even worse because it is not often possible to map the monitoring application on the same core from which packets are coming. In addition the use of semaphores that, as a side effect, invalidates the processor's cache, which represents the basic ingredient for preserving multi-core performance. In a nutshell, current network layer design needs to "merge and split" packets a couple of times and access them using semaphores, instead of providing a straight, lock-less path to the application with no performance limitation due to cache invalidation.

2.6 TNAPI

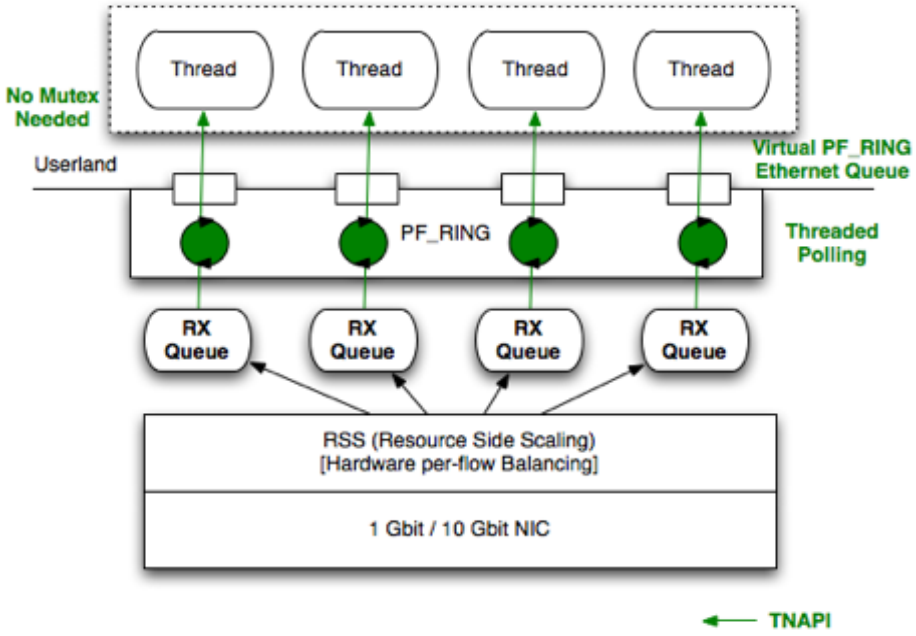

Figure 7: Multi-queue aware packet capture design

Source: http://www.ntop.org/products/pf_ring/tnapi/

TNAPI overcomes the problems imposed by the normal monitoring architecture like

- Distribute the traffic across cores for improving scalability.
- Poll packets simultaneously from each RX queue for fetching packets as fast as possible hence improve performance.
- Through PF_RING, expose the RX queues to the user space so that the application can spawn one thread per queue hence avoid using semaphores at all.

TNAPI achieves all this by starting one thread per RX queue. Received packets are then pushed to PF_RING (if available) or through the standard Linux stack. However in order to fully exploit this technology it is necessary to use PF_RING as it provides a straight packet path from kernel to user space [10].

Though TNAPI solves the problems faced by the existing monitoring modules, it is based on a heavily customized driver, which detaches parallel polling threads instead of relying on NAPI. Besides, its heavy use of kernel level polling leads to high CPU utilization.

Chapter 3

PFQ kernel module on multi-core architecture

3.1 Drawbacks of existing Monitoring Applications

Though the solutions proposed above have many advantages they all have some drawbacks in common (i.e.) either they don't exploit the parallel architecture properly or significant modification in existing device driver required and doesn't have beneficial improvement in case of vanilla drivers or High CPU Utilization.

3.2 PFQ

To overcome the disabilities encountered by existing packet capturing engines, a novel packet capturing engine, named PFQ is designed to allow efficient capturing and in-kernel aggregation, as well as connection-aware load balancing [11]. This engine is based on a novel lockless queue and allows parallel packet capturing to let the user-space application arbitrarily define its degree of parallelism. Therefore, both legacy applications and natively parallel ones can benefit from such capturing engine. PFQ outperforms its competitors both in terms of captured packets and CPU consumption.

PFQ is designed with an aim that allows parallelizing the packet capturing process in the kernel and, at the same time, to split and balance the captured packets across a user-defined set of capturing sockets. PFQ allows application writers to arbitrarily choose its level of parallelism, hiding within the kernel the full parallelism of the system. In particular, an

application can either use a single capturing socket (as in the case of legacy applications) or have PFQ balance incoming frames across a configurable set of collection points (sockets) or even use a completely parallel setup, where packets follow parallel paths from the device driver up to the application. In all of those cases, PFQ yields better performance than its competitors, while burning a lower amount of CPU cycles. Differently from many existing works for accelerating software packet processing, PFQ does not require driver modification (although a minimal few lines patch in the driver can further improve performance). Scalability is achieved through batch processing (which, in turn, leverages the hierarchical cache structure of modern CPUs) and through lockless techniques, which allow multiple threads to update the same state with no locking and minimal overhead. In particular, we designed a novel double buffer multi-producer single-consumer lockless queue which allows high scalability.

3.3 PFQ Packet Capturing Engine

PFQ Engine is made up of the following components: the packet fetcher, the de-multiplexing block and socket queues [11]. The fetcher dequeues the packet directly from the driver, which can be a standard driver or a patched aware driver, and inserts it into the batching queue. The next stage is the de-multiplexing block, which is in charge of selecting which socket(s) need to receive the packet. The final component of PFQ is the socket queue, which represents the interface between user space and kernel space. All of the kernel processing (i.e.) from the reception of the packet up to its copy into the socket queue is carried out within the NAPI

context; the last processing stage is completely performed at user space, thanks to memory mapping.

3.3.1 Driver Awareness

PFQ embodies a patched version of the ixgbe driver that just involves minimal code modifications (around a dozen lines of code); such a simple patch can be easily applied to new and existing drivers. This block is completely optional as PFQ shows good performance with vanilla drivers too.

This driver directly forwards the packet to the capturing module instead of passing it to the standard Linux networking stack thus improving performance. On the other hand, the capturing module has exclusive ownership of the packet, which is invisible to the rest of the kernel when at least one PFQ socket is open for monitoring a given device. Otherwise, the packet is forwarded to the Linux kernel and receives the classical default processing.

3.3.2 Packet Fetcher

The packet fetcher acts on every packet that is received. It receives the packets and inserts the associated pointer into its *batching queue*. Once such a queue (whose length is configurable) is filled, all of its enqueued packets are processed by the next block in a single batch [11]. Batch processing turns out to be more efficient in that, it improves the temporal locality of memory accesses, thus reducing the probability of both cache misses and concurrent access to shared data. In particular, a significant advantage comes from de-allocating packets in batches that, according to our measurements, can reduce the de-allocation cost by as much as 75%.

Notice that, as the packet is time stamped before queuing, this component does not influence timing accuracy.

3.3.3 Packet Steering Block

The steering block selects which sockets need to receive the captured packets. Although this is a single functional block, the steering processing is completely distributed and does not represent a serialization point. It uses a routing matrix to flexibly dispatch the incoming packets across multiple capturing sockets [11].

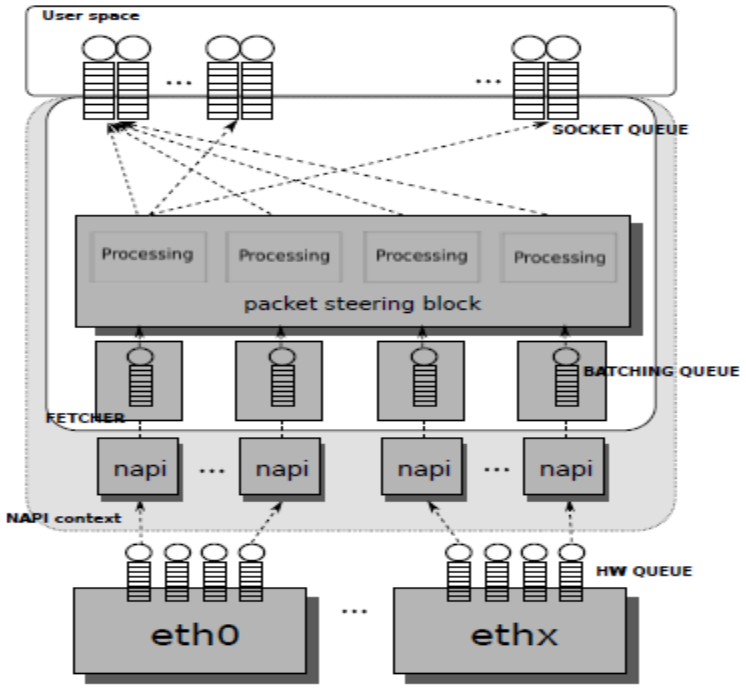

Figure 8: PFQ Architecture

Source: PFQ: a Novel Architecture for Packet Capturing On Parallel Commodity Hardware

