



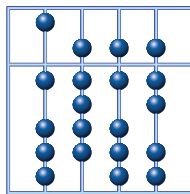
TECHNISCHE UNIVERSITÄT MÜNCHEN
FAKULTÄT FÜR INFORMATIK

Diplomarbeit

Performance evaluation of packet capturing systems for high-speed networks

Fabian Schneider

Aufgabenstellerin: Prof. Anja Feldmann, Ph. D.
Betreuer: Dipl.-Inf. Jörg Wallerich
Abgabedatum: 15. November 2005



Ich versichere, dass ich diese Diplomarbeit selbständig verfasst und nur die angegebenen Quellen und Hilfsmittel verwendet habe.

Datum

Fabian Schneider

Abstract

Packet capturing in contemporary high-speed networks like Gigabit Ethernet is a challenging task when using commodity hardware. This holds especially for applications where full packets (headers and data) are needed and packet loss is unwanted. Most network security tools—particularly network intrusion detection systems—have these demands.

Therefore, it is interesting to know if today's customary hardware and software are able to keep up with the network in terms of throughput of packet data. A methodology for evaluating the performance of different systems for packet capture is described and applied in this thesis. Four different PC based systems are compared with respect to their maximum capturing rate.

As modern PC's are available with different types of processor architectures which can then be equipped with different operating systems, it is interesting to find out which combination performs best for the task of packet capturing. The measurements performed for this thesis evaluate Intel Xeon against AMD Opteron based systems running either Linux or FreeBSD.

For this purpose, the Linux Kernel Packet Generator has been extended by the feature to not only generate packets of a given size, but to generate packets according to an underlying packet size distribution. This workload source provides all four systems with high bandwidth traffic which has to be captured.

The results show that the combination of AMD Opterons with FreeBSD outperforms all others, independently of running in single or multi processor mode. Moreover, the impacts of packet filtering, using multiple capturing applications, adding packet based load, writing the captured packets to disk, and available enhancements are measured and looked into.

Contents

1	Introduction	1
1.1	Motivation	1
1.2	Related Work	2
1.3	Structure of this Thesis	3
2	Background	5
2.1	How does Packet Capturing Work?	5
2.1.1	Capturing with FreeBSD – the BSD Packet Filter	5
2.1.2	Capturing with Linux – the Linux Socket Filter	7
2.1.3	libpcap – the packet capturing library	7
2.2	General Problems while Packet Capturing	8
2.2.1	Receive Interrupt Load	8
2.2.2	Packet Copy Operations	9
2.2.3	I/O Throughput	10
2.3	Environment	10
2.4	Architecture Comparison: Intel Xeon vs. AMD Opteron	12
2.5	Network Traffic Characteristics	14
3	Methodology	15
3.1	Items in the Testing Environment	15
3.2	Requirements	16
3.3	Test Setup	18
3.4	Measurement Cycle	18
4	Workload generation	21
4.1	Existing Tools for Packet Generation	21
4.1.1	TCPivo/NetVCR and tcpreplay	21
4.1.2	“Monkey See, Monkey Do” and Harpoon	21
4.1.3	Linux Kernel Packet Generator	22
4.1.4	Live Data	22
4.1.5	Summary	23
4.2	Identification of Packet Size Distributions	23
4.2.1	Analysis of Existent Traces	23
4.2.2	Representing Packet Size Distributions	24
4.2.3	Calculation of the Resulting Distribution	27
4.3	Enhancement of the Linux Kernel Packet Generator	28
4.3.1	Results	28

5 Profiling	31
5.1 cpusage	31
5.2 Postprocessing of the cpusage Results	32
6 Measurement and Results	35
6.1 Influencing Variables	35
6.2 Measurement Procedure	36
6.2.1 Testing Sequence	36
6.2.2 Calculation of the Results	37
6.3 Results	38
6.3.1 Impact of Buffers	40
6.3.2 Using Packet Filters	44
6.3.3 Running Multiple Capturing Applications	47
6.3.4 Adding Packet Based Load	49
6.3.5 Writing to Disk	53
6.3.6 Patches to the Linux Capturing Stack	54
6.3.7 Hyperthreading	56
7 Conclusion	59
7.1 Summary	59
7.2 Future Work	60
A Programs Written for the Thesis	63
A.1 createDist	63
A.1.1 Possible Input and Output	63
A.1.2 Functionality of createDist	64
A.1.3 Command-line Options	65
A.2 Linux Kernel Packet Generator – Packet Size Distribution Enhancement	66
A.2.1 How it Works	66
A.2.2 How to Use the new Enhancements	67
A.2.3 Changes Made	68
A.2.4 How to Compile and Install the New Module	69
A.3 cpusage	69
A.3.1 Command-line Options	69
A.3.2 Building and Installing	70
A.4 trimusage	70
B Plots	73
C Acknowledgements	77
List of Figures	79
Bibliography	81

1 Introduction

Packet capturing is the process of reading the transferred data units from a network link¹. When capturing on an Ethernet² link, one reads these whole data units, called frames, except for the preamble and the checksum which are stripped off by the network card. Because there are different protocol implementations for the data link layer³, and because the more interesting part of the captured data is inside the encapsulated network layer³ part of the frame, the term “packet” will be used for an Ethernet frame, an IP packet, or a transport protocol³ data unit. Furthermore, the term “packet capturing” refers to this “relaxed” meaning of “packet”.

1.1 Motivation

Packet capturing, or sniffing⁴, is the instrument used most often for network security tools, such as intrusion detection or prevention systems, and network analysis tools. In either case it is important to capture all the packets transferred over the link. This holds especially for application layer protocol analysis which requires the complete stream of packets of the examined connection. One could argue that often the necessary information needed for the analysis can already be found in the first few packets of a connection and thus losing packets (later on) does not hurt. But the opposite is the case: If only few packets per connection are required, it is exceptionally bad if exactly these packets are lost. Altogether, no packet should get lost.

Therefore, it is interesting to know if the system is able to keep up with the amount of data or with the arrival rate of the single packets. Usually, in a Fast-Ethernet² environment this is no problem for most of today's hard- and software. But looking at Gigabit-Ethernet², today's limits in terms of PCI bus⁵ throughput, CPU capabilities, storage speed, or application (resource) demands can easily be exceeded. And with the next step to 10-Gigabit-Ethernet² in mind, it is necessary to understand the complexity of packet capturing.

¹*Link* refers to a physical connection between network devices. In addition all attached devices are required to be ready for sending and receiving data for calling this connection a link.

²For protocol details of *Ethernet* please see [Tan03, Section 4.3, pages 271 et seqq.].

³These *layers* are part of the ISO/OSI reference model. See [Tan03, Section 1.4.1, p. 37 et seqq.].

⁴*Sniffing* is a more informal word and often bears a negative or illegal meaning. The term “sniffers” will be used for the machines performing packet capture.

⁵A *PCI bus* is used to connect peripheral components to the mainboard of a contemporary PC. Besides the slot itself the protocol and the speed of the bus is defined by PCI. Meanwhile different variants of PCI are available: PCI-X or PCIexpress are the newest among them.

The intended purpose of this thesis is to compare computers in multiple configurations while capturing packets. Thereby, many performance influencing factors are evaluated in order to gain more insight in the complex topic of packet capturing. The interpretation of the results of this comparison shall enable the reader to choose an appropriate system for his or her special task. The measurements for this comparison are all done with respect to the percentage of the captured packets in relation to the total available packets. This ratio is a metric for the packet capturing performance of the system.

The major factors influencing the performance of packet capturing, physical system throughput, and the delay and the amount of work between receiving the packet and delivering it to the application, are quite obvious. For the purpose of this thesis, it is necessary to supply the possibility to compare different flavors of these factors. On the one hand, two different architectures affecting the throughput were purchased: dual Intel Xeon and dual AMD Opteron machines. One of the fastest network cards and a system bus capable of at least 1 Gbit/s of continuous bandwidth were chosen. On the other hand, different operating systems supply different means and mechanisms to provide applications with raw network data. Thus, FreeBSD and Linux, the most popular operating systems for the task of capturing packets, are examined.

1.2 Related Work

The project thesis [Sch04] describes similar work on single-processor systems performed last year. The following paragraphs will give a brief summary of that work consisting of two main parts.

The first part was an analysis of the “capturing stacks” of Linux and FreeBSD. The mechanism used to capture packets is called capturing stack in analogy to the network stack which is used to receive and send packets and which is located in the kernel as well. The purpose was to identify the differences between Linux and FreeBSD, two readily available Open Source operating systems. A source-code investigation of the kernel and the libpcap [JLM94] (see Section 2.1.3) turned out that during the processing of a packet Linux copies the data two times whereas FreeBSD copies it three times, thus giving an advantage to Linux. On the other hand it is necessary to do reference counting to keep track of the packet data stored in the kernel memory. On that account the Linux kernel has to deal with big buffers that—in worst case—do not deplete.

The one additional copy which FreeBSD performs is due to a double buffer strategy used by the kernel built-in filter from which the application can read the packets. In contrast Linux stores the packets in an area of kernel memory reserved for incoming packets. The double buffer of FreeBSD is replaced by a queue of pointers. This leads to another difference: Under FreeBSD the whole content of one half of the

double buffer is copied into the user space, whereas Linux performs a separate copy operation for every single packet. The general principle of packet capturing with Linux and FreeBSD is explained briefly in Section 2.1.

The second part of the project was a comparison measurement of Linux and FreeBSD. It was not possible to generate test packets at line speed—so that the maximum throughput of the link is fully utilized—with the available hardware. Hence, only data rates between 150 MBit/s with minimal small packets (40 Bytes) and about 500 Mbit/s with maximum large packets (1500 Bytes) could be achieved.

With only one capture application, Linux performed better than FreeBSD. But with multiple concurrent capturing applications, FreeBSD shares resources more evenly between the applications than Linux does. In this case all applications obtain nearly the same number of packets with a deviation of about five percent under FreeBSD. Under Linux it happens that one application only sees about five percent and another application captures nearly all of the generated packets. Comparing the average capturing rates yields that this time Linux is inferior. Together with the findings from the first part, this shows that different operating systems cause different capturing performance.

Further results show that different network cards with their corresponding drivers just as buffer settings have a great influence on the loss ratio. For precise results please take a look at [Sch04].

In addition, Luca Deri [Der04, Der03] has to be mentioned who worked on improving the packet capturing abilities of Linux using a ring buffer design. He also did measurements on this topic. The performance of his patch was also subject to the project work described above.

1.3 Structure of this Thesis

In the next section (2.1) the principal procedure of packet capturing for Linux and FreeBSD is outlined as well as the most important procedures of the libpcap, a common library used for packet capturing. The difficulties of packet capturing—throughput, copy operations, and receive livelock—are discussed in Section 2.2. Subsequently, Section 2.3 presents our regular environment and applications of the systems under test. Because this thesis compares systems equipped with Intel Xeon and AMD Opteron processors, the differences of these architecture designs are pointed out in Section 2.4. A short summary of network traffic characteristics in Section 2.5 ends Chapter 2.

Chapter 3 introduces the methodology underlying the measurements. Section 3.2 first identifies the requirements on the hard- and software participating in the experiments. The test setup is shown in Section 3.3, and the measurement cycle is illustrated in Section 3.4.

Chapter 4 begins with the identification of the capabilities of existing tools for packet generation with respect to the intended measurements in Section 4.1. Because none of these tools meets the requirements stated in the previous chapter, the following sections describe how the Linux Kernel Packet Generator was improved. How to acquire suitable packet size distributions is explained in Section 4.2. The performance of this enhanced Linux Kernel Packet Generator is tested in Section 4.3.

Chapter 5 defines the form of profiling used to monitor the CPU usage of the systems under test.

Before the results of the measurements are shown in Chapter 6 the factors influencing the capturing process are itemized in Section 6.1. The parameters of the used methodology are specified in Section 6.2 and the results are stated in Section 6.3.

Chapter 7 summarizes the results and motivates further work.

A short guide to the programs written for this thesis is given in Appendix A. Supplemental plots of further results (Appendix B), and acknowledgements (Appendix C) complete this thesis.

2 Background

As this thesis aims at identifying benefits and drawbacks for packet capturing, different scenarios have to be evaluated. To understand which settings and circumstances affect the process of packet capturing, the process itself has to be apprehended. Furthermore, the surroundings of packet capturing have to be spotted. This chapter intends to provide the background for the remainder of this thesis.

2.1 How does Packet Capturing Work?

Before delving into the used environment and problems arising when capturing packets, this section explains how the capturing is done in general. But since every operating system has its own functionalities to actually perform the capturing and the filtering, this cannot be done quick and easy. Therefore, only the researched operating systems FreeBSD and Linux are elucidated. Please refer to [Sch04] for a detailed description. The research of further operating systems, such as Windows, MacOS or Solaris, would go beyond the scope of this thesis. But the equivalent mechanisms for capturing packets exist for these operating systems as well.

2.1.1 Capturing with FreeBSD – the BSD Packet Filter

The BSD Packet Filter (BPF) was introduced in 1992 by Steven McCanne and Van Jacobson [MJ93] as a packet filter for BSD. The main advantage of this at that time new design is to be able to filter the packets before they are copied to the user space¹, thus saving an expensive copy operation if packets are rejected by the filter. Another improvement is the new filter description language which is comparable to assembler. The benefit of this new language is that, instead of using a boolean decision tree, it uses a control flow graph, which allows shortcuts and smart filter programs. The filter is implemented as an independent machine within the kernel code. Figure 2.1 demonstrates the design of the BPF.

When packets arrive at a network card the card verifies the checksum, extracts the link-layer data, and triggers an interrupt². This interrupt calls the corresponding

¹In contrast to kernel space the *user space* is the segment of the memory of a system which is available to the applications. In opposite device drivers and kernel stuff is stored in kernel space.

²An *interrupt* is used to suspend the normal processor operation for special events. These events concern peripheral devices such as disks, or—in this case—network interface cards. In general, an interrupt signals the processor that data is ready to fetch. For further information on interrupts and their processing please refer to [Tan01, Sections 1.4.3 and 5.1.5, pages 30 and 279 et seqq.].