In particular, such a matrix associates each reception queue of each handled card with one or more capturing sockets. Such sockets can be independent from each other (thus receiving one copy of the packet each) or can be aggregated into a load balancing group. In this latter case, a hash function is computed for each packet and only one socket in the balancing group is chosen. An additional advantage of such approach is the possibility of performing a bidirectional load balancing. As described in the previous section, RSS (Receive side Scaling) performs its native form of load balancing by computing a hash function over the 5-tuple of incoming packets. However, such scheme may not be appropriate for some applications, as RSS is not symmetric. For example, applications that monitor TCP connections need to observe packets from both directions which RSS would dispatch to different cores. For this reason, the packet steering block re-computes a symmetric hash function that will rebalance the packets with small overhead. Notice that load balancing and copy are not mutually exclusive: packets from the same hardware queue can be copied to a set of sockets and load-balanced across a different group. In greater detail, the de-multiplexing block is composed by a bit -field matrix and a load balancing function. The switching matrix stores, for each queue, a bitmap specifying which sockets have to receive its packets. Such a design allows dynamic insertion and removal of sockets with no need for mutexes on the fast data path allowing a great performance optimization. No serialization of threads of executions running on different cores is required.

3.3.4 Multiple Provider Double Buffer(MPDB) Socket queue

It is the last component of our architecture and the only one which is subject to inter-core contention. Our design shares some similarities with that of the FreeBSD zero-copy packet filter, but it improves the state of the art by introducing a wait-free solution which is optimized for a multi-core environment. Indeed, the whole mechanism implements a multiple producer - single consumer wait-free queue. The main components of this block are two memory mapped buffers: while one of them is being filled with the packets coming from the de-multiplexer, the other one is being read from the user application. The two buffers are periodically swapped through a memory mapped variable that stores both the index of the queue being written to and the number of bytes that have been already inserted (in particular, its most significant bit represents the queue index). Each producer (i.e. a NAPI kernel thread) reserves a portion of the buffer by atomically incrementing the shared index; such reservation can be made on a packet by packet basis or once for a batch. After the thread has been granted exclusive ownership of its buffer range, it will fill it with the captured packet along with a short pseudo header containing meta-data (e.g. the timestamp). Finally, it will finalize it by setting a validation bit in the pseudo header after raising a write memory barrier. Notice that, when the user application copies the packets to a user space buffer, some NAPI contexts may still be writing into the queue. This will results in some of the slots being half filled when they reach the application; however, the user-space thread can wait for the validation bit to be set. On the application side, the user thread which needs to read the buffer will first reset the index by specifying another active queue (so as to direct all subsequent writes to

it). Subsequently, it will copy to the application buffer a number of bytes corresponding to the value shown by the old index. Such copy will be performed in a single batch, as, from our past measurements, batch copy can be up to 30% faster. Alternatively, packets can be read in place in a zero-copy fashion.

3.4 PFQ Performance Analysis

In [11], performance of PFQ system was assessed under several configurations and it was mainly compared against PF RING because it was the obvious competitor for PFQ, in that it is a general architecture that increases the capturing performance with both vanilla and modified drivers. Two main performance metrics were taken into consideration: number of captured packets and average CPU consumption.

3.4.1 Experimental Setup

The testbed for experiments consists of two identical machines, one for generating traffic, and the other in charge of capturing. Both of them come with a 6 cores Intel X5650 Xeon (2.66 Ghz clock, 12Mb cache), 12 GB of DDR3 RAM, and an Intel E10G42BT NIC, with the 82599 controller on board. In order to test our system with the maximum degree of parallelism, we kept Intel Hyperthreading enabled, thus carrying out the experiments with 12 virtual cores. Due to the high cost of hardware based traffic generators and to the limited performance of software based ones, self written generators were used. It was able to generate up to 12 Millions minimum-sized packets per second. Lets us consider only the parallel setup and for any further reading refer to [11].

3.4.2 Parallel Setup

In this scenario each hardware queue is associated with its own user space thread, so that the processing paths of packets are completely parallel. Notice that in this scenario recently introduced quick mode option of PF_RING is used, which allows avoiding per-queue locks.

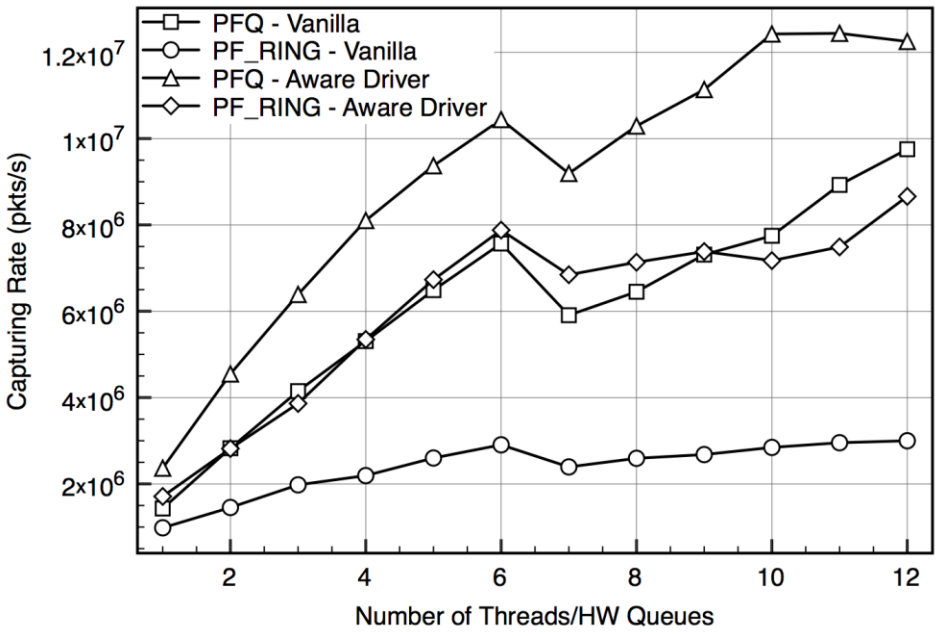

Figure 9: Complete parallel processing paths

The results shown in figure 10 show that, although PF_RING manages to achieve good performance by preventing locking, PFQ still outperforms it. Besides, PFQ shows the same behavior with both vanilla and aware drivers (apart from a scale factor), while PF_RING only scales well with aware drivers.

Notice that PFQ is able to capture all of the incoming packets with 10 cores (its throughput steadies because there is no additional traffic to capture); unfortunately, our generator is not able to produce more input traffic and, therefore, we can only obtain a lower bound of PFQ's performance.

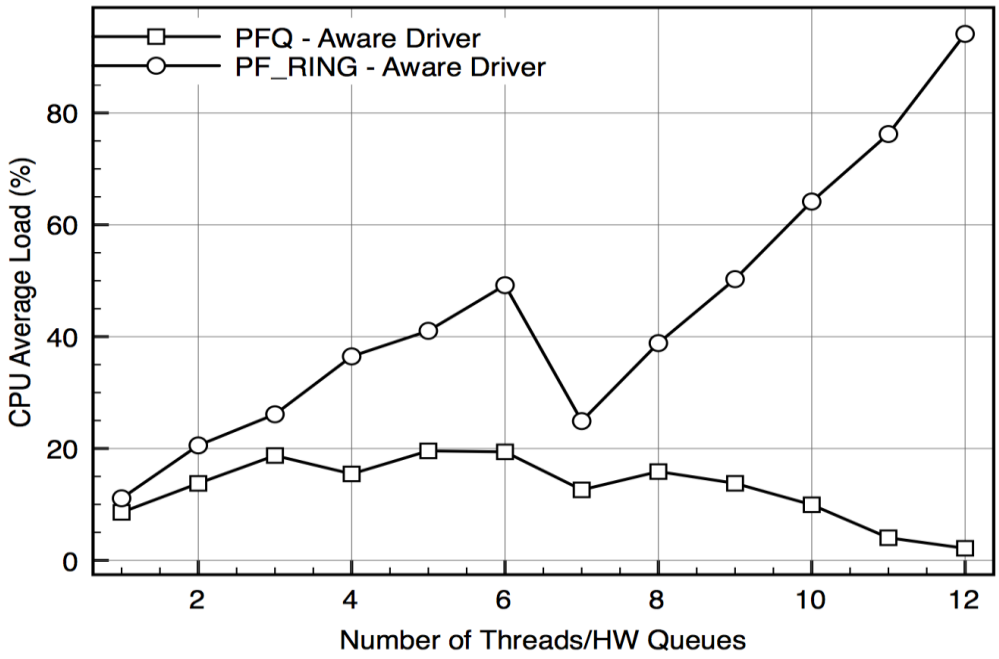

Figure 10: Complete parallel processing paths (CPU Consumptions)

From figure 11, we can understand the CPU utilization (in the case of aware drivers): while PF_ RING saturates the CPU, the global CPU consumption in the case of PFQ is roughly constant and well below 20%.

Chapter 4

PFQ Packet Filter Design and Implementation

4.1 What is Packet Filter?

When a packet reaches the interface, the node decides whether the packet belongs to it or it should be dropped in case it is not destined for it based on packet's header information. This is the basic routing principle. When packet filtering is added to routing devices, another level of packet analysis is done. Each packet will be put through the normal routing analysis and when determined that it has to be processed, the routing device applies the filter rules. Filter rules normally reflect security policies, which services are allowed, where they are allowed, which are not, which types of packets can reach a target device, which should be dropped, etc. So, packet filtering is the process of passing or blocking data packets, based on a set of user-provided rules, as they pass through a network interface.

In recent years, with dramatically increasing network speed and escalating protocol complexity, packet filters have been facing intensified challenges posed by more dynamic filtering tasks and faster filtering requirements. However, existing packet filter systems have not yet fully addressed these challenges in an efficient and secure manner.

Packet filtering can also be defined as the selective passing or blocking of data packets as they pass through a network interface. Filter rules specify the criteria that a packet must match and the resulting action, either block or pass, that is taken when a match is found. Filter rules are evaluated in sequential order from first to last. Packet Inspection criteria's

are generally based on the Layer 3 and Layer 4 headers. The most often used criteria are source and destination address, source and destination port, and protocol(the so-called *canonical 5-tuple*).

Many versions of UNIX provide facilities for user-level packet capture, making possible the use of general purpose workstations for network monitoring. Because network monitors run as user-level processes, packets must be copied across the kernel/user-space protection boundary. This copying can be minimized by deploying a *packet filter*, which discards unwanted packets as early as possible. A packet filter can be implemented as a single predicate, that is a function returning a Boolean value on the basis of a set of filter rules when applied to a given packet. If the value of the function is *true* the kernel copies the packet for the application; if it is *false* the packet is discarded.

Because network monitors often want only a small subset of network traffic, a dramatic performance gain is realized by filtering out unwanted packets in interrupt context. Thus, if the packet is not accepted, only those bytes that were needed by the filtering process are received by the host.

Packet filtering techniques have evolved over time in order to cope up with the increasing speed of the interface and increasing traffic rate.

4.2 Filtering Techniques

Filter rules normally reflect security policies, which services are allowed, where they are allowed, which type of packets can reach a target device, which ones should be dropped, etc. So, packet filtering is the process of passing or blocking data packets, based on a set of user-provided rules, as they pass through a network interface. There are many

approaches to packet filtering. As the main focus of this thesis is on traffic monitoring, we are merely interested in packet filtering as a simple function that drops/accepts packets out of a network interface. Several filtering techniques are well known in literature: some of the most frequently used filtering approaches are BPF, CMU Filter and Dynamic Packet Filter.

4.2.1 CMU/Stanford Packet Filter

CMU/Stanford Packet Filter (CSPF) is the first user-level packet filter. For the existing computers in that era, CSPF performed efficiently and provided flexibility in the environment, with an interpreter based filter mechanism. The specified filter language is stack based, and operates on binary expressions as well as Boolean operators. It uses a tree model to configure its filter engine. It believes that user-level de-multiplexing will incur more context switches and inter-processes. In order to provide both flexibility and efficiency, CSPF specifies filter in user-level and processes in kernel-resident using a specific packet field as a key. The problem of this approach is the bottleneck in operations that has a data structure of 16-bit words array.

4.2.2 The BSD Packet Filter

The BSD Packet Filter (BPF) is a more efficient interpreter than CSPF. It is an interpreter based on a register-based filter mechanism. BPF with register-based and assembly-like language can access more instructions, multiple registers, one input data and a scratch memory. It uses a computationally equivalent directed acyclic control flow graph, in order to avoid redundant computation [14]. It learns packet parse states in

the graph, so it reduces some paths and comparisons. It also uses a Boolean expression tree. The major performance improvements of BPF comparing to CSPF, are due to architectural improvements (registered based RISC CPU).

When a packet arrives at a network interface the link level device driver normally sends it up the system protocol stack. But when BPF is listening on this interface, the driver first calls BPF.

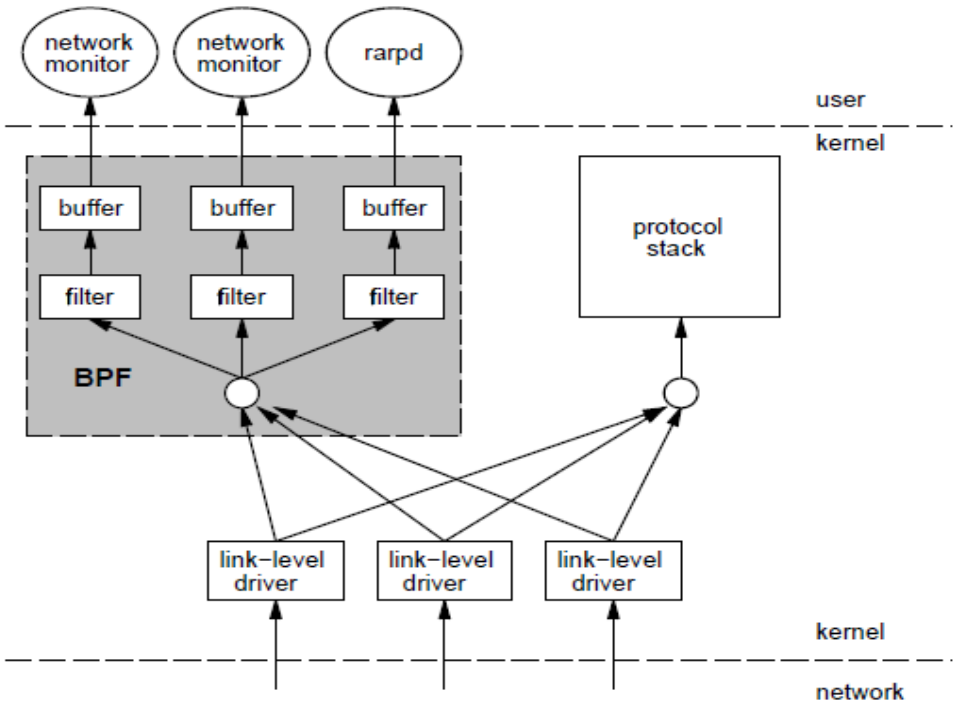

Figure 11: BPF Architecture

Source: The BSD Packet Filter: A New Architecture for User-level Packet Capture

BPF feeds the packet to each participating process filters. This user-defined filter decides whether a packet is to be accepted and how many bytes of each packet should be saved. For each filter that accepts the packet, BPF copies the requested amount of data to the buffer associated with that filter. The device driver then regains control. If the packet was not addressed to the local host, the driver returns from the interrupt. Otherwise, normal protocol processing proceeds.

4.2.3 Just in Time BPF

Similar to the normal BPF, but it adds a just-in-time compiler into the kernel to translate BPF code directly into the host system's assembly code. The simplicity of the BPF machine makes the JIT translation relatively simple; every BPF instruction maps to a straightforward machine instruction sequence. There are a few assembly language helpers which help to implement the virtual machine's semantics; the accumulator and index are just stored in the processor's registers. The resulting program is placed in a bit of vmalloc() space and run directly when a packet is to be tested. A simple benchmark shows a 50ns savings for each invocation of a simple filter - that may seem small, but, when multiplied by the number of packets going through a system, that difference can add up quickly.

4.2.4 Dynamic Packet Filters

Dynamic Packet Filter (DPF) approaches are currently the state of the art in the packet filtering world. DPF provides the most wanted flexibility of packet filters and the speed of hand-crafted demultiplexing routines. DPF filters are the fastest packet filters available. DPF achieves

high performance by using a carefully-designed declarative packet filter language that is aggressively optimized using dynamic code generation. The declarative packet filter language is used by protocols to describe the message headers that they are looking for.