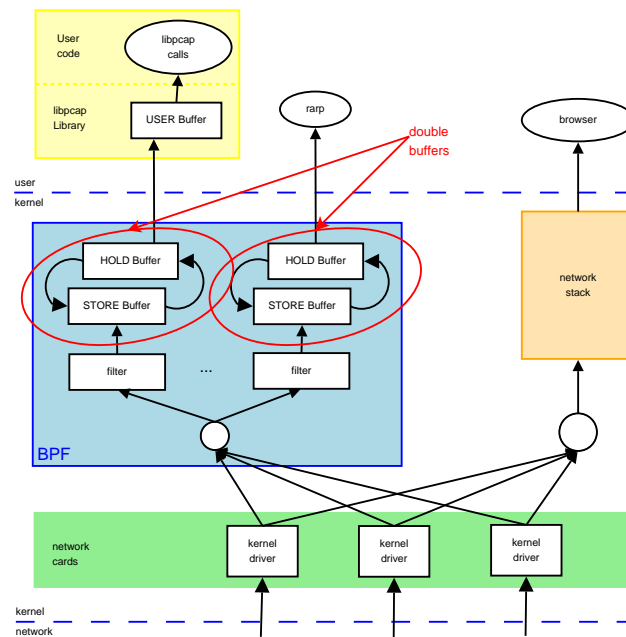


Figure 2.1: Block diagram of the BPF: This figure shows the general concept of the BSD Packet Filter.

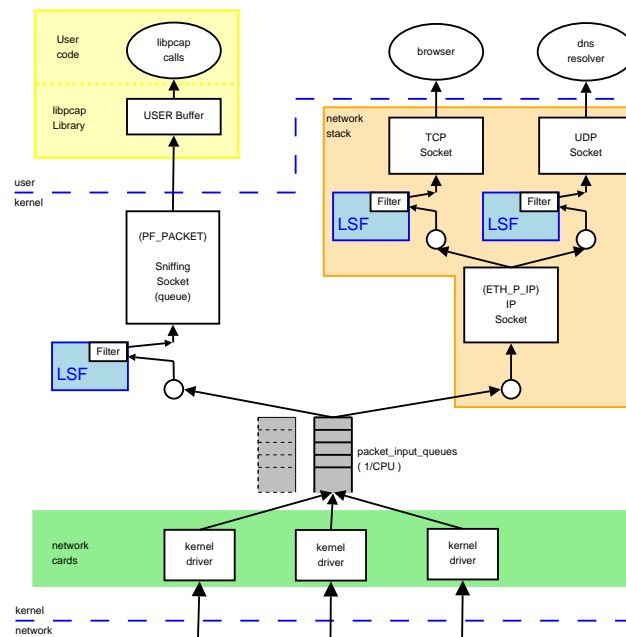


Figure 2.2: Block diagram of the LSF: This figure shows the general concept of the Linux Socket Filter

kernel driver for the card which generates the kernel structure for network packets. This is like adding a handle for the operating system to the packet. Then the kernel passes the packet to every BPF in use (one for each application which is capturing) and to the normal network stack. Still in the interrupt handler, the previously installed filter is applied, and if the packet is accepted it is copied into the STORE half of the double buffer. Now the captured data can be fetched through a read on the `/dev/bpf[0-9]+` device from the HOLD buffer. These buffers are switched if either the STORE buffer is full and a packet is waiting, or if the HOLD buffer is empty and the application performs a read. A more convenient way of accessing the data through the BPF is to use the interfaces supplied by the libpcap (see Section 2.1.3) which was developed in parallel to BPF.

2.1.2 Capturing with Linux – the Linux Socket Filter

The Linux Socket Filter (LSF) is a slightly extended BPF. The only conceptual difference is the ability to use it at diverse positions in the network stack in addition (see [Ins01, Ins02a, Ins02b] for details). The internal handling of packets within the Linux kernel is performed via so called “sockets”. These “sockets” have nothing in common with sockets known from UNIX, which are used to access the network from user space. A “socket” in the current context is a smart queue for pointers to packets stored by the kernel. Each “socket” is in charge for a class of packets. For example ETH_P_IP knows what to do with IP packets within the network stack.

In contrast to FreeBSD, the interrupt spawned by the network card only puts a pointer to the packet into a queue and schedules a soft-interrupt³. This soft-interrupt hands the packets over to every “socket” which fits to the given packet (see Figure 2.2). Thus, a “socket” matching any packet (PF_PACKET) is needed. As it is possible to bind an LSF on any “socket”, this concept is comparable to BPF, especially when considering that the filter description language is the one from BPF. The packets accepted by the attached LSF are queued. If no filter is attached, all packets are queued. If there are packets available, the applications registered for that socket are woken up to fetch the data. Contrary to FreeBSD, the packets are copied separately to the user space whereas FreeBSD copies the whole buffer at once.

2.1.3 libpcap – the packet capturing library

As previously mentioned, the libpcap [JLM94] is a C library for capturing packets. The procedures included in libpcap provide a standardized interface to all common (UNIX-based) operating systems, including Linux and FreeBSD. The interface of

³Unlike a (hardware) interrupt which is originated from some kind of attached hardware, a *soft-interrupt* is generated by the kernel to remind itself to do something. Soft-interrupts do not disturb the actual processing, but wait until it is the kernel's turn to use the CPU.

the libpcap is usable even under Windows but there the library is called winpcap. So it is useful to design applications to use libpcap for ensuring portability.

The most popular tool for quick network monitoring and network debugging is tcpdump (www.tcpdump.org). Nowadays libpcap is developed in parallel with tcpdump.

Among others, these are the most important procedures of libpcap:

`pcap_open_live()` opens a new “capturing session” which is represented by a `struct` of type `pcap`. This `struct` is necessary for any further processing in the session.

`pcap_next()` sets a pointer to the next packet.

`pcap_loop()` binds a user defined function for packet data processing to a `pcap struct`. This function is called for every captured packet. This procedure can be used instead of `pcap_next()`.

`pcap_setfilter()` can be used to install a previously (with `pcap_compile()`) compiled filter to the “capturing session”. When supported (as under Linux and FreeBSD), the filtering mechanisms of the kernel are used for this.

Further procedures deal with writing captured packets to a file, reading and rewriting these files, with management of the “capturing session”, and statistics processing. For a detailed description of the procedures please refer to the manpage of libpcap (`man pcap`).

2.2 General Problems while Packet Capturing

This sections addresses the problems most relevant for packet capturing.

2.2.1 Receive Interrupt Load

As every received packet generates one interrupt, at high packet rates the incoming packets produce such a high interrupt load that the packet processing application does not get enough computing cycles to do its work or read the packets from the filter. This leads to full buffers and massive packet drops. In [MR97] the authors describe this problem called *receive livelock* in detail. To avoid receive livelock, Mogul and Ramakrishnan proposed the following solutions:

- *Using polling in overload situations to limit the interrupt arrival rate and returning to use interrupts afterwards.* Instead of generating one interrupt per packet the operating system frequently polls the network card for new packets, thus conserving resources for packet processing. More information on device

polling can be found in [SOK01, for Linux (NAPI)], [Riz01, for FreeBSD], [RD01, for Windows] and [Der04].

Another way of reducing the interrupt load is to do interrupt moderation which actually is nothing else than gathering some interrupts before originating one. This approach is available with the up-to-date Intel and Syskonnect Network Cards.

The disadvantage of both device polling and interrupt moderation is that the timestamping of the packets which is usually performed by the receiving interrupt in the kernel assigns the same timestamp to multiple packets. Thereby, the packet order might be falsely interpreted by the analyzing application. In any case, the timestamps of most packets and along with this the inter-packet gaps are not correct. To counter this problem, some network cards are able to timestamp the packets themselves.

- *Processing received packets to completion* to avoid buffering packets which are overwritten later by others or dropped because of full queues. This ensures that even under excessive load the packets that were captured are useful.
- *Explicitly regulating the CPU usage for packet processing to grant more processing time.* This can be achieved using systems trimmed for real-time operation (see [AGK⁺02, GAK⁺02, Kuh04]). This helps granting the kernel a unique utilization of the resources and therefore leads to better performance.

2.2.2 Packet Copy Operations

As described in Section 2.1, packet data is copied multiple times until it arrives at the application in user space. Copying packets within the kernel is expensive in terms of CPU cycles. Thus, reducing the number of copy operations improves the capturing capabilities.

In the FreeBSD capturing stack, the packet is copied three times:

- The first time from the network card to the main memory,
- then into the double-buffer behind the filtering engine,
- and finally from this buffer to the user space application.

There are several approaches to reduce these copy operations:

Linux avoids the second copy because there is no buffer behind the filtering engine. Instead, Packet data is located via a pointer in the queue behind the filter and delivered to user land.

Memory-mapped buffers can be used instead of the double-buffers or the kernel memory where the packets are kept. The principle of memory-mapping is to allow access from kernel as well as from user space to some piece of memory. When used as buffer, the access to this memory has to be shared. Therefore, they are often implemented as ring-buffers. This reduces the number of copies by one.

The measurements of Luca Deri for testing his ring buffer patch on the “capturing stack” of Linux [Der04, Der03] were only done on a single processor machine. Our experience is that this ring-buffer patch does not work stable with a multiprocessor kernel. Therefore, this patch will not be considered in this thesis.

2.2.3 I/O Throughput

When writing captured data to disk or when displaying per-packet information on a terminal which is probably accessed via some kind of remote login, a further performance bottleneck is added. As one can never capture more packets than the harddisk can write, it is crucial to consider this bottleneck. For example, terminals have a maximum printing rate which is easily exceeded with the bandwidth of a fully loaded Ethernet.

As experience has shown, even the PCI bus can be the bottleneck in a fully utilized Gigabit Ethernet environment, even though the theoretical maximum throughput suffices. Hence, it is necessary to use PCI enhancements like PCI-64bit, PCI-X, or PCIexpress in such situations.

2.3 Environment

As mentioned before, different machines are to be examined. One reason to purchase them was to evaluate their performance regarding packet capturing, as done in this thesis. Another reason was the intended purpose of these machines. In production deployment, the four double-processor computers are arranged for traffic capture at the edge of the scientific network of Munich (MWN) (see Section 4.1.4 for details). *snipe*, *swan*, *moorhen* and *flamingo* are the machines used for capturing. The captured data can then be used for traffic analysis. Another objective of these systems in regular operations is further development of intrusion detection systems. Thus, “production” is used in terms of academia here, mainly for research. The table in Figure 2.4 shows the differences between the machines.

All sniffers are equipped with 2 GBytes of RAM, an Intel 82544EI Gigabit (Fiber) Ethernet Controller, and a 3ware Inc. 7000 series ATA-100 Storage RAID-Controller with at least 450 GBytes of harddisk space attached. The Intel Xeon architecture machines are capable of Hyperthreading (see Section 6.3.7 for details).

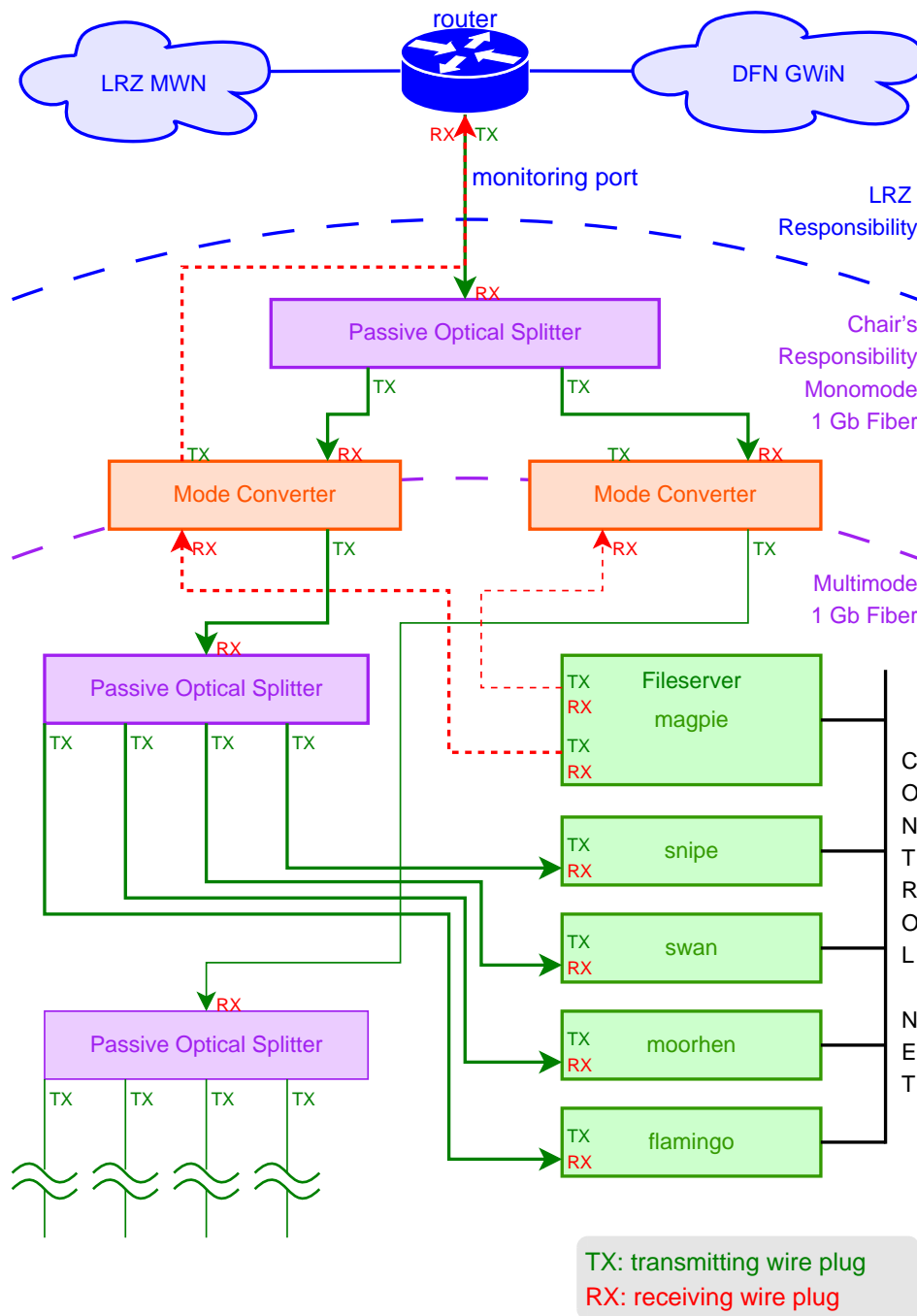


Figure 2.3: Network diagram of the regular setup of the capturing machines: The capturing interfaces of the different sniffers are attached to an optical splitter which distributes the input from a monitoring port at the uplink router of the LRZ.