4.3 Linux Packet Filter

Functioning of a Linux packet filter can be explained in a nutshell as follows [3]

- Creates a special-purpose socket (i.e., PF_PACKET)
- Attach a BPF program to the socket using the *setsockopt* system call
- Sets the network interface to promiscuous mode with *ioctl*
- Read packets from the kernel, or send raw packets, by reading/writing to the file descriptor of the socket using *recvfrom/ sendto* system calls

Sample Usage:

```
static void attach_filter(void) {
struct sock_fprog filter;
if ((sock = socket(PF_PACKET, SOCK_RAW, htons(ETH_P_ALL)))== -1)
{
        (void)fprintf(stderr, "Error: %s\n", strerror(errno));
        exit(4);
}
if (ioctl(sock, SIOCGIFFLAGS, &req) == -1){
        (void)fprintf(stderr, "Error: %s\n", strerror(errno));
        exit(4);
```

```
        }
        req.ifr_flags |= IFF_PROMISC;
        if (ioctl(sock, SIOCSIFFLAGS, &req) == -1){
                (void)fprintf(stderr, "Error: %s\n", strerror(errno));
                exit(4);
        }
        filter.filter = bpf_code;
        filter.len = FT_LEN;
        if (setsockopt(sock, SOL_SOCKET, SO_ATTACH_FILTER, &filter,
        sizeof(filter)) == -1){
                (void)fprintf(stderr, "Error: %s\n", strerror(errno));
                exit(4);
        }
        return;
}
```

4.4 PFQ Packet Filter

Packet filtering module in PFQ Engine is located right after the NIC (Network Interface Card) in order to drop unnecessary packets before being copied into kernel buffer. The aim of this thesis is to implement a bank of bloom filters for the PFQ in order to speed up packet processing by discarding unwanted packets.

4.5 Bloom Filter

As the speed of the network interfaces are increasing drastically, a fast, time and space efficient strategy is required to check whether a rule is

defined on a particular interface or not to speed up packet filtering. Bloom filter is a space-efficient probabilistic data structure for group membership query. It is widely used in network monitoring applications which involve the packet header/content inspection. To provide fast membership query operation, this data structure resides in the main memory in most of its applications. Each membership query consists hashing for a set of memory addresses and memory accesses at these locations. The space efficiency is achieved at the cost of a small probability of false positive, that is, an element may be announced as in the set while it is not. By exploiting the probabilistic nature of this data structure, the probability of false positive, i.e., false positive rate, is usually very small and outweighed by the space saving.

In Bloom filters, False positives are possible, but false negatives are not; i.e. a query returns either "inside set (may be wrong)" or "definitely not in set". Elements can be added to the set, but not removed (though this can be addressed with a counting Bloom filter). More the elements are added to the set, the larger the probability of false positives. The size of the bloom filter is to be selected in such a way that the false positive is below a certain level.

4.6 Working of Bloom Filter

A Bloom filter consists of a bit array of m bits, which are all initially set to 0. Adding the elements of a set S = $\{x_1, x_2, \ldots. x_n\}$ of *n* elements is done as follows. For each element x_i that is added, k different hash functions $h_1, \ldots, h_k$ each with a range {1,...,m} are used to calculate k different hash values $h_1(x_i),\ldots.h_k(x_i)$. We assume that these hash functions map each element to a random number uniform over their range. Then, the

bits $h_j(x_i)$ are set to 1, for j = 1, 2, …., k. The bits in the Bloom filter can be set to 1 multiple times, but only the first time this has effect.

For example, consider a bloom filter of 8 bits which is all set to zero at the startup. Consider two hash functions h_1 *and* h_2 which returns values between 1 and 8 inclusive. Given an element y=4 to be inserted, hashing using both the functions h_1 *and* h_2 are done.

0	**0**	**0**	**0**	**0**	**0**	**0**	**0**

Figure 12: Empty Bloom Filter of 1 byte width

Let us suppose that $h_1(y)$=4 and $h_2(y)$=2 , then bit 4 and 2 are set in the bloom filter indicating that the membership of the element is marked. Similarly consider the insertion of another element say z=6. Let's suppose that the result of hashing is $h_1(z)$=7 and $h_2(z)$=4 .

0	**1**	**0**	**1**	**0**	**0**	**1**	**0**

Figure 13: Bloom Filter after insertion of element y and z

It can be noted that, though $h_1(y) = h_2(z) = 4$, it is set only once. If a bit is set once, setting it again doesn't have any effect. Once an element is inserted into the bloom filter it cannot be removed which signifies that, a bit set in the bloom filter cannot be cleared because some other element might have set the same bit as a result of hashing during its insertion.

When we want to check if a certain element is in our Bloom filter, a similar approach is used as during the addition of elements. The same k hash functions are calculated over the element y. Then we test if the bits $h_i(y)$ for i=1, 2, …., k are equal to 1. If one or more of these bits are still 0, the element is certainly not in the set. If all bits are 1, the element was probably in the set, although there is a small probability that the tested bits were set to 1 due to the addition of different elements. Then we have a false positive.

For example, suppose we search for the element x. we perform the hash of x and we get $h_1(x)$ =2 and $h_2(x)$ =7. After checking the bloom filter to determine whether the corresponding bits are set or not, we find that they are set. It means x may be present because there may be some other element which could have set these bit positions. In this case these bits are set by y and z. From this we can clearly say that there is some probability of false positive but false negative can never occur in the bloom filter which is an important feature of bloom filter.

An interesting property about Bloom filter is that, any Bloom filter can represent the entire universe of elements. In this case, all bits are 1. Another consequence of this property is that add never fails due to the data structure "filling up." However, the false positive rate increases steadily as elements are added until all bits in the filter are set to 1, so a false response is never returned. At this point, the Bloom filter completely ceases to differentiate between differing inputs, and is functionally useless.

4.7 Calculating probability of false positive

There is a trade-off between the probability of false positives and the size of the Bloom filter. The false-positive probability can be calculated from *m* and *k* in the following way.

Assume that a hash function selects each array position with equal probability. If *m* is the number of bits in the array, the probability that a certain bit is not set to one by a certain hash function during the insertion of an element is then 1 – (1 /m).

If there are *k* hash functions, then the probability that it is not set by any of the hash functions is 1- $(1/m)^k$. If we have inserted *n* elements, the probability that a certain bit is still 0 is $(1 – (1/m))^{kn}$. The probability that it is 1 is therefore 1 - $(1 – (1/m))^{kn}$.

Each of the *k* array positions computed by the hash functions is 1 with a probability as above. The probability P of one of the *m* bits still being zero after the addition of *n* elements is

$$\mathbf{P = (1- (1/m))^{kn} \approx e^{-(kn/m)}}$$

The probability of a false positive *f* is then equal to the probability that all the *k* bits that we test are equal to 1, which is equal to

$$\boldsymbol{f = (1 – P)^k \approx (1 – e^{-kn/m})^k}$$

By taking the derivative of above equation, from simple calculations it follows that for a given *m* and *n*, the value of *k* that minimizes the false-positive probability *f* is equal to

$$\boldsymbol{k = (m / n) \ln 2}$$

A proper and careful design helps to minimize the probability of false positive.

4.8 Why PFQ Packet Filter uses Bloom Filter?

Bloom filters have a strong space advantage over other data structures for representing sets, such as self-balancing binary search trees, tries, hash tables, or simple arrays or linked lists of the entries. Most of these require storing at least the data items themselves, which can require anywhere from a small number of bits, for small integers, to an arbitrary number of bits, such as for strings (tries are an exception, since they can share storage between elements with equal prefixes). Linked structures incur an additional linear space overhead for pointers. An important property of bloom filter that makes it efficient is that the elements themselves are not stored in the Bloom filter, only their membership is stored. Hence it is space efficient.

Bloom filters also have the unusual property that the time needed either to add items or to check whether an item is in the set is a fixed constant, O(k), where k is the number of hash functions and is completely independent of the number of items already in the set.

It is possible to store the entire bloom filter in the cache and lookup can be done much faster without moving the structure between primary and secondary memory when compared to other data structures which often requires that. Hence bloom filter possesses an extremely efficient Cache Utilization.

The following graph taken from [15] shows that, even for a cache size of as small as 512 KB, many number of rules can be stored in the bloom filter for

a given false positive percentage. Actually speaking, it is much more than the requirement in this scenario, because no more than 1, 00, 000 rules will be inserted even by a high end network monitoring application. Hence bloom filter is chosen for implementing a filter module on top of PFQ.