Figure 2.4: The diversity of the sniffers.

Name	Architecture (Cache)	Chipset	OS
swan	AMD Opteron 244 (1024 kB)	AMD 8111	Linux 2.6.11.x
moorhen	AMD Opteron 244 (1024 kB)	AMD 8111	FreeBSD 5.4
flamingo	Intel Xeon 3.06GHz (512 kB)	SW GC-LE/CSB6	FreeBSD 5.4
snipe	Intel Xeon 3.06GHz (512 kB)	SW GC-LE/CSB6	Linux 2.6.11.x

Figure 2.3 demonstrates the regular setup opposed to the measurement setup for this thesis explained in Section 3.3. The network traffic obtained from the monitoring port at the edge router has to be duplicated several times. For that purpose optical splitters are used because their only influence is a reduced signal strength. This seems to be no problem, at least with the short cables that are used in this environment. Since there are no intelligent circuits or even electronic parts used as in a switch, the problem of timing issues, packet drops caused by full buffers, or a complete breakdown is avoided.

The optical splitters are used to ensure that each sniffer gets the same input. The mode converters are needed to transform the optical monomode⁴ signal from the monitoring port to a multimode⁴ signal for the capturing cards.

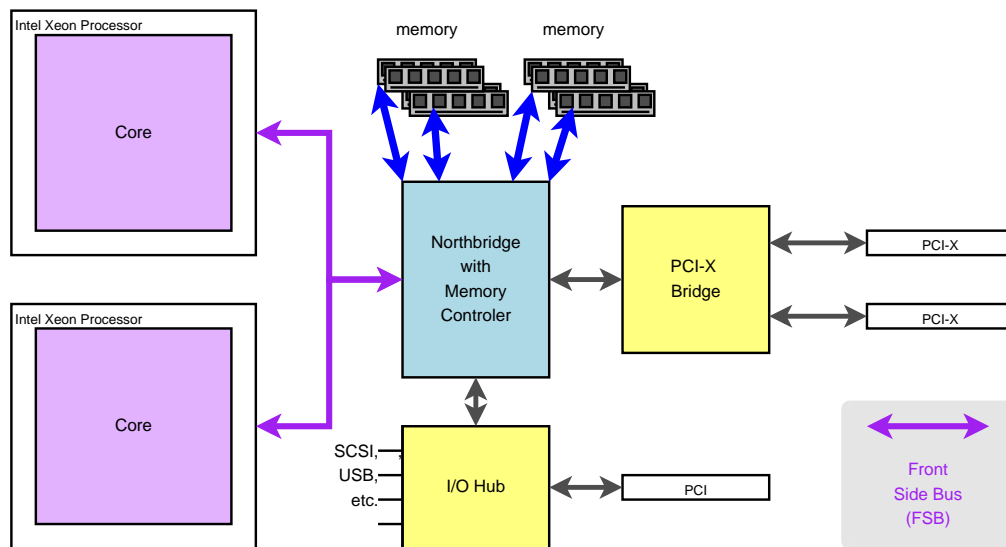
2.4 Architecture Comparison: Intel Xeon vs. AMD Opteron

When comparing systems with Intel Xeon and AMD Opteron processors, one has to understand the differences in the architecture of the compared systems. Since this should only be a short introduction, this section concentrates on how the processors are connected with each other, with the memory of the system, and with the rest of the computer. For supplementary information on the AMD64 architecture please refer to [Vil02] and for a detailed comparison of the architectures with benchmarks refer to [Vil03]. Figure 2.5 illustrates the differences using [Köh05] as basis.

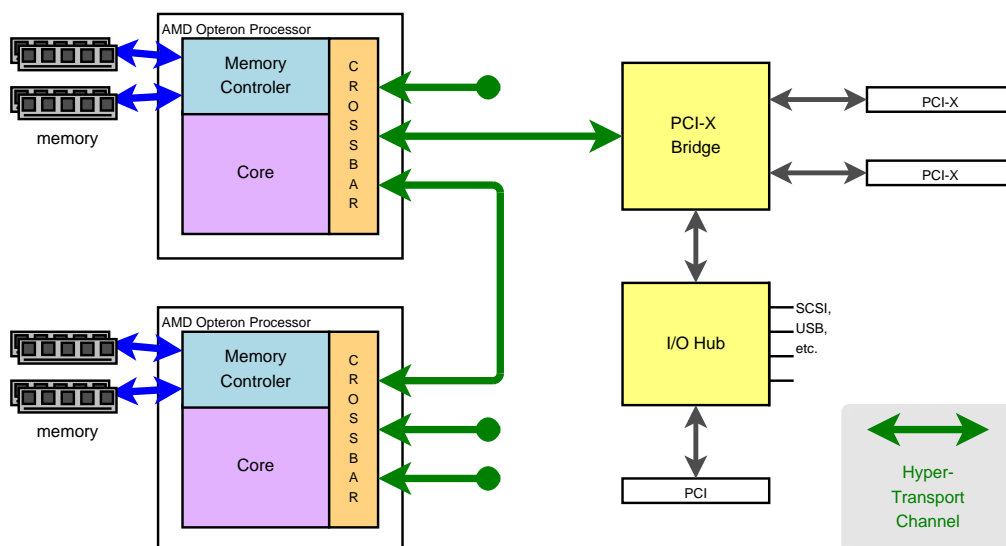
The main difference of the two designs is the way how the processors communicate with each other and the rest of the system. Where Intel uses an ordinary bus⁵ design, AMD comes up with point-to-point links of which three are available per processor. For the Intel Xeon processor (Figure 2.5(a)), this means that every memory access (the memory is attached to the Northbridge) must share the bandwidth of the front side bus (FSB) with any inter-processor communication and the normal I/O of the

⁴Fiber optic cables come in two different flavors: expensive *monomode* fiber for long distances and *multimode* fiber at lower price. The difference is that monomode fibers are that thin that the signal propagates in a straight line, thus prohibiting the usage of multiple modes (see [Tan03, Section 2.2.4, pages 93 et. seqq.] for details).

⁵A medium which is shared by all attached devices is called a *bus*. Therefore, it is necessary to separate the transfers from another. This design is fairly simple and well known.



(a) Intel Xeon



(b) AMD Opteron

Figure 2.5: Block diagrams of the designs of Intel Xeon 2.5(a) and AMD Opteron 2.5(b) for comparison: Every Xeon processor is connected via the Front Side Bus with the Northbridge and with each other processor, whereas every Opteron processor has three independent HyperTransport Channels to communicate with other processors or system components. In the Opteron design, the memory is directly attached to the processors which is not the case for the Xeons having their memory attached to the Northbridge.

system. Elsewise AMD (Figure 2.5(b)), where every processor has its own memory. This grants fast access even if all three communication channels (called HyperTransport) are busy. To avoid interfering with the processor operation (core) for memory transfers, the memory can be accessed from any other processor via crossbars.

The network interface cards used for our measurements are attached to the PCI-X bridge, because they need busses faster than standard PCI. Regarding packet capturing, both designs are equal in terms of copying hops from the network card to the main memory.

2.5 Network Traffic Characteristics

Since this thesis aims at a comparison of the performance of machines while packet capturing, it is necessary to provide traffic for that purpose. It is imprudent to use a random source for this objective because it has to match the characteristics of real network traffic. Therefore, it is inevitable to know these characteristics when searching for a suitable traffic source.

Research of local area networks (LANs) [LTWW93] as well as of wide area networks (WANs) [PF95] has shown that the traffic in common networks is best described as *self-similar*. This means that over a wide range of time-scales bursts can be identified. If, for example, the traffic is modelled as a Poisson arrival process, that would not be the case. Then the peaks would average out when looking at long enough time scales. This is not the case for real network traffic as [CB96, page 5, figure 2] and [WTSW97, page 24, figure 7] show. Network traffic is considered with fractal properties: Regardless of the scale similar structures can be observed. As reasons for this behavior [CB96] mentions file size distributions and superposition of many traffic sources. These sources depend on user bearing along which caching mechanisms. A more formal analysis of Internet traffic can be found in [CLS00, WTSW97].

The impact of these and other understandings in the field of network traffic characteristics is explained outstandingly in [FP01] in the context of simulating networks. When trying to simulate network traffic as demanded by the topic of this thesis, it is important to know about the above mentioned properties.

The subject of this thesis is to identify a way to capture all packets in transit from a link. When a system cannot keep up with bursts of the data rate, usually buffers are used to absorb peak rates. Thus, the system is for a while able to handle higher data rates than the average rate it can usually deal with. But keeping self-similarity in mind, for every imaginable buffer size there will be a long enough burst of the data rate to completely consume the available buffer space. So it is desirable to be able to permanently capture higher data rates than the average data rate of the traffic imposed on the systems. Thus, the intension is to find out how to capture packets at the highest possible data rate.

3 Methodology

Before performing any measurements, it is necessary to build up a testing environment. Therefore, it is essential to define the requirements for this setup first. Then it has to be evaluated how the demands can be fulfilled.

3.1 Items in the Testing Environment

The influence of different factors on the capturing performance is to be evaluated. This thesis considers commodity systems only. But commodity hardware comes in different flavors with respect to processor architecture. There are IBM, Sun, Intel and AMD producing processors—just to mention the popular ones. Furthermore, there is the option to build multiprocessor systems or not, and a whole bunch of mainboards is available. The systems were chosen to represent the two most common dual processor architectures: AMD Opteron and Intel Xeon. This was done to be able to evaluate the benefit or penalty of multi-processor systems. To conduct a fair comparison, all systems have been purchased at the same time with comparable components and capacities. Using different architectures covers one major factor: the physical system throughput.

The other major factor can be covered by using different operating systems. Although there are many other operating system available like Windows, Solaris and MacOS X, this thesis concentrates on Linux and FreeBSD. There are two reasons for choosing the two: (i) these systems are the most popular ones in terms of packet capturing and (ii) they are Open Source and free of charge, thus everyone can use them. To be able to evaluate all operating system / architecture combinations, four systems are used in parallel for each measurement.

Besides of these *Systems under Test (SUTs)* whose details are given in Section 2.3 and Figure 2.4, the following components are required for the intended measurements:

Workload Since packet capturing performance is to be tested, a source for the traffic which shall be captured is to be set up. This traffic has to be provided to all SUTs.

Transmission Media Because multiple systems have to be supplied with the same packets, it is necessary to find an appropriate connection of the source and the SUTs.

Capturing Application The traffic has to be captured. Therefore an application, which uses the same mechanisms as common network monitoring tools, has to be used. Furthermore, this can be used as a means of determining the performance.

Profiling The utilization ratio of the resources of the SUTs is helpful for an accurate evaluation. Therefore, an instrument for monitoring the CPU usage of the capturing machines is required.

3.2 Requirements

All of the above mentioned items have special requirements. Many of them have been identified during the project work described in [Sch04]. These will be discussed in the remainder of this section.

Systems under Test

The systems which were chosen for the measurement had to fulfill the following requirements.

- Different processor architectures are to be represented. In terms of architectures, AMD Opteron and Intel Xeon Systems are considered.
- Linux and FreeBSD, the two operating systems which are to be investigated, must run on these architectures.
- The network interface card used for capturing should be well supported by the considered operating systems. The Intel fiber Gigabit cards were chosen. Preliminary investigations have shown that this card is among the fastest well supported commodity cards.
- The PCI bus should be at least PCI-64bit aware for high throughput of the network cards. In fact, our systems can handle PCI-X as well but there are no commodity cards available for this bus.

Workload

Finding an appropriate traffic source for Gigabit Ethernet environments is a non-trivial task. The difficulty is to generate a reproducible sequence of packets at high speeds which is similar to real network traffic. In [Sch04] the Linux Kernel Packet Generator was utilized with different packet sizes ranging from 64 to 1500 bytes. But common network traffic usually consist of packets of the various sizes. So, the following three requirements have to be fulfilled:

Speed The tool should be able to generate packets nearly at line speed (1 Gbit/s). If this requirement cannot be fulfilled, the load imposed on the capturing machines is not high enough to drop any packets. Therefore, a system bus faster than PCI is required. The throughput of standard PCI (32 bit, 33 Mhz) is at a theoretical maximum of 133 Mbytes/s which would just be enough for 1 Gbit/s. But taking into account that normally more than one device is attached to the PCI bus and that the capacity is shared by the devices, the effective throughput is reduced.

Realness The input for the SUT's should be as similar as possible to real network traffic because, in general, it is necessary to capture genuine and not artificial traffic. The resemblance to real traffic should at least be satisfied with respect to the packet size distribution. This is important because different packet sizes produce distinct packet inter-arrival times, and different packet size distributions generate varying packet rates (measured in packets per second—pps). This packet rate determines the interrupt rate which is directly influencing the capturing performance as described in Section 2.2. The type and content of the packets have no influence on the process of capturing. But since the sizes of the packets influence the required memory and the required costs for analysis, the packet size distribution should be as real as possible.

Reproducibility The sequence of packets should be identical across different measurements, especially with respect to the above mentioned requirements on realness. This is due to the fact that, otherwise, exceptional behavior or failures are not reproducible and, therefore, cannot be examined in detail.

Transmission Media

Besides of actually transporting the generated traffic to the systems under test, it is necessary to supply the packets to all four capturing machines. Therefore, the traffic has to be duplicated several times. Based on our positive experience with passive optical splitters, they are used in this work.

In addition, an opportunity to monitor the traffic source in terms of number of generated packets is convenient to verify that all generated packets are indeed sent over the fiber.