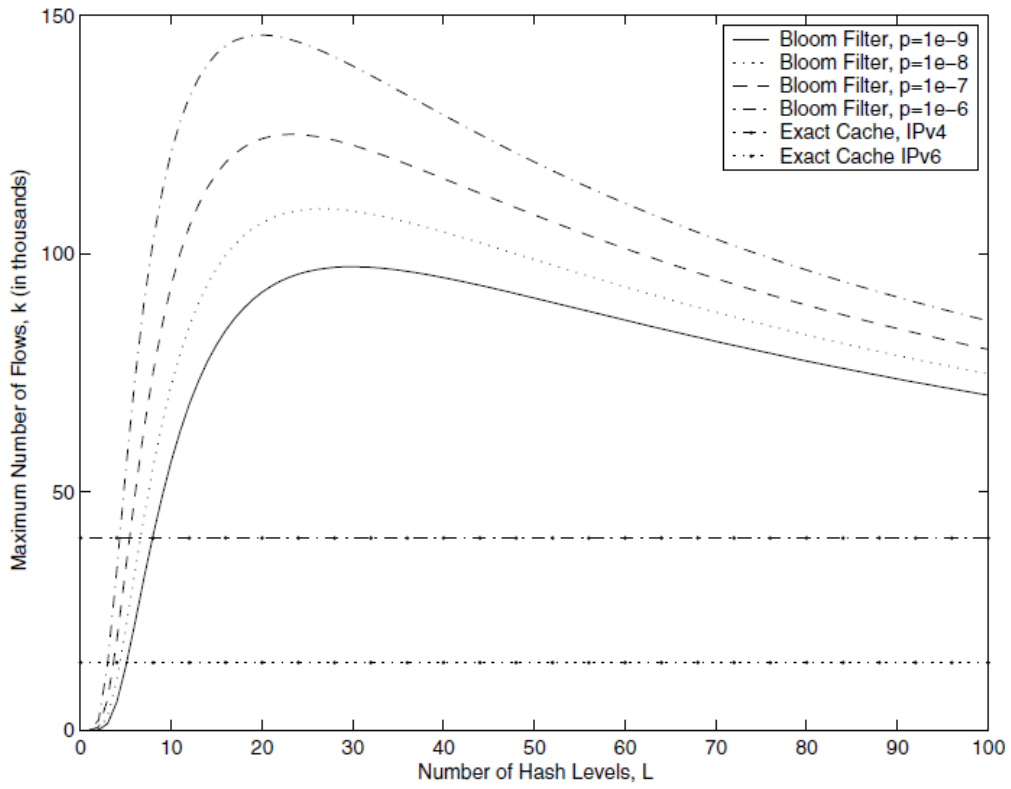

Figure 14: The maximum number of elements that can be stored by a 512KB cache

Source: F Chang, K Li, W Feng, Approximate Packet Classification Caching, IEEE INFOCOM 2004

4.9 Functioning of PFQ Packet Filter

To start with, PFQ must be enabled by the network monitoring application. Once the PFQ is enabled, the fetcher dequeues the packet directly from the driver, which can be a standard driver or a patched aware driver, and inserts it into the batching queue. Then the de-multiplexing block, selects which socket(s) need to receive the packet and steers the packets to respective socket queue, which represents the interface between user space and kernel space. The entire kernel processing (from the reception of the packet up to its copy into the socket queue) is carried out within the NAPI context;

After enabling PFQ, the next stage is to enable the bloom filter using the provided user space library function which is similar to enabling the PFQ itself. Once enabled, the kernel internally sets up the environment by allocating space for the PFQ packet filter for each interface and enables the filter on all interfaces.

Once enabled, the monitoring application can now start inserting the rules. The rule to be inserted is a combination of rule and its corresponding signature. Rule is nothing more than the usual 5-tuple *(Source IP, destination IP, Source Port, Destination Port, Protocol)* information. The signature for a rule is a provision to provide the mask for the corresponding field in the rule 5-tuple. The possible rule and signature combination that any network monitoring application will use is one among the following:

(1) The mask for Source and Destination IP field will be the subnet mask. And this field can generally vary from /8 to /32. If the monitoring application is looking for a specific IP address then corresponding mask will be /32 else if it is looking for a range of IP addresses, then the

corresponding subnet mask has to be set as the mask. For example, if the application is looking for traffic from a source IP 131.114.53.2 then it must use a mask of 32 and if it interested in traffic from a range of IP say, 195.114.53.1 through 195.114.53.254, then it has to use a mask of 24.

(2) Similarly the network monitoring application may be either interested in a particular source or destination port else it may want to listen any port. If the application wants to listen to a specific port then the mask used is 16 which is the width of the port (number of bits) field in TCP or UDP header else if doesn't care about the port, then it can denote the wildcard behavior by setting the port mask as 0. If the mask is '0', then it means that the application will accept traffic from any port number.

(3) Similarly the network monitoring application may be either interested in a particular higher layer protocol say, TCP/UDP/ICMP or it doesn't worry about the type of the higher layer protocol. If the application is protocol specific, then the mask for protocol is set to 8 which is the width of the protocol (number of bits) field in IP header. If the application is not protocol specific, then it means it accepts packets from any layer 4 protocols which are specified by setting the mask as 0 indicating the wildcard behavior.

An example of a rule, signature combination is as follows

	Rule	**Signature/Mask**
Source IP	10.99.124.99	32
Destination IP	87.16.135.137	32
Source Port	Any	0 (*)
Destination Port	Any	0 (*)

Protocol	TCP(6)	8

According to the above rule, the PFQ filter will check all incoming packets to determine whether it is from a source with IP 10.99.124.99 to a destination with IP 87.16.135.137 irrespective of what the source port and destination port is and the protocol must be TCP. If a packet passes the above check, then it will be forwarded to the monitoring application(s).

After inserting the required rules with its corresponding signatures, the monitoring application must invoke the sort function defined in the PFQ user library. The job of the sort function is to order the signatures in such a way that most of the matches of the incoming packets occurs at the beginning of the signature list during the filtering phase. Internally, the unique signatures are stored as a list in order to be memory efficient. List implementation doesn't have any memory overhead as it doesn't require any pre-allocation of memory. As and when a new signature is to be inserted, the space is allocated for it dynamically.

This trivial sorting function used sorts the signature in such a way that more specific signature requirements are moved to the end of the list and those signatures with few or all wildcard options set appear at the beginning of the list. It uses a simple ranking method that provides highest ranking to the more specified signatures and least ranking to less specified or wildcard signatures and are sorted in increasing order of their ranks. The ranking algorithm is as follows

```
typedef int rank_t
rank_t rank(struct signature *sign)     {
```

```
        return (sign->source_ip_mask) + (sign->destination_ip_mask)
        + (sign->source_port_mask) + (sign->destination_port_mask)
        + (sign->protocol);
    }
```

For example say if the monitoring application has two rules as follows

Rule 1:

	Rule	**Signature/Mask**
Source IP	10.99.124.99	32
Destination IP	87.16.135.137	32
Source Port	Any	0 (*)
Destination Port	Any	0 (*)
Protocol	Any	0(*)

The ranking function returns 64 for rule 1 as its rank.

Rule 2:

	Rule	**Signature/Mask**
Source IP	12.0.0.0	8
Destination IP	87.16.0.0	16
Source Port	Any	0 (*)
Destination Port	Any	0 (*)
Protocol	Any	0(*)

For rule 2 the ranking function returns 24 as its rank. So after sorting the rule 2 will appear ahead of rule 1 in the signature list.

In order to understand why less specified rules are sorted to appear ahead of more specified rules, consider the following scenario. Since it is difficult to visualize the problem in five dimensions (Source IP, Destination IP, Source port, Destination port, Protocol) which is a Hyper-Cube, let us consider only two dimensional (Source and destination IP) representation in order to better understand the scenario. Let us assume the random distribution of packet (i.e) the incoming packet is random and probability of a packet with any (source IP, destination IP, source port, Destination port and protocol) of the 5-tuple combination is equal.

Figure 15: Visualization of Ranking Mechanism

In order for a filter to be fast and efficient it should be able to return true or false in minimum comparisons. It can be seen that, it is more probable for a random packet to fall into 2D space of less specified rule than into the 2D space of well specified rule. Hence in order for the filter to be optimal, it is therefore logical to place less specified signature above the more specified signature.

For example, if the Rule 1 and Rule 2 above mentioned are taken into account, then a randomly generated packet is more likely to be of type

Rule 2 because the 2D space of Rule 2 is much bigger than the Rule 1 and the packets are likely to fall into that space.