Capturing Application

The capturing application has to be highly configurable to allow a wide range of possible settings. For example, it should be possible to apply filter expressions and configure additional load to simulate traffic analysis tools. In any case, it has to produce statistics on the number of packets received. To ensure compatibility and portability as well as relevance for real monitoring tools, the application needs to be build on top of libpcap.

Profiling

Regarding profiling, only two requirements have to be met. It has to be easy to use, and its impact on the system load should be small.

3.3 Test Setup

Considering these requirements a commodity system for workload generation, a switch, an optical splitter, the systems under test and a lot of wires were put together and form the test setup. Workload generation (Chapter 4) and profiling (Chapter 5) as well as the capturing application (Section A.1) are discussed separately.

The test setup shown in Figure 3.1 is derived from the existing infrastructure (Figure 2.3). All dispensable links were removed leaving the shown devices. The monitoring link of the original setup described in Section 2.3 was replaced by a workload generator. The requirement of counting the number of generated packets is fulfilled by using a Gigabit (fiber) Ethernet Switch (Cisco C3500XL). The optical splitter is now attached to a port of the switch configured to monitor the input from the workload source.

The workload generator, called `gen` is a double processor AMD Athlon MP 2000 with a PCI-64bit bus (with 133 Mhz frequency) and a Syskonnect SK-98xx Gigabit (fiber) Ethernet controller. It is used to generate packets which can be captured simultaneously by all four sniffers to see which one loses how many packets. The second network card in `gen` is used to perform SNMP¹ requests for the packet counters of the switch. To avoid seeing these SNMP packets at the sniffers, a VLAN² was configured for the two data ports (input from `gen` and output to the splitter).

3.4 Measurement Cycle

To perform the measurements, it is necessary to synchronize the sender with the receivers. For this task a host within the control net (Figure 3.1) was chosen. This host performs the following steps as shown in Figure 3.2:

1. Login to the four sniffers to start the capturing and profiling applications. Save the process ID's of these applications to designated files. Wait for all sniffers to become ready.
2. Login to `gen` to read the SNMP packet counters of the switch.

¹*SNMP* is the simple network management protocol, which standardizes the communication between different network devices.

²A *VLAN* (Virtual LAN) separates different ports of a switch into different network segments. Usually there is no guaranty that packets arriving at a certain switch port are not sent out on another port. See [Tan03, Section 4.7.6, pages 328 et seqq.] for details on VLANs.

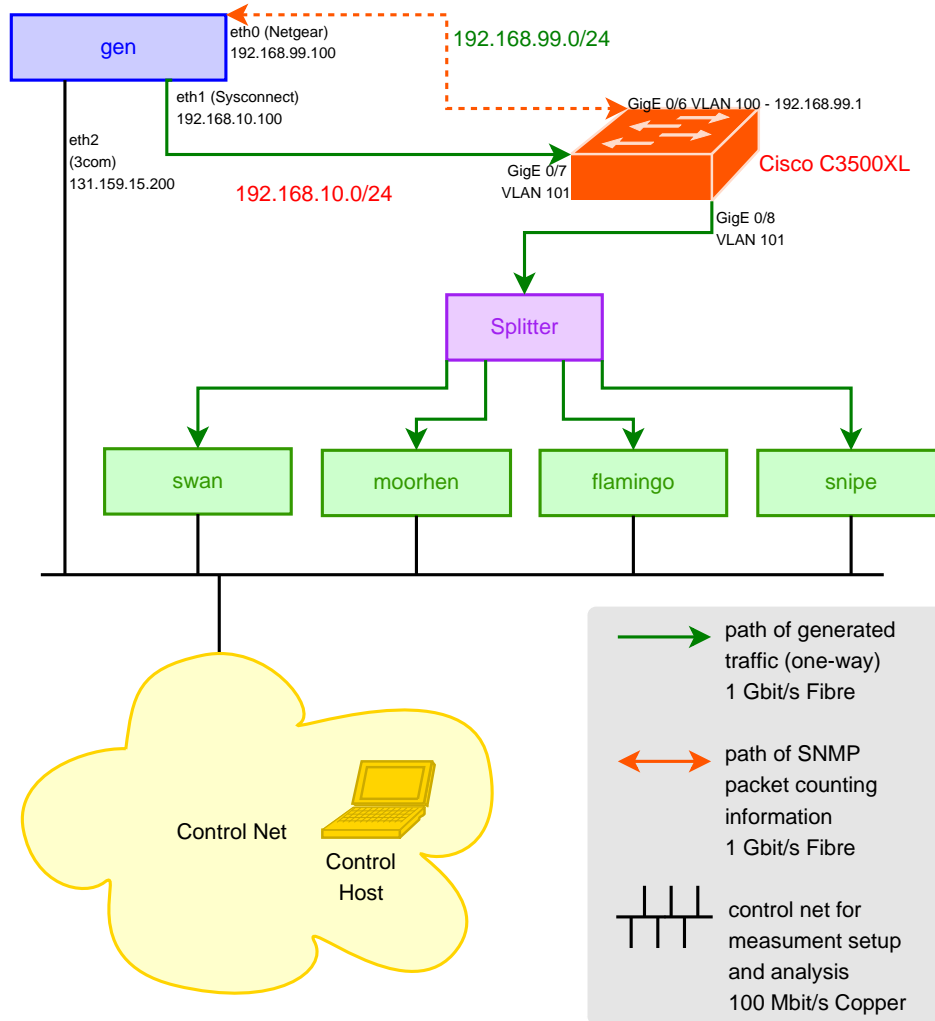


Figure 3.1: Network diagram of the setup of the capturing machines for measurements: Instead of feeding the splitter with real network traffic as shown in Figure 2.3 another computer (*gen*) is used to generate the packets which are to be sniffed.

3. Login to *gen* to start the packet generation. Proceed if the generation is finished and the statistics are saved.
4. Login to *gen* to read the SNMP packet counters of the switch.
5. Login to the four sniffers to stop the applications using the saved process ID's.

Again, when all sniffers are done we can start a new measurement. To avoid strange behavior, this procedure is repeated several times.

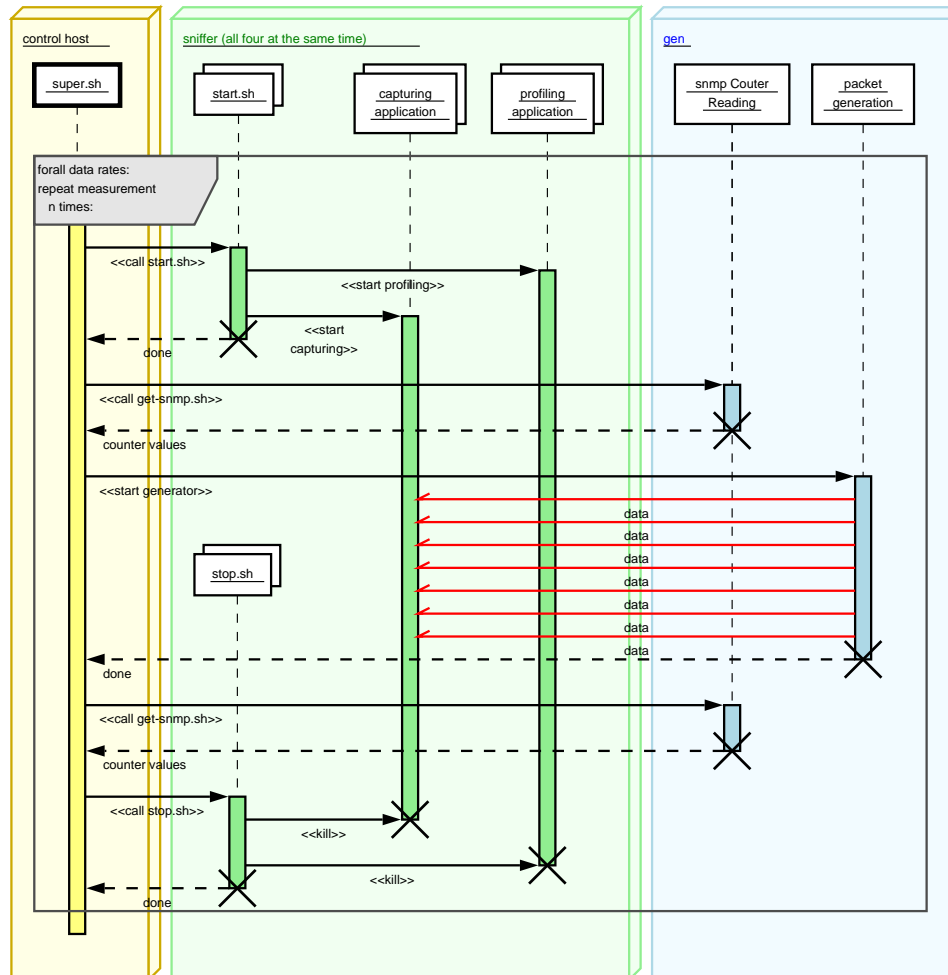


Figure 3.2: Sequence diagram of the measurement cycle: For each measured data rate, this measurement cycle is repeated seven times to avoid outliers or unwanted influences. The action happens at three places: The host to control the measurement, the sniffers, and the generator. The red lines represent the generated packets.

4 Workload generation

The task of finding a proper tool for workload generation appears to be easy at first glance. But when exploring the range of available tools for packet generation, none of them matches the requirements from Section 3.2. A solution to this problem is presented in this chapter.

4.1 Existing Tools for Packet Generation

The following sections outline the advantages and disadvantages of existing free tools for generating packets.

4.1.1 TCPivo/NetVCR and tcpreplay

With reference to realness and reproducibility the best choice would be a program which replays a previously captured trace¹ from a real network. Such tools include TCPivo/NetVCR² [FGB⁺03] and tcpreplay [Tur04].

Replaying a trace of real network data meets the requirements of realness and reproducibility easily. The problem with these tools is the low maximum data rate they can produce. As Sebastian Lange showed in [Lan04, page 15 et seqq., figure 7 and table 2] the maximum transfer rate achievable with these tools is about 480 Mbit/s. Admittedly, the computer he used for generation had only a standard PCI bus. Yet, a test measurement using gen reproduced his results (~ 476 Mbit/s). This shows that such tools are too slow to generate traffic at line speed at the moment. In addition, the mentioned data generation rate is only possible with maximum sized packets.

4.1.2 “Monkey See, Monkey Do” and Harpoon

Monkey See, Monkey Do—described in [CHC⁺04]—is a tool for replaying live and online captured traffic from real networks into a test network for evaluating new network components or applications under real conditions. Since the output is the same as the input, another traffic source is needed. Furthermore, it does not focus

¹A collection of packets sniffed from a link is called a *trace*. Generally the packets of such traces are in order as captured from the link and provided with a timestamp.

²The original name was NetVCR, but it was renamed to TCPivo due to name registration problems.

on speed as it only replays what it captures. Therefore, it is no option for this work. The same holds for Harpoon [SB04] which does nearly the same, except that the output is generate based on statistics gathered from real networks.

4.1.3 Linux Kernel Packet Generator

The Linux Kernel Packet Generator (pktgen) [Ols02] is a Linux kernel module which can be used to generate UDP packets for network testing. It can be controlled via the `/proc` filesystem. Generated packets are sent as fast as the kernel is able.

In contrast to the previously mentioned tools, the main advantage of the Linux Kernel Packet Generator is the speed with which it is generating packets. Data rates of up to about 938 Mbit/s on Gigabit Ethernet, which is nearly line speed considering Ethernet preamble, checksum, and inter-packet gaps, are achievable with a Syskonnect network card using 1500 bytes sized packets. It is worth noting that the same measurement using a Netgear card results in 930 Mbit/s or 890 Mbit/s using an Intel network card.

As mentioned in Section 3.2, the main disadvantage is its unhandiness since only one packet size can be configured. Mixing different instances of these kernel modules to produce different packet sizes is not possible without risking non reproducible data streams.

Besides these shortcomings, there are other interesting features. It is possible to configure an artificial inter-packet gap, setting the source and destination IP's and MAC addresses, as well as cycling through such addresses.

4.1.4 Live Data

Our research group has the possibility to capture packets crossing the uplink of “a powerful communications infrastructure called the Munich Scientific Network (Münchner Wissenschaftsnetz, MWN)” (from [LRZ]) operated by the Leibniz Computing Center, a “joint computing center for research and education for all Munich universities” (from [LRZ]). This uplink connects the MWN with the G-WiN, “Germany’s National Research and Education Network. It provides a high-performance infrastructure for the German Research and Education Community. [...] Being connected to the European Backbone Géant, G-WiN is an integral part of the worldwide community of research and education networks.” (from [DFN]).

The utilization of this Gigabit link ranges from about 220 Mbit/s (90 Mbit/s downstream and 130 Mbit/s upstream) to about 1200 Mbit/s (400 Mbit/s downstream; 800 Mbit/s upstream) at peak times. Data rates greater than 1024 Mbit/s are possible because this link is bi-directional, thus allowing 1024 Mbit/s inbound and 1024 Mbit/s outbound at the same time. This traffic is produced by 50 000 hosts.

The average rate (about 400 Mbit/s) is not high enough for the requirements of this thesis. But the systems must be able to capture and process the load of the link in production use. Among the applications running on the sniffers are Bro [LBN, Pax99], a free intrusion detection system for scientific research and further development, and a “time machine”³ [KPD⁺05, Kor05] for efficient recording and retrieval of high-volume network traffic.

But the biggest problem with this workload “source” is its non-deterministic output with respect to data and packet rate. This makes it really difficult to conduct comparable measurements, not to mention reproducible measurements. Recapitulating, this source is totally inappropriate for performance measurements.

4.1.5 Summary

None of the available tools or sources is able to satisfy all of the requirements. Therefore, an existing tool has to be extended. The Linux Kernel Packet Generator is chosen as basis. The ability to generate packets of different sizes as well as a mechanism to feed a distribution of packet sizes to the generator has to be added.

4.2 Identification of Packet Size Distributions

Since the Linux Kernel Packet Generator has to be enhanced to generate packets of different sizes, the goal of this section is to examine the nature of real packet size distributions. First the characteristics of these distributions have to be identified before a representation model can be developed.