This is further optimized by removing the signature subset problem. For example, there can be a more specific rule defined earlier. And followed by its definition there can be a new rule definition which is a wildcard one and it also encompasses the space of the previously defined rule. So in this case instead of storing both the rules signatures we can only store the superset of both the rules in the signature set as the number of different signatures affects the performance of the system.

For example, if we have two rule as follows

Rule 1:

	Rule	**Signature/Mask**
Source IP	10.99.124.99	32
Destination IP	87.16.135.137	32
Source Port	80	16
Destination Port	8080	16
Protocol	TCP(6)	8

Rule 2:

	Rule	**Signature/Mask**
Source IP	10.99.124.0	24
Destination IP	87.16.0.0	16
Source Port	Any	0(*)
Destination Port	Any	0(*)
Protocol	Any	0(*)

In above example, it is clearly visible that Rule 1 is a subset of Rule 2. Hence instead of storing both the signatures in the signature set only the Rule 2 signature will be stored. There is an obvious tendency to think whether the corresponding rule of the signature which is dropped will also be omitted. The answer is no. Here our consideration is only the signature and not the rule. Always all the rule are set. There can be any number of rules and it doesn't affect the performance of the system. It is only the number of signatures which determines the performance of the system because as the number of signatures increases it is possible that for each and every packet received we may have to traverse the entire signature list if there is no match found. So, it is always desirable to minimize the number of signatures by merging few of them. It may result in explosion with respect to the number of rules. But since the number of rules set doesn't affect the systems performance it is not of much significance until the number of rules doesn't exceeds the bloom filter threshold.

It can be assumed that logically each different signature inserted is associated with a bloom filter. Whenever there is an exact signature match during insertion, their corresponding rules are stored in the same bloom filter. The procedure for rule insertion is discussed in the next section. In practice, these logical bloom filter's can be realized using a single large bloom filter such that the probability of false positive in it is below the expected threshold offered by the logical bloom filter's.

Once the rules are inserted then it is left to the PFQ to handle things. PFQ inspects the incoming packets and when the inspection succeeds then it forwards the packet to the monitoring application for further processing.

If not, the packet is dropped because if the inspection failure, it indicates that no monitoring application is interested in this packet.

4.10 Rule Insertion and Inspection

4.10.1 Outline of Rule Insertion procedure

Once the monitoring application specifies the filtering rules, then those rules are sorted in such a way that more specified rules are moved below and the less specified rules are retained at the top of the list. After the sorting is performed, the rules which are subset of other rules are merged. Now the rules and its corresponding signatures are sent to PFQ filter module which performs the insertion signature as follows

(1) Check whether both PFQ as well as PFQ Bloom filter is enabled. If not, no rule insertion is performed and all the packets are forwarded to the monitoring applications without filtering.

(2) Determine the interface for which the rule is set. Check the filter corresponding to the interface to determine whether the signature to be inserted is already present in the signature list of the interface. If yes goto step3. Else, insert the current signature in the signature list.

(3) perform a bit-wise AND operation of the rules and its corresponding mask. Some shift are done on certain fields of the outcome in order to stretch the 32 bit AND operation to a width of 64 bits.

(4) The hash function is a simple XOR of the obtained 5-tuple information which is the result of previous AND operation. The obtained 64-bit hash value is split into four 16-bit hash values and the corresponding 4 (even if not optimal, is chosen here as the number of k hash functions for simplicity) bits are set in the filter of the specified interface. The signature

count is also increased for that particular interface if a new signature is appended to the list.

4.10.2 Filtering Procedure

After setting the rules in the bloom filter corresponding to each interface specified by the monitoring application, the following task is implemented to carry out the filtering functionality.

When the packet is received by the network card, the device driver creates a socket buffer called sk_buff, puts the packet into it and passes it to the Linux networking stack for further processing. Sk_buff structure contains all the information about the received packet including the interface through which the packet was received.

1. The PFQ filter first checks whether the bloom filter is enabled or not. If it is not, then it returns true else continue to step 2.
2. Get the interface index of the packet received from sk_buff structure and check whether the filter is enabled for such interface. If not returns true else continue to step 3.
3. Get the higher level protocol from the Ethernet header. If it is not IP then returns true else go to step 4.
4. Get the higher level protocol from the IP header. If it is not TCP or UDP then returns true else go to step 5.
5. Extracts the 5 tuple(*Source IP, destination IP, Source Port, Destination Port, Protocol)* Information and perform a bit-wise AND operation with the corresponding mask fields in the current signature set element,

then perform the hashing which is nothing more than a simple XOR of all the 5 tuple resulted from the previous AND operation . The obtained 64-bit hash value is split into four 16-bit hash values. The filter corresponding to the packet's received interface is examined for all these 4 bit positions. Goto step 6.

6. If all the 4 bit positions are set in the filter, then return true because a matching rule is present for the interface and the packet has to be allowed. If not make the current signature to be the next signature in the set and goto step 5. If the signature set is empty, then return false because no match for the packet is found and hence it should not be allowed as the application monitoring is not interested in this packet.
7. Repeat the above steps 1 through 6 for each of the received packet to perform the filtering.

Chapter 5

PFQ Filter – Software Development and Performance Analysis

5.1 Bloom Filter Design

In the case of four hash functions and a false positive probability of 0.2%, the number of bits to be in the bloom filter is calculated as 65535. The device considered has 64 interfaces which can be modified depending on the requirement. The PFQ filter is defined in the next section.

The Packet Filter Module allows the network monitoring applications to specify a set of rules and uses these rules to take respective actions on each incoming packet. Figure 17 depicts the filtering mechanism. The filtering module allows the user to specify the rules, which is the 5-tuple with the corresponding masks and wildcards thus providing additional provisioning. The masks are considered as signatures of the rules. Given the set of rules and their corresponding signatures, the signature insertions into the kernel space are sorted out in a manner according to certain ranking criteria such that most of the packet matches with the initial signature in the signature set thus reducing the number of per packet comparison resulting in improved throughput as discussed in the previous chapter.

Before insertion of the signatures into the signature set, care is taken to determine whether any signature is a subset of an already existing signature for a given interface. If so, it is eliminated or else is accepted to be a part of signature set for that interface.

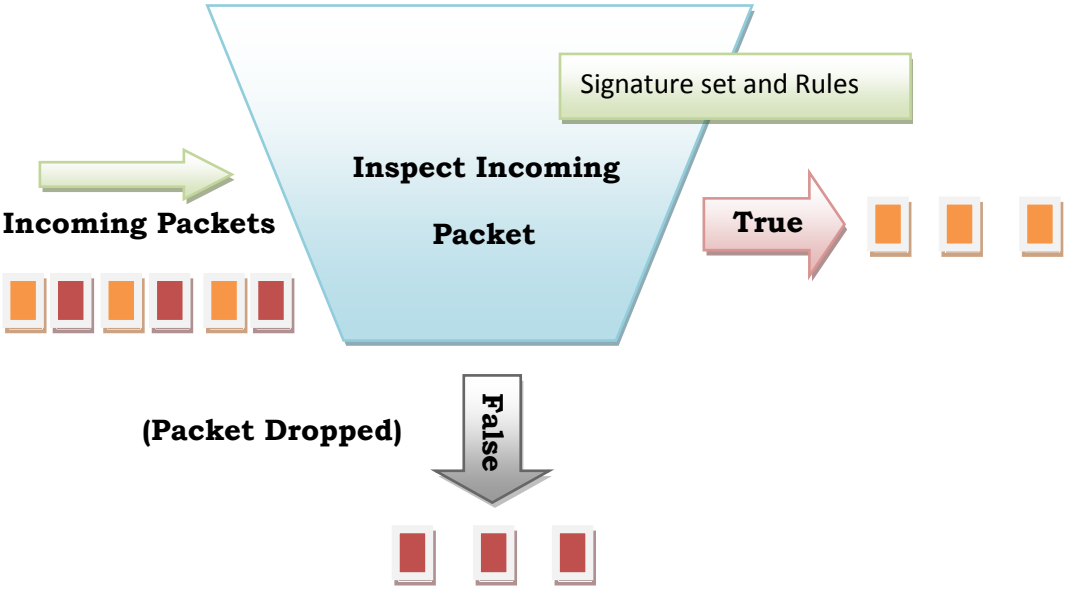

Figure 16: PFQ Packet Filter

5.2 Data Structure

In this section the main data structures used in the PFQ filter are examined in detail.

```
typedef struct Filter {
char filter[8192];
MASKSET *signature_set;
int signature_set_count;
bool enabled;
int rule_count;
}FILTER;
extern FILTER packet_filter[NO_INTERFACES];
```

The bloom filter is defined for each interface and it consists of 5 fields as defined above. The maximum number of interface in the system is configured to 64. It can be changed depending on the requirement. Each interface consists of a *filter* which is the bloom filter itself*, enabled* flag to check whether filter is enabled for that interface or not. The *rule_count* member is used to determine the number of rules set for this interface. If *rule_count* reaches a certain limit, rule insertion for that interface is not allowed and a notification is raised indicating that further addition of rules to the bloom filter will increase the false positive. *signature_set* contains the list of signatures that are inserted for this particular interface and *signature_set_count* contains the number of signatures set for this interface. Of course point to be noted here is that the signatures and the rules are not be confused. The system performance is not affected by the numbers of rules set but it depends on the number of signatures present. In a normal scenario, one can assume that only a limited number of signatures are required.

5.3 Filter Specific Options

For the Bloom filter that has been defined, a few set of *get* and *set* socket options at the socket level to perform the required operations.

#define SO_GET_BLOOM_RULE_COUNT 129

To check if a maximum rule is reached in the bloom filter or not in order to guarantee requested false positive percentage (%). This is done for each and every rule insertion performed by the monitoring application to

ensure that the rule count doesn't exceed the maximum allowed to maintain the false positive probability below the threshold.

#define SO_GET_BLOOM_STATUS 130

To check whether bloom filter is enabled or not.

#define SO_GET_BLOOM_SIGNATURE_COUNT 131

Returns the number of signatures set for this interface

#define SO_TOGGLE_BLOOM 108

To enable or disable bloom filter. Enable = 1, Disable =0

#define SO_SET_BLOOM 109

To apply a rule by performing hash function and enabling the corresponding bits in the bloom filter

#define SO_RESET_BLOOM 110

Reset Bloom filter is about clearing bloom filter and the signature list for all the interfaces. This consists of performing a set of actions on each interface.

(1) If the signature set for that interface is not empty, then traverse along the signature list and free each and every element in the list

(2) Reset all the bits in the bloom filter to 0

(3) Set *signature_set_count* and *rule_count* to 0.

(4) Set the enabled flag for that interface to false.

#define SO_RESET_BLOOM_INTERFACE 111

This is variation of **SO_RESET_BLOOM** in that, it allows to reset bloom filter and the signature list on a specific interface.

5.4 User API

Before monitoring applications set their rules on bloom filter corresponding to their respective interfaces, the rules and signature manipulations are to be done. These manipulations consists of ordering the signatures in an optimal way such the matching occurs in the initial part of the signature list and also the handling of signature subset elimination problem. To support these manipulations, *pfq_trivial_merge* class is defined which contains the methods to handle signature sorting and signature subset elimination problem.

To begin with, an instance of *pfq_trivial_merge* class is defined and all the rules are inserted. The insertion method is as follows

pfq_trivial_merge m;

m.insert(address("10.99.124.54",source_mask), address("131.114.55.255.137",destination_mask),any<port_t>(), any<port_t>(), protocol(17));

Here *source_mask* and *destination_mask* is the required subnet mask for source and destination IP (or sub network) the monitoring application is looking for. *source_mask* and *destination_mask* can take any value between 8 and 32.

```
template <typename Tp>
inline std::pair<Tp, int> any() {
        return std::make_pair(0,0);
}
```

any<T>() is a generic template function which can be used by both protocol and port fields. Depending on the whether the port or protocol type is passed as type, the corresponding type mask pair is returned.

```
inline std::pair<std::string, int> address(std::string addr, int
mask) {
        return std::make_pair(std::move(addr), mask);
}

inline std::pair<uint16_t, int>  port(uint16_t p)   {
        return std::make_pair(p, 16);
}

inline std::pair<uint8_t, int>  protocol(uint8_t p)     {
        return std::make_pair(p, 8);
}
```

The above are the other methods that assist the insertion of the rules. After the insertion, the sort function is invoked. As discussed in the previous chapter, each of the signatures is assigned a rank and they are sorted in the increasing order so that less specified signature comes ahead of the

more specified rule. Once this is done, then the signature subset problem has to be solved as discussed in the previous chapter. Now, the monitoring applications are ready to go on with setting the rule in the corresponding interfaces.

The monitoring applications are provided with a set of simple API to work with the filters. As of now, few API's are available. One for enabling the bloom filter for all the interfaces, one for setting the bloom filter for the specified interface with the corresponding rule, one for determining the number of rules set for that interface, one for determining the number of unique signatures set for that interface and the final API for disabling the bloom filter for all the interfaces. *Setsockopt()* and *getsockopt()* are used to pass user space information to the kernel space and vice versa.

void setBloom(int interface,SIGN_AND_RULES sign_rules);

where SIGN_AND_RULES is a structure which consists of both the rules and signatures as is defined as follows.

```
typedef struct sign_and_rule {
        RULES rule;
        MASK mask;
}SIGN_AND_RULES;
```

RULES and MASK by themselves are five member structures which contains both the 5-tuple information and their corresponding masks.

When setBloom() API is called, it internally first checks whether the bloom filter is enabled or not. If not, returns run time error else invoke the *Setsockopt() with SO_SET_BLOOM* option and *the SIGN_AND_RULES* structure information is passed to kernel space with opval and oplen parameters.

5.5 Kernel level Implementation

5.5.1 PFQ Setsockopt()

*Static int pfq_setsockopt(struct socket *sock, int level, int optname,*

*char __user * optval, unsigned int optlen)*

Parameter Description

Level

The level at which the option is defined. Here in this case it is PF_Q level.

optname

The socket option for which the value is to be set (for example SO_TOGGLE_BLOOM,SO_SET_BLOOM,SO_RESET_BLOOM).

The *optname* parameter must be a socket option defined within the specified *level*, or behavior is undefined.

optval

A pointer to the buffer in which the value for the requested option is specified.

optlen

The size, in bytes, of the buffer pointed to by the *optval* parameter.

When the *optname* is SO_TOGGLE_BLOOM, copy the optval from user space and checks whether it is set or not. If optval is set, then *filter* is cleared, *enabled* flag is set to true and *rule_count* is set to 0 for all the interfaces. This option just enables the bloom filter for all the interfaces.

When the *optname* is SO_RESET_BLOOM, reset all the interfaces filter, set *enabled* to false and *rule_count* to 0.

When the *optname i*s SO_SET_BLOOM, copy the optval from the user space using copy_from_user() system call. Now, the optval contains the SIGN_AND_RULES structure which contains two 5-tuple information. One is the rule *(Source IP, destination IP, Source Port, Destination Port, Protocol)* and other is the corresponding signature *(Source IP Mask, destination IP Mask, Source Port Mask, Destination Port Mask, Protocol Mask)* information and the interface for which the rule is to be enabled. The point to be noted here is that all the duplicate and subset signatures are removed by functions implemented in PFQ user space library before passing the signatures to kernel space as discussed in the previous section. Now, perform a bit-wise AND operation of the rules and its mask received. The hash function is a simple XOR of the obtained 5-tuple information which is the result of previous AND operation. The obtained 64-bit hash value is split into four 16-bit hash values and the corresponding 4 bits are set in the filter of the specified interface. The *signature_count* is increased for that particular interface if it is a unique signature. The *rule_count* is also incremented irrespective of whether the signature is unique or not.

5.5.2 PFQ GetSockopt()

*Static int pfq_getsockopt(struct socket *sock, int level, int optname,*

*char __user * optval, int __user * optlen)*

Parameter Description

level [in]

The level at which the option is defined. Example. Here in this case it is PF_Q level.

optname [in]

The socket option for which the value is to be retrieved. Example: SO_GET_BLOOM_RULE_COUNT,SO_GET_BLOOM_STATS, SO_GET_BLOOM_SIGNATURE_COUNT. The *optname* value must be a socket option defined within the specified *level*, or behavior is undefined.

optval [out]

A pointer to the buffer in which the value for the requested option is to be returned.

optlen [in, out]

A pointer to the size, in bytes, of the *optval* buffer.

When *optname* is SO_GET_BLOOM_RULE_COUNT, then the interface index for which the rule count is to be determined is copied from user space by using the call copy_from_user() and then the rule count for that interface is returned to the user space using copy_to_user() system call.

When *optname* is SO_GET_BLOOM_SIGNATURE_COUNT, then the interface index for which the signature count is to be determined is copied from user space as before and then the signature count for that interface is returned to the user space using copy_to_user() system call.

When *optname* is SO_GET_BLOOM_STATUS, then a Boolean value indicating whether the bloom filter is enabled or not for the system is returned to the user.