4.2.1 Analysis of Existent Traces

The first step is to identify the characteristics of packet size distributions of real network traffic. A short existing trace is analyzed with `ipsumdump` [Koh] to extract the packet sizes which are grouped by packet size and counted with an `awk` script. Comparing these results with [CMT98, page 8, figure 3(a)] shows that in our environment the jumbo-frames⁴ of Gigabit Ethernet do not exist at all (compared to some few in [CMT98]). The distribution of packet sizes between 0 and 1500 Bytes is similar to [CMT98].

For larger trace files this proceeding proved to be very slow. Therefore, a small application in C was written to speed up the reading of trace files (see Section A.1

³The idea behind this *time machine* is to supply a mechanism for intrusion detection systems, to access interesting packets when they already have been transferred. If such a mechanism can be utilized, the intrusion detection system can apply a much stricter filter which reduces its load.

⁴Ethernet frames are limited in size. The maximum size is 1500 Bytes. With the new Gigabit Ethernet specification so called *jumbo-frames* were created to raise this limit.

for details on this program). With the help of the new tool a twenty-four hour trace obtained at the uplink of the scientific network in Munich is examined to enlarge the data basis for the following calculations. The results are similar to those of the above mentioned trace. But at a different scale. The results are depicted in Figure 4.1 and it shows the usual peaks at 40–64, 552, 576 and 1420–1500 bytes.

4.2.2 Representing Packet Size Distributions

Because of the high packet rates that are to be generated, the representation of these kinds of distributions needs to be efficient but accurate. Using hash tables⁵ is too expensive because the size for each packet has to be looked up.

The goal is to match the sizes appearing often in the distribution with high accuracy. Unfrequent sizes do not need to be as accurate because more than three-quarter of the packets appear in the twenty most common packet sizes, see Figure 4.2. Using this insight, a two-stage system for representing such a packet size distribution is used. The desired procedure to achieve the packet size for the next packet is shown in Figure 4.3.

Needing a fast access method during the generation, the usage of simple arrays was chosen. Their content will be the packet size for the packets to generate. The variability comes in with the index to the entries of these arrays. The plan is to calculate a random number as index to the array entries and return the packet size of the indexed position.

The first stage uses the exact packet sizes with the probabilities of the heavy hitters (outliers). In our example, the heavy hitters are the twenty most often counted packet sizes of Figure 4.2. The second stage uses bins of configurable size in which multiple packet sizes are combined. These bins receive the sum of all included packet size probabilities as their own probability. This averages the diversity of the probabilities of the included packet sizes out, thus leading to a somewhat inaccurate output distribution. But it reduces the amount of memory needed and the time required to find the next packet size to a feasible extend. To allow the representation to be flexible, the following customizable parameters are used:

precision (ρ) This value can be used to change the size of the arrays. Smaller (larger) arrays lead to lower (higher) precision of the represented distribution. The default value is 1000.

⁵A *hash* or *hash table* is a data structure which allows to save items with a large key space to a memory as large as needed to contain the items. In general a hashing function is used to calculate an index from the key of the item. This index is shorter than the original key in terms of binary digits. See [CLRS03, chapter 11, pages 221 et seqq.] for details.

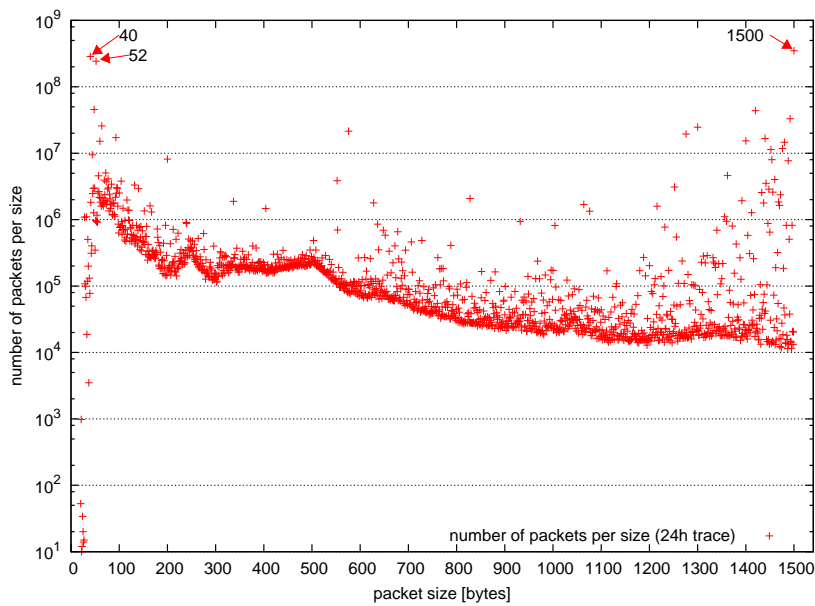


Figure 4.1: Scatterplot of an example distribution of packet sizes (y-axis in logscale): The most frequent sizes can be identified at 40, 52 and 1500 bytes.

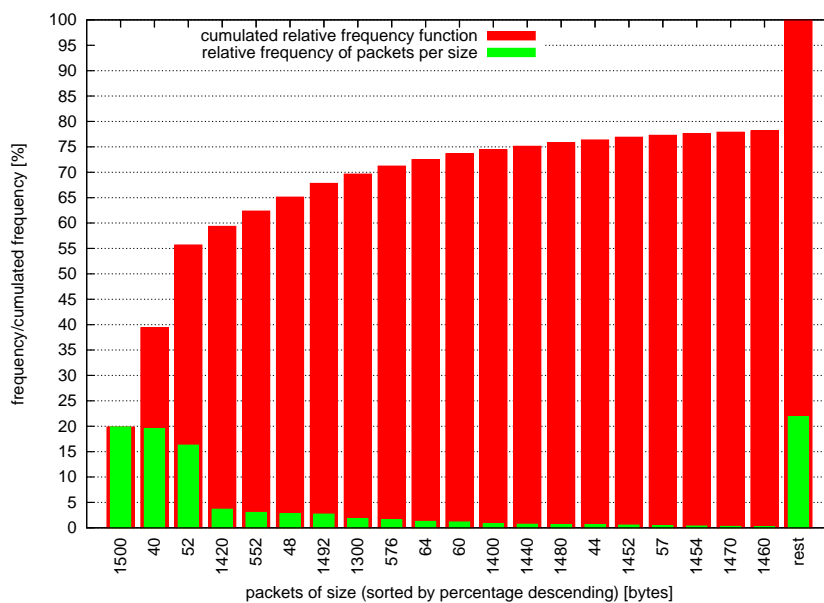


Figure 4.2: Histogram plot of the percentages for the example distribution: Each packet size from Figure 4.1 is shown with its fraction (green) of all packets. This plot also shows the cumulative sum (red) of the decreasingly sorted percentages from left to right. The three most frequently appearing packet sizes represent more than 55% of all packets, and the top 20 packet sizes account for over 75% of all packets.

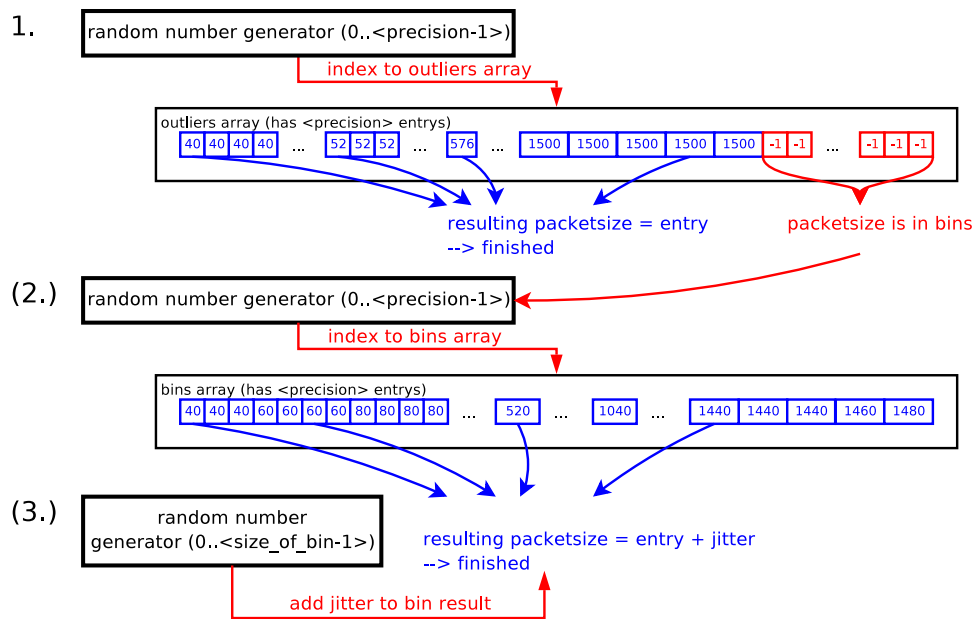


Figure 4.3: Flow diagram: This figure shows how a new packet size is selected using the arrays. First an entry is chosen randomly from the outliers array. If the acquired packet size (the value of the indexed position of the outliers array) is equal to -1 the searched packet size was not yet found. Therefore, an entry from the bins array has to be selected randomly and a jitter (with a randomly chosen amount between 0 and $\sigma_{\text{bin}} - 1$) is added to the result to obtain the packet size.

maximum packet size (N_{ps}) The biggest packet size which will be considered by the distribution can be configured with this parameter. Its value defaults to 1500.

outliers boundary ($p_{\Omega_{\text{bound}}}$) This specifies how many packets relative to the total number of considered packets must be of same size, so that this packet size is associated with the first stage array. If this lower bound (default: 2%) is exceeded, this packet size is called an outlier. Taking the top n most frequently used packet sizes would also be possible but requires maintaining a list.

binsize (σ_{bin}) This value specifies how many sequential packet sizes are merged into one bin of the second stage array. Default: 20.

(number of bins) (n_{bin}) n_{bin} depends on σ_{bin} and N_{ps} : $n_{\text{bin}} = \left\lceil \frac{N_{ps}}{\sigma_{\text{bin}}} \right\rceil$

4.2.3 Calculation of the Resulting Distribution

This section explains the exact manner in which the arrays described in Section 4.2.2 are computed. The numbers c_i of packets of all sizes, where $0 < i < N_{ps}$ is the packet size, are assumed to be known as initial values. These values are obtained from a trace or a live capture. Furthermore, the number of all packets c_{all} can either be calculated or given as well. The following steps are performed:

1. The fractions p_i are calculated from the number of packets per size c_i and the total number of packets c_{all} :

$$\forall i: \quad p_i = \frac{c_i}{c_{all}} \quad (4.1)$$

2. The set Ω of packet sizes which are outliers (first stage items) is identified, along with the number n_Ω of outliers:

$$\Omega = \{i \mid p_i \geq p_{\Omega\text{bound}}\} \quad n_\Omega = |\Omega| \quad (4.2)$$

3. The bins—indexed by j —are generated (second stage items):

$$\forall j \in \{1, \dots, n_{\text{bin}}\}: \Omega_j = \{i \mid j \cdot \sigma_{\text{bin}} \leq i < (j+1) \cdot \sigma_{\text{bin}} \wedge i \in \Omega\} \quad (4.3)$$

$$n_{\Omega_j} = |\Omega_j| \quad (4.4)$$

$$b_j = \sum_{\substack{j \cdot \sigma_{\text{bin}} \leq i < (j+1) \cdot \sigma_{\text{bin}} \\ i \notin \Omega}} c_i \quad (4.5)$$

$$avg_j = \frac{b_j}{\sigma_{\text{bin}} - n_{\Omega_j}} \quad (4.6)$$

$$\forall i \in \Omega_j: c_i = c_i - avg_j \quad (4.7)$$

$$b_j = b_j + avg_j \quad (4.8)$$

Before calculating the probabilities of any bin, the set Ω_j of outliers for each bin have to be identified (4.3) and the number n_{Ω_j} of outliers in this set is counted (4.4). Within each of the bins the numbers c_i of packets of size i are summed up in b_j if the recent packet size is no outlier (4.5). Afterwards, the average frequency avg_j of the bin is calculated (4.6) and subtracted from the frequency of all corresponding outliers (4.7). Likewise, it has to be added to the frequency of the bin as many times as outliers have been found in the bin (4.8).

4. Now the probability of all outliers and bins can be calculated:

$$\begin{aligned} \forall i \in \Omega: \quad p_i^{\text{outl}} &= \frac{c_i}{c_{all}} \\ \forall j \in \{1, \dots, n_{\text{bin}}\}: \quad p_j^{\text{bin}} &= \frac{b_j}{c_{all}} \end{aligned} \quad (4.9)$$

Because it is not possible to work with floating point numbers within the kernel, the Linux Kernel Packet Generator cannot be “fed” with floats. Therefore, this is done in user space while the following step transforms the floats into integers. This will produce arrays with packet sizes for randomized access:

5. The arrays described in Section 4.2.2 are generated with the number of cells f_{index}^{type} which have to be filled with the same packet size:

$$\begin{aligned} \forall i \in \Omega : \quad f_i^{outl} &= \rho \cdot p_i^{outl} \\ \forall j \in \{1, \dots, n_{bin}\} : \quad f_j^{bin} &= \rho \cdot p_j^{bin} \end{aligned} \quad (4.10)$$

As shown in Figure 4.4 the distribution of the generated packet sizes is not perfect but matches the heavy hitters well. This can be easily recognized in the magnifications of Figure 4.4(b), where the heavy hitters are located on the left. Figure 4.4(a) shows clearly the bins of the generation process. But it is close to the original distribution. For reasons of simplicity, the random numbers generated to index the arrays and to produce the jitter are equally distributed. This leads to similar frequencies for packet sizes in the same bin.