5.6 Performance Analysis

5.6.1 Experimental Setup

Traffic generator used generates around 14 million packets per second. In order to consider the possible worst case, each packet is of minimum size which is not the case in real traffic. If a real time traffic is considered then the number of packets in the worst case will be around 1-2 million per second which is far lower than the scenario that is considered for study. A twelve core processor is used as a part of this experimental setup.

In order to understand the performance of the system in the presence of signature set on per interface basis, a worst case analysis is done by traversing through all the signatures on per packet basis even if a signature match occurs earlier in the signature list. Two characteristics are analyzed for determining the performance.

5.6.2 CPU load

The number of different signatures that are inserted is increased linearly and the CPU load under each case is evaluated. It can be seen that as the number of signature increases, the CPU load also increases until a threshold which is 12 and then there is a drastic increase in the CPU load. CPU load reaches 100% when the number of signatures is 26. Another important point to be noted is that this 26 is the worst case threshold since in normal scenarios the whole signature list is generally not checked on a per packet basis. The signatures are ordered in such a way that most of the matches happen in the beginning of the list itself. In average case the number of signatures supported can be much higher.

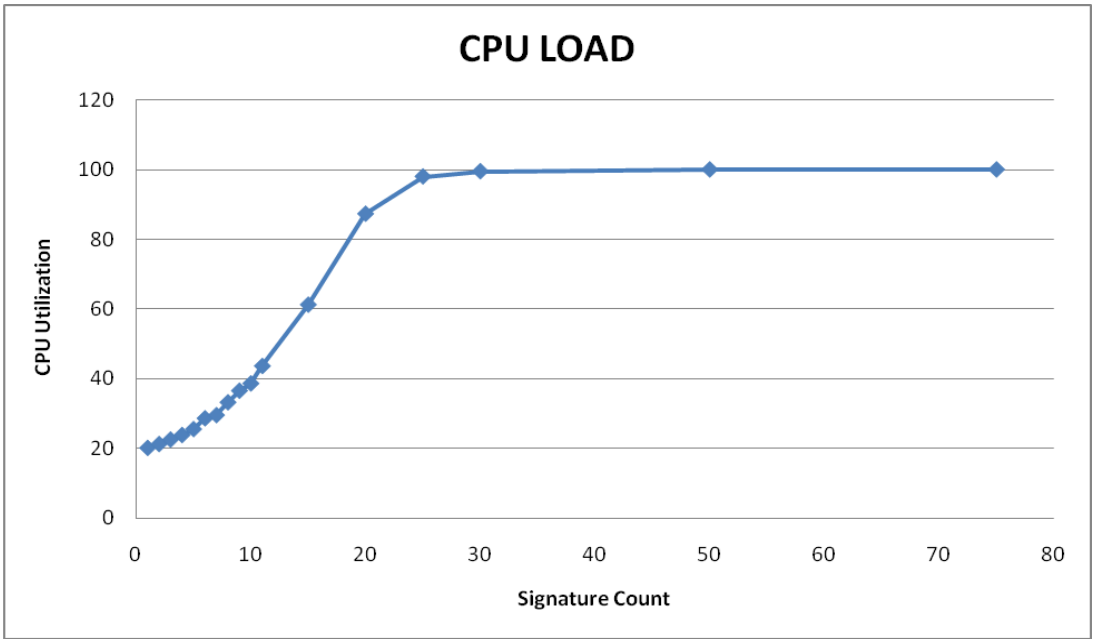

Figure 17: CPU Load

5.6.3 Percentage of Packet lost/Packet Offered

Similar to the previous analysis, the same experimental setup is maintained and the percentage of packet lost over packet offered is analyzed. It can be noted that we have loss of packet right from the point where signature count is zero. This is due to the Intel driver used because it is a default driver which is not optimized for high traffic load. Here the maximum traffic captured by the PFQ engine is 78.5% of the traffic generated. It was found that till the number of signature count reached 20, there was no loss due to the signature checking but still the initial losses due to the driver remained. When the signature count starts increased above 25, the packet loss begins and the loss increases.

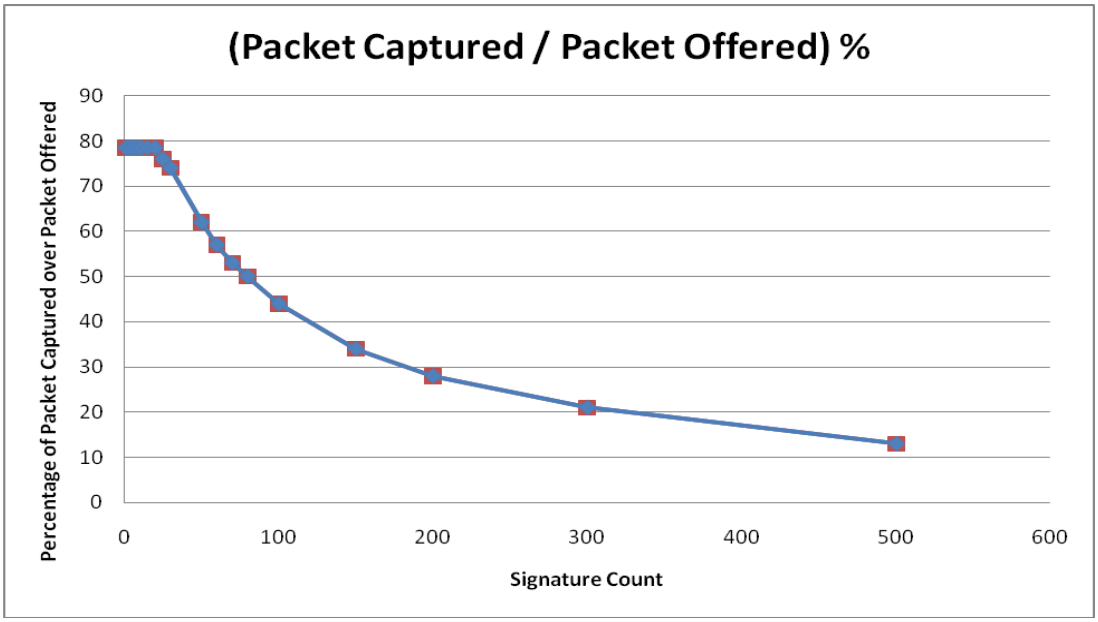

Figure 18: Percentage of Packet Captured vs Packet Offered

When the number of signature count is set to 500, only 10% of the total packet generated is captured. As mentioned in the CPU load analysis, it has to be noted that this is the worst case behavior and not all the signatures are verified for each packet. Hence in the average case, it can be expected that the packet capturing rate will be closer to 80%, just as in the best case.

Future work

In this thesis, the algorithm used for selecting the signatures associated with Bloom filters out of the ruleset specified by the user level applications is rather simple and does not reflect the actual statistical distribution of field's prefixes. The most natural step ahead for this research is to investigate upon possible optimization mechanisms in the selection of the best signatures to represent the whole ruleset. This may involve the use of operational research methods as well as clustering techniques to derive the "best" signatures according to the statistics of prefixes distribution within the rules database.

References:

. [1] http://www.gsp.com/cgi-bin/man.cgi?section=2&topic=socket

[2] http://swoolley.org/man.cgi/7/packet

[3] http://www.linuxjournal.com/article/4659

[4] J. H. Salim, R. Olsson, A. Kuznetsov. Beyond Softnet. Proceedings of the 5th Annual Linux Showcase & Conference 2001.

[5] http://en.wikipedia.org/wiki/Pcap

[6] Libpcap. Homepage http://www.tcpdump.org.
[7] L. Deri. Improving passive packet capture: beyond device polling. Proc. of SANE, 2004.
[8] http://www.ntop.org/products/pf_ring/
[9] M. Dashtbozorgi, and M. A. Azgomi. A high-performance and scalable multi-core aware software solution for network monitoring
[10] L. Deri Exploiting Commodity Multi-core Systems for Network Traffic Analysis

[11] N.Bonelli, A. Di Pietro, S. Giordano, and G. Procissi.PFQ: a Novel Architecture for Packet Capturing on Parallel Commodity Hardware.
[12] N.Bonelli, A. Di Pietro, S. Giordano, and G. Procissi. Packet capturing on parallel architectures. In IEEE workshop on Measurements and Networking, 2011.
[13] Intel. Accelerating high-speed networking with intel I/O acceleration technology. White Paper, 2006.
[14] S. McCanne, and V.Jacobson. The BSD Packet Filter: A New Architecture for User-level Packet Capture, USENIX conference, 1993
[15] Francis Chang, Kang Li, Wu-chang Feng, Approximate Packet Classification Caching, IEEE INFOCOM 2004