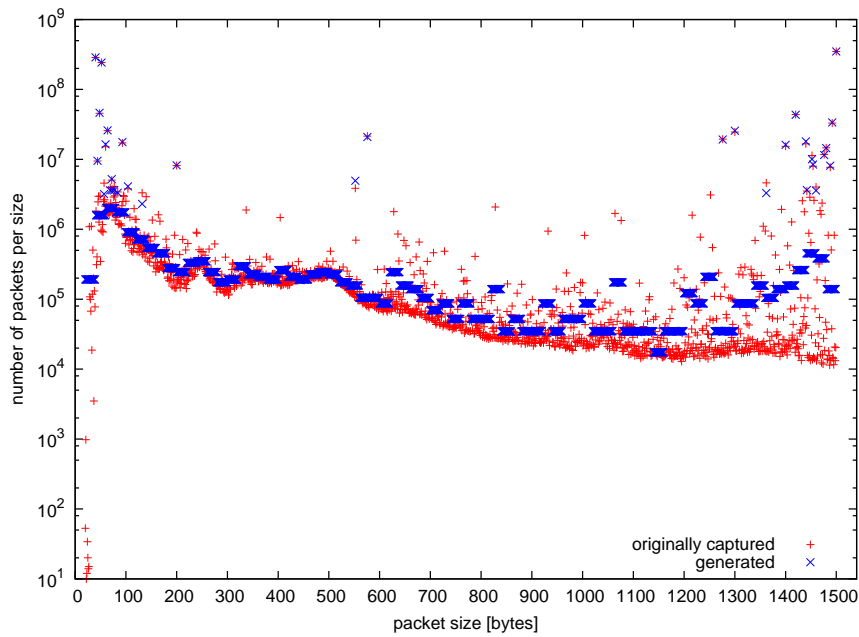
4.3 Enhancement of the Linux Kernel Packet Generator

The computation of the input—the arrays from the last section—for the enhanced packet generator is done by the same application (`createDist`, Section A.1) that is used for counting the different packet sizes.

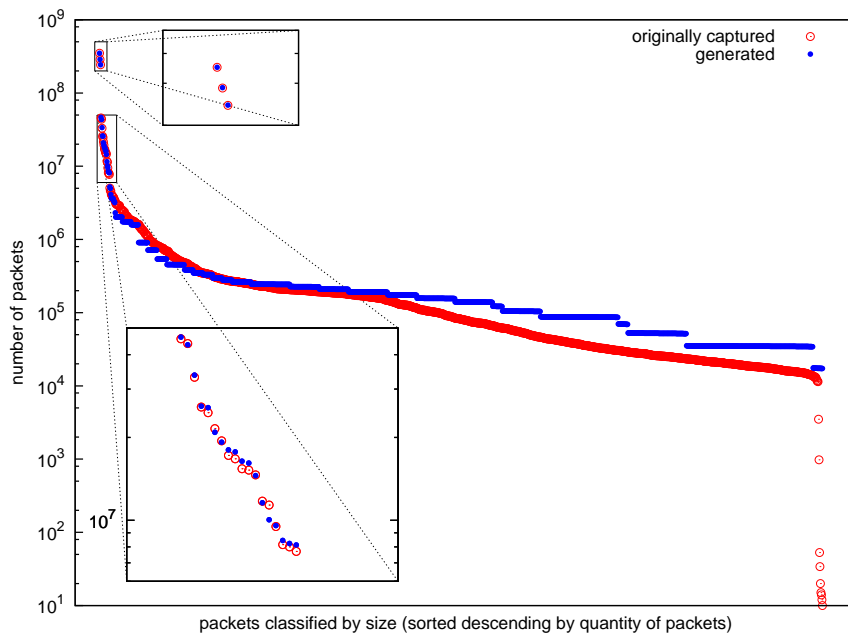
To use these packet size distributions, it is necessary to change the source code of the Linux Kernel Packet Generator module [Ols02]. For a detailed list of changes as well as information on using, compiling, and obtaining this changes please refer to Section A.2.

4.3.1 Results

With a sample distribution similar to the one shown in Figure 4.1 packet rates of about 180.000 pps and data rates of about 915 Mbit/s (as shown in Figure 4.5) can be achieved. This corresponds to a nearly fully utilized Gigabit Ethernet link. The drop in the generated bandwidth—when compared to maximum sized packets—is negligible when considering the benefit of having a “real” packet size distribution. Another pleasing side effect in this comparison is the doubled packet rate. Since higher packet rates require faster interrupt processing and higher data rates require higher system throughput, both rates should be high to impose a remarkable load on the systems under test.



(a) Scatterplot (y-axis in logscale): packet size against frequency



(b) Frequency plot (y-axis in logscale): All packets of the same size are joined in a class. These classes were sorted descending by the quantity of packets belonging to it. The discrepancy comes from averaging the frequency in bins.

Figure 4.4: These plots show the original (captured) distribution of packet sizes and the distribution of the generated packet sizes.

This new method of generating packets meets all goals mentioned in Section 3.2. So the demands of this part are fulfilled. But the results also show that when high packet rates are needed it is better to use equally and minimum sized packets.

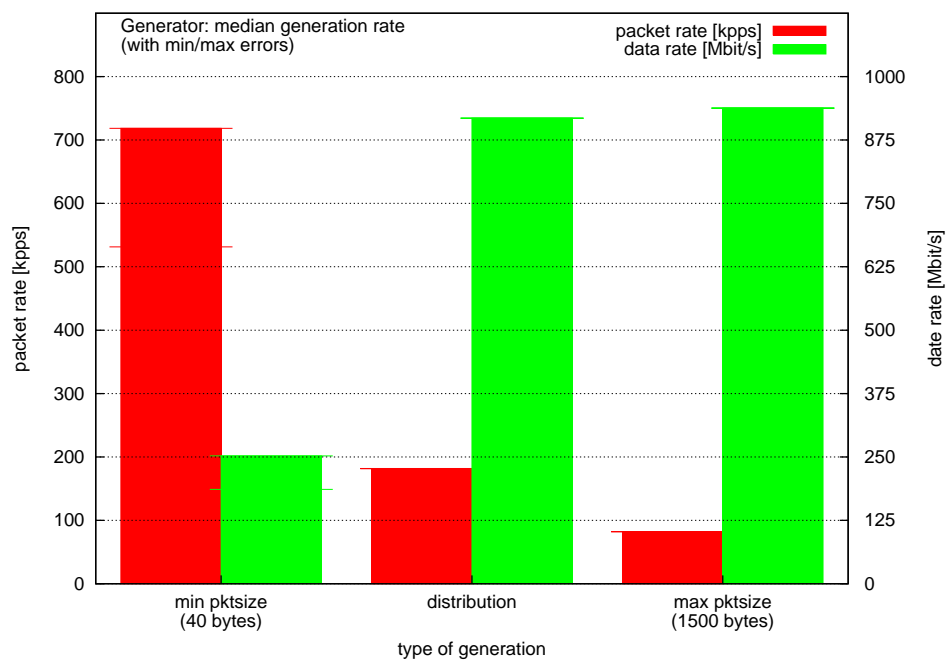


Figure 4.5: Histogram the packet and data rates generated by the Linux Kernel Packet Generator: This figure shows how the generator performs with minimum and maximum sized packets in comparison to the new packet size distribution mechanism.

5 Profiling

One of the performance bottlenecks is the processing power of the machines performing the packet capturing. On that account, it is interesting to know the utilization ratio of the processors during packet capture. Therefore, some sort of profiling is needed. At least the overall CPU usage should be recorded to be able to tell if the system was fully loaded or not. In this chapter the tool used for this task is presented.

5.1 cpusage

In order to be able to monitor the CPU load while capturing packets, a small application has been written in C, called `cpusage` (Section A.3). It reads out the systems CPU accounting information. These are counters of how many tics the CPU spends in one of the following modes:

- user applications,
- niced (low priority) applications,
- system/kernel,
- idle,
- (Linux only:) input or output,
- interrupts,
- (Linux only:) soft-interrupts.

The total number of tics spend can be computed as the sum of all this counters. The modes marked as “Linux only” are counted as system/kernel or interrupts under FreeBSD.

The problem is that these counters are started at boot time and cannot be reset. Hence, the percentages cannot be calculated from a single readout. Instead, the counters have to be read twice. Then the differences between corresponding counter values yield the result. The percentages are acquired by dividing the differences by the sum of all differences.

Finally, the percentages for each mode are available. This is the same way how `top` [War02] operates. The difference between Linux and FreeBSD is the way the

```

:
user: 35.2%, nice: 0.0%, system: 3.1%, idle: 38.3%, interrupt: 23.4%,
user: 32.8%, nice: 0.0%, system: 1.6%, idle: 35.9%, interrupt: 29.7%,
user: 32.3%, nice: 0.0%, system: 1.5%, idle: 41.5%, interrupt: 24.6%,
user: 10.9%, nice: 0.0%, system: 1.6%, idle: 68.0%, interrupt: 19.5%,
user: 32.0%, nice: 0.0%, system: 3.1%, idle: 32.0%, interrupt: 32.8%,
user: 33.9%, nice: 0.0%, system: 0.0%, idle: 39.5%, interrupt: 26.6%,
---Summary---
Min: user: 10.9%, nice: 0.0%, system: 0.0%, idle: 32.0%, interrupt: 19.5%,
Max: user: 35.2%, nice: 0.0%, system: 3.1%, idle: 68.0%, interrupt: 32.8%,
Avg: user: 28.4%, nice: 0.0%, system: 1.6%, idle: 43.4%, interrupt: 26.6%,

```

Figure 5.1: Sample output of the `cpusage` application.

counters data is accessed. (Under FreeBSD `sysctl`¹ has to be used instead of a read to `/proc/stat`¹ under Linux.)

`cpusage` reads the counters twice a second keeping the last readout at any one time for calculation and writes the calculated percentages to a file. This is like using `top`, with the improvement of being able to go back in time. Another advantage compared with `top` is a built-in minimum, maximum, and average percentage calculation during the measurement. The periods of measurement or, in other words, the periods when packets are captured, are correlated with periods of heavy load. Thus, `cpusage` assumes packet capturing if the load exceeds a configurable threshold.

When trying to find out the average CPU usage for the time when the machine is capturing packets, the start and end times are difficult to define. According to the measurement procedure explained in Section 3.4 the profiling application is started far before the the packets are generated. Thus, the average accounting is started when the idle percentage drops below a user defined limit, and stopped when the same limit is penetrated again. A sample output is shown in Figure 5.1.

5.2 Postprocessing of the `cpusage` Results

In some measurements the calculated average values seem to be incorrect, because the value is too low to fit with the other comparative values. This happens, because `cpusage` reacts the first time the idle value falls below the user defined threshold. If this is not caused by the capturing, the period determined for the average usage during the capturing is incorrect.

¹A `sysctl` under FreeBSD as well as the `/proc` filesystem under Linux are used to get and set kernel parameters.

To solve this problem, a small awk script `trimusage.awk` (see Section A.4) was written which reads the unaggregated data (the lines before `---Summary---` from Figure 5.1) and calculates the correct average values. To acquire the wanted result, the script reads all input lines and builds periods of times when the idle value is under-usage. When all lines have been read, the longest period is chosen as the one caused by the capturing. Then the average value is calculated from this block of lines.

For the plots in the following chapter these values are used, but the original is kept as well for supervision.

6 Measurement and Results

Before actually describing the results of the measurements, this chapter records the factors which influence capturing performance. Furthermore, the measurement procedure which is identical for all experiments is explained in detail. At last the results of the different measurements are illustrated.

6.1 Influencing Variables

Although the measurement categories are just the capturing rate and the CPU usage, there are several different factors or parameters influencing the performance of packet capturing. Some of them are hardware bound and therefore difficult to change. The factors being subject are listed as follows:

Symmetric Multiprocessing can be turned off by using a kernel which does not support multiple processors. This requires having different kernels in terms of compile time options and rebooting to change between them. The goal is to detect the impact of using one or two processors and of using Hyperthreading (where available). To activate or deactivate Hyperthreading it is usually necessary to change a BIOS parameter.

Operating systems, which are investigated, are Linux and FreeBSD. Both of them are free of charge and Open Source projects. They are installed on the four systems under test, so that each possible combination of architecture (Intel Xeon and AMD Opteron) and operating system (Linux and FreeBSD) can be tested in parallel.

Buffer sizes which affect the capturing must be identified. All buffers can be controlled via the operating system without reboot. For FreeBSD this is done via the `sysctl` command and under Linux via the `/proc` filesystem.

Number of applications per machine which are capturing at the same time induces different load. This affects the distribution process of the “capturing stacks” of the operating systems. It is measured separately because it is unsound to change multiple parameters at the same time.

Filters can be used while capturing. The cost of adding a filter is to be measured.

Per Packet processing is added to simulate an analysis of the captured data.

Writing to disk is limited by the throughput of the disks and has a strong relevance for real operations because when acquiring traces this is often the bottleneck.

Modifications were written to improve the performance of the “capturing stacks”. The benefit of these patches is to be measured.

6.2 Measurement Procedure

Now that all parts for the measurement are at hand, the details of how the measurements are conducted can be explained. As capturing application `createDist` (see Section A.1) is chosen because it is at hand and already performs packet capturing. The necessary extensions for additional load and writing to disk are easy to implement.

6.2.1 Testing Sequence

The sequence introduced in Section 3.4 is repeated several (for most of the measurements seven) times to reduce the impact of outliers or external interference. These repetitions are called reruns in the remainder. Furthermore, a “measurement” includes many of these repeated sequences with continuously increasing data rates. Thereby, it is possible to identify the maximum data rate the system can deal with.

The different data rates are produced by using different inter-packet gaps when generating the packet stream. The Linux Kernel Packet Generator includes this feature therefore it was not necessary to add it. Figure 6.1 shows the data and the packet rate (vertical axis) resulting from the different inter-packet gaps (horizontal axis) used. 26 different inter-packet gaps were chosen, ranging from 100 000 to 0 nanoseconds. The largest inter-packet gap (100 000 ns) produces a data rate of 50 Mbit/s and a packet rate of 10 000 packets per second (pps). Decreasing the inter-packet gap increases the data rate and the packet rate as well. This holds until the inter-packet gap reaches about 3 000 ns, resulting in maximal data rates of 920 Mbit/s and maximal packet rates of 180 000 pps. Although there is no change in the data rate or the packet rate, inter-packet gaps smaller than 3 000 are examined for the sake of completeness.

For the following measurements, every invocation of the Linux Kernel Packet Generator produces 1 million packets which are sent to the systems under test.

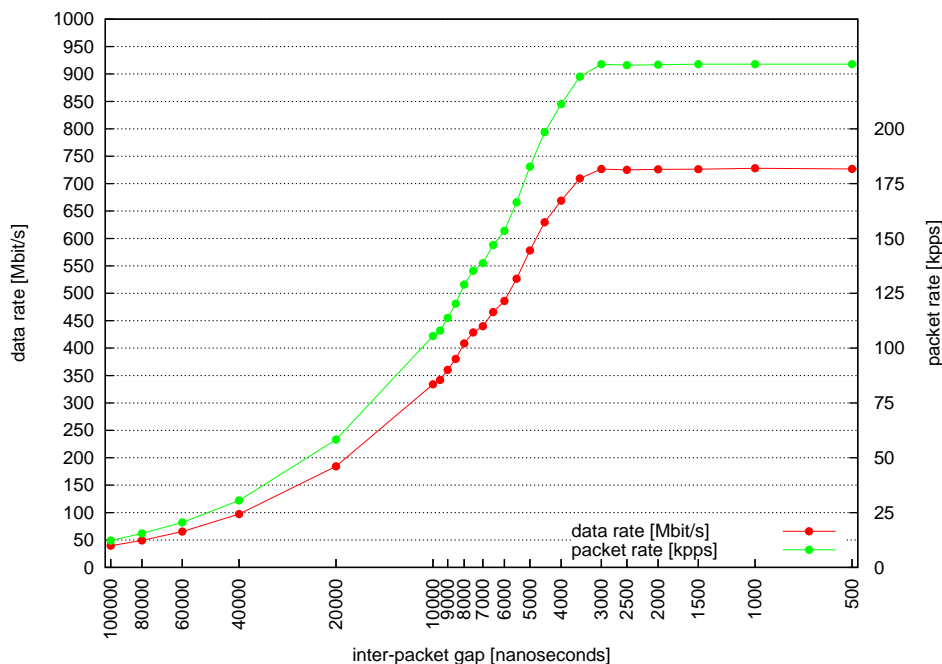


Figure 6.1: Line plot of the data and packet rate against the inter-packet gaps: When the inter-packet gap decreases both the data rate as well as the packet rate increase.

6.2.2 Calculation of the Results

All the evaluation is done separately for each of the four sniffers. The average of the results from several reruns of measurements with identical inter-packet gaps is used. Exemplary evaluations have shown that using the median instead of the average produces only negligible differences. The average was chosen because of its higher explanatory power.

Capturing Rate

For the measurements the capturing rate is defined as follows:

$$\text{capturing rate} = \frac{\text{packets received by the capturing application}}{\text{total generated packets}} \quad (6.1)$$

No distinctions are drawn between packets that were lost before the kernel filtering and which correspond to the receive counter of libpcap and packets that are lost between the filter and the user-space application. This is due to the fact that nearly all packets “reach” all sniffers—confirmed via the SNMP counters of the switch—which lose them on the way to the capturing application.

When a measurement pass is done, a large amount of data is gathered. For every single generation a capturing rate $r(rr, ipg)$ is obtained depending on the index of the rerun $rr \in \{1, \dots, n\}$, where n is the number of reruns, and the inter-packet gap ipg . When plotting, the average rate $r_{\text{avg}}(ipg)$ (6.2) with error bars for minimum ($r_{\text{min}}(ipg)$, (6.3)) and maximum ($r_{\text{max}}(ipg)$, (6.4)) values from identical inter-packet gap values is used:

$$r_{\text{avg}}(ipg) = \frac{\sum_{rr=1}^n r(rr, ipg)}{n} \quad (6.2)$$

$$r_{\text{min}}(ipg) = \min_{rr \in \{1, \dots, n\}} \{r(rr, ipg)\} \quad (6.3)$$

$$r_{\text{max}}(ipg) = \max_{rr \in \{1, \dots, n\}} \{r(rr, ipg)\} \quad (6.4)$$

CPU Usage

For the measurements, the average idle value is taken and subtracted from the maximum of 100 % to receive the overall CPU usage. Similar to the capturing rate, error bars indicate the minimum and maximum CPU usage values per identical inter-packet gap.

Data Rate

Unfortunately, the generated data rate is not constant for similar inter-packet gaps. Especially in the region around 5 microseconds the variation is very high. Therefore, error bars are added in the direction of the data rate (the x-axis in the plots).

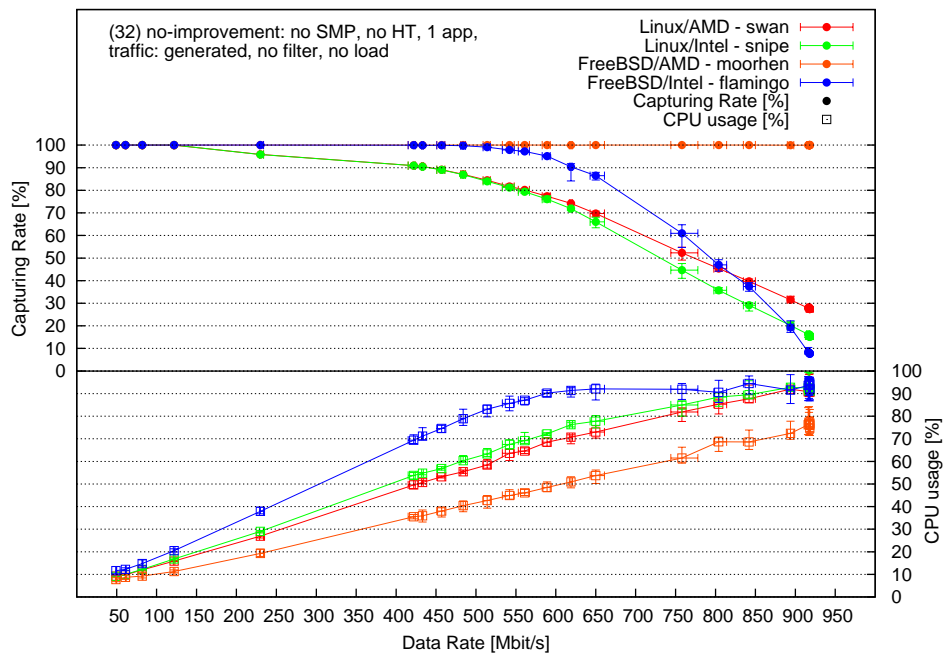
6.3 Results

To be able to compare the impacts of the influencing factors described earlier, it is necessary to know about the performance of the systems in base configuration when capturing packets. Figure 6.2 shows the performance of off-the-shelf systems with operating systems without any improvements. The number in parentheses denotes the id of the performed measurement. These result and most of the following are shown as line plots of the capturing rate and the CPU usage against the data rate.

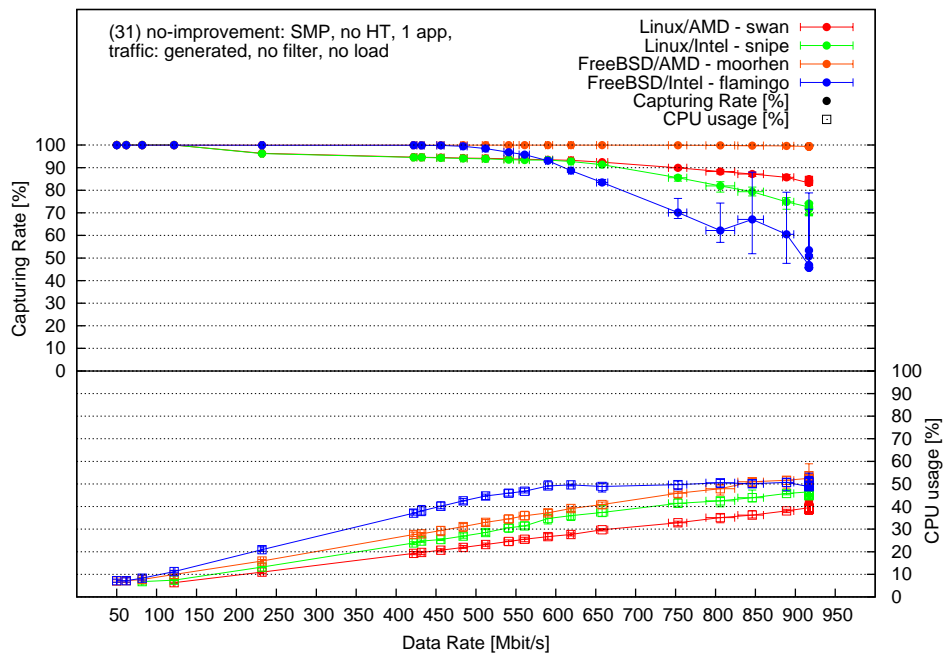
The terms “SMP”¹, “multi processing”, and “dual processor mode” will be used conterminously. The same applies for “no SMP” and “single processor mode”.

The plots show clearly that, independent from the used hardware or software, the capturing rate decreases when the data rate increases. Simultaneously with the data

¹SMP is the abbreviation for symmetric multi-processing, describing the feature of modern operating systems to benefit from more than one processor.



(a) single processor mode



(b) dual processor mode

Figure 6.2: Linespoints plots of the performance of the sniffers without any improvements. The capturing rate and the CPU usage are plotted against the increasing data rate.

rate the packet rate increases. Considering the CPU usage, it increases when the data rate increases because of the higher load caused by the higher rates. These findings were rather unsurprisingly.

The parameters of the capturing application `createDist` (see Section A.1) were set to capture whole packets from the input interface (see Figure 3.1). The only thing the application does is to increment the counter (there is one for every packet size) for the size of the packet. Thus, it does almost nothing in terms of packet analysis or subsequent processing.

When utilizing the second processor, a significant performance increase can be observed. Comparing the single processor measurement (Figure 6.2(a)) with the dual processor measurement (Figure 6.2(b)), one can see that the capturing rate is more than two times higher in multi processor mode. Looking at the CPU usage this result is unexpected. In single processor mode the available computing cycles are nearly completely used. Considering the SMP measurement, the same demands seem to exist: Only one processor is utilized. This is indicated by 50% CPU usage as two fully utilized processors consume 100% of the CPUs in a dual processor system.

Regarding how the CPU usage is related to the capturing rate, it is conspicuous that the Linux systems begin dropping packets before they even fully utilize their CPU(s). That is true for both, the single processor mode and the dual processor mode. Searching for a reason for this matter, buffer sizes have been spotted. Predicting that the performance might increase especially in regions where the CPU is not fully loaded the influence of buffer sizes on the capturing performance is looked at next.

6.3.1 Impact of Buffers

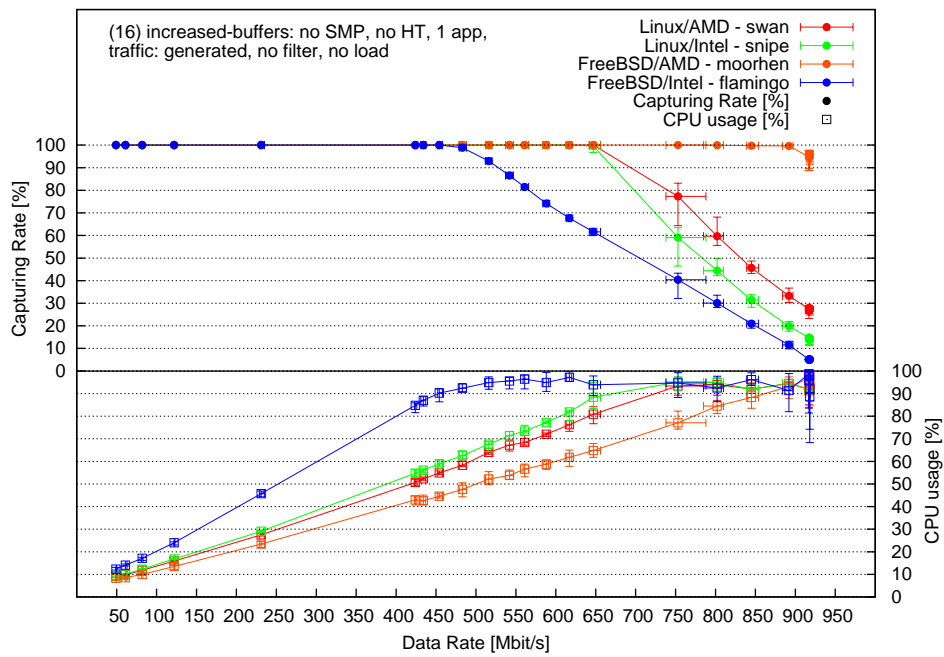
The buffers which can influence the process of packet capturing have to be located within the path the packets take through the kernel as described in Section 2.1. One buffer for each of the operating systems has been identified.

For FreeBSD this is the size of the double buffer, which can be controlled via a `sysctl` that sets the size of both, the HOLD and the STORE buffer (see Figure 2.1). As preliminary investigations have shown, a useful size for these buffers is approximately 10 Mbyte. Larger buffers do not help much. But it was detected that if the size of the virtual kernel memory is increased (to 2 Gbytes) by adding the following line to `/boot/loader.conf`, even bigger buffers can be used:

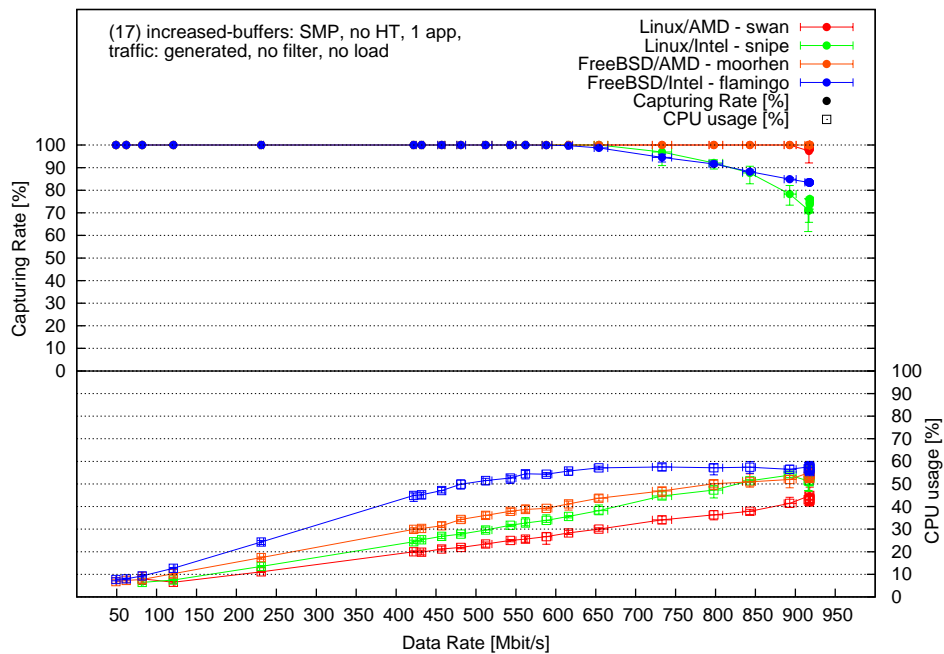
```
vm.kvm_size="2139086848"
```

The following command-lines set the buffer size:

```
sysctl debug.bpf_bufsize=<size in bytes>  
sysctl debug.bpf_maxbufsize=<size in bytes>
```



(a) single processor mode



(b) dual processor mode

Figure 6.3: Linespoints plots of the performance of the sniffers with increased buffer sizes. The capturing rate and the CPU usage are plotted against the increasing data rate.

Since Linux does not implement such a double buffer as used by FreeBSD, a different buffer has to be enlarged in order to improve Linux. It was found that the available memory within the kernel which is used to store the received packets until they are delivered to the appropriate application can be set. This is done via the `/proc` filesystem:

```
echo <size in bytes> > /proc/sys/net/core/rmem_default
echo <size in bytes> > /proc/sys/net/core/rmem_max
```

Measurement with Increased Buffers

Figure 6.3 shows the result for using 10 Mbyte double buffers for FreeBSD and a 128 Mbyte buffer for Linux. One effect of the increase is that Linux does not drop packets any more until it fully utilizes one processor. In the area beginning at 110 Mbit/s the capturing rate, which is at about 90% at the Linux machines (from Figure 6.2), can be elevated to a full capturing rate. Unfortunately, in single processor mode there is no noticeable increase at high data rates (800 Mbit/s or higher).

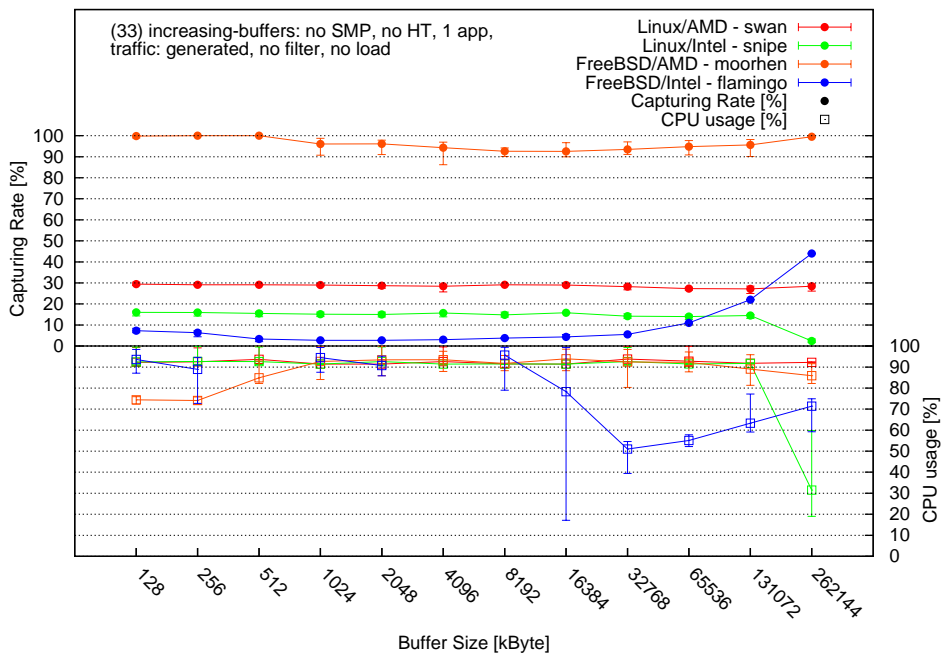
The result for FreeBSD is more complicated. Whereas in single processor mode (Figure 6.3(a)) the increased buffers lead to a slight deterioration, in double processor mode (Figure 6.3(b)) the increased buffers are helpful. This is especially observable for flamingo. A reason for that deterioration could be the delay caused by the copy operation to the user space. This operation copies one complete buffer, as large as it is, continuously into userspace.

Recapitulatory, Linux benefits from the increased buffers even at low data rates, whereas FreeBSD only ameliorates with SMP and at higher data rates. Motivated by and a bit curious about the findings of Figure 6.3, a further investigation of the impact of buffer sizes has been prepared.

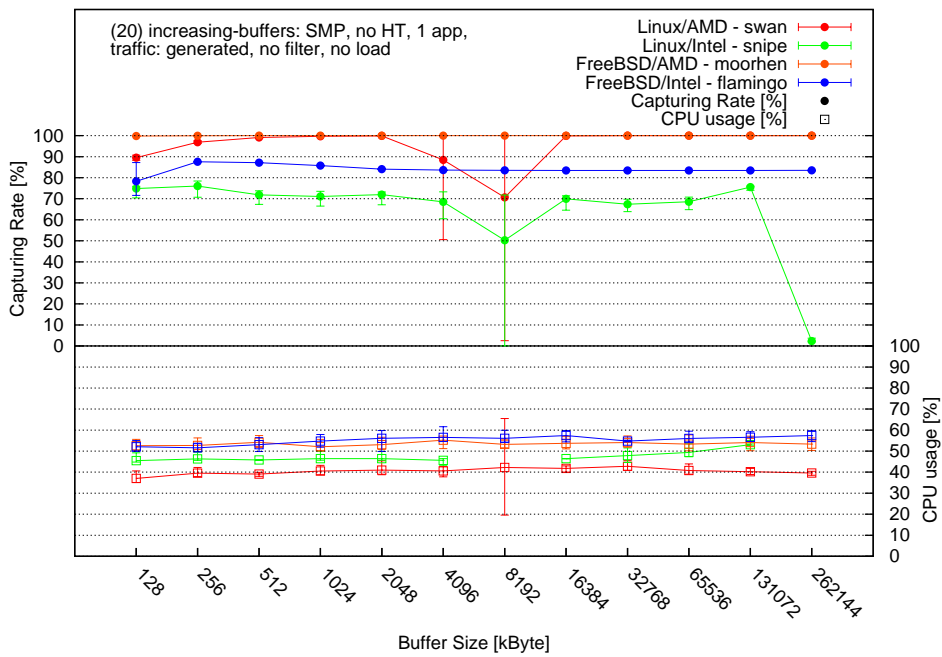
Changing the Buffer Size

Since the impact of these buffers is fairly significant, a measurement with increasing buffer sizes instead of increasing data rates was carried out. The highest possible data rate (no inter-packet gap) was used, because the differences are most significant at this rate. To conduct a fair measurement, the buffer size was reduced by a factor of two for FreeBSD. Since FreeBSD utilizes a double buffer, the effective buffer size is equal with that from single buffered Linux.

Figure 6.4 shows the results. With regard to the dual processor measurement (Figure 6.4(b)), there is no noticeable improvement beyond a size of 512 kByte. But then again the single processor plot (Figure 6.4(a)) shows that both FreeBSD systems perform inferior with buffer sizes between 512 kByte and 131 072 kByte. This is quite surprising but could be explained with the increased expense of copying the



(a) single processor mode



(b) dual processor mode

Figure 6.4: Linespoints plots of the performance of the sniffers with the impact of buffer size. The capturing rate and the CPU usage are plotted against the increasing buffer size.

complete buffer from the kernel into userspace as explained in the last section for `flamingo`.

The performance increase of `flamingo` with large buffers (again in the single processor measurement Figure 6.4(a)) can be explained with the size of the buffer. Within a buffer of 256 MByte about 415 958 packets can be stored given an average packet size of about 645 Bytes. (This average packet size is directly calculated from the used packet size distribution). Since 1 million packets are generated per run, the fraction of packets which can be stored in the biggest buffer tested fits perfectly with the plotted capturing rate. However, this result shows that `flamingo` is not able to handle such high data rates at all.

The break-in of the performance of the Linux systems at 8192 kBytes in the SMP plot and of `snipe` with the largest buffers seems to be some kind of artefact or side-effect.

For the following measurements the increased buffers (10 Mbyte double buffers for FreeBSD and 128 Mbyte buffer for Linux) are used, being aware that this is not optimal for single processor FreeBSD systems. Nevertheless, this avoids changing buffer sizes all the time and keeps the plots comparable.

6.3.2 Using Packet Filters

As explained in Section 2.1, it is possible to set up a packet filter. Only the packets that match the filter expression are captured and made accessible to the application. In FreeBSD this is realized by simply not copying the packet into the STORE buffer, whereas in Linux no pointer is inserted to the queue for the capturing “socket”.

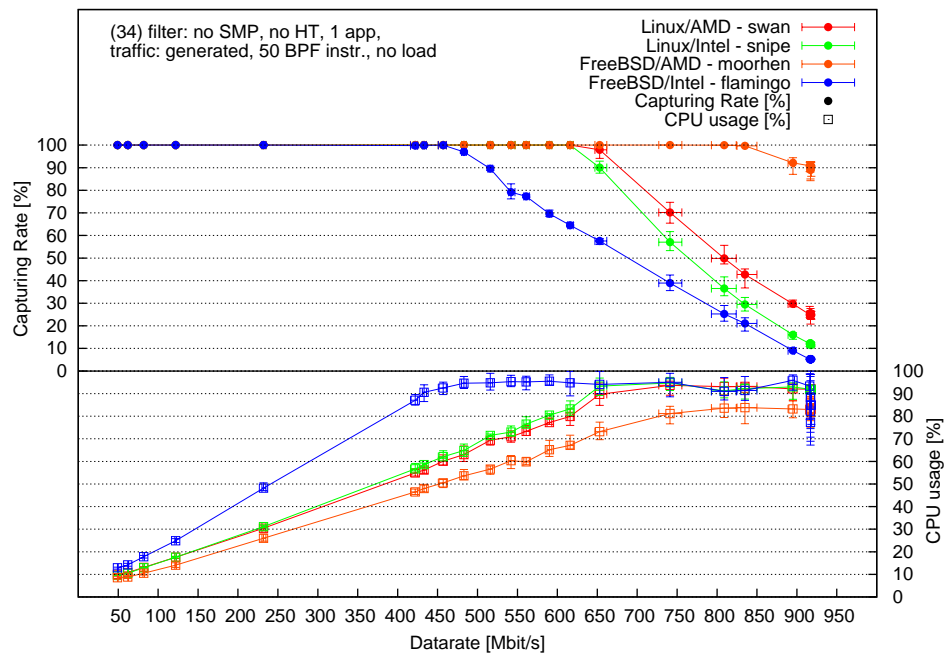
BPF filter expressions can match any packet or no packets at all. In either case it is necessary to evaluate the filter expression. Thus, if the packet is accepted, this filtering generates additional load. Therefore, it is interesting find out about the cost of this filtering in terms of CPU usage and capturing rate.

For the measurement shown in Figure 6.6 the possibility to use a filter was added to the capturing application, utilizing the `pcap_compile()` and `pcap_setfilter()` functions from `libpcap`. Since the filter language defined for the BPF is assembler like, it is impracticable to write filter programs from scratch. For this reason the `libpcap` defines a higher level language also used by `tcpdump` [JLM94] for defining filters. The filter expression listed in Figure 6.5 was used for the measurement.

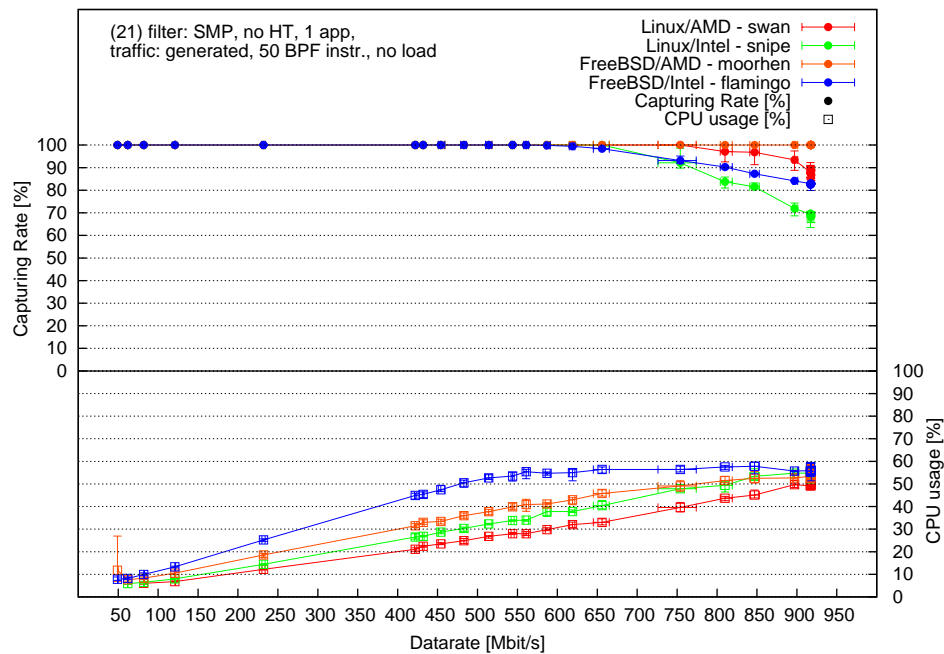
Compiled into the BPF language (as introduced by [MJ93]) this filter is 50 instructions long. The filter was chosen so that all generated packets are accepted but not until all of the instructions are evaluated. The destination IP of the generated packets is set to 192.168.10.12, the source IP is 192.168.10.100, and the source Ethernet address is set to cycle between 00:00:00:00:00:00 and 00:00:00:00:00:02, hence this expression really matches all generated packets.

```
1 ether[6:4]=0x00000000 and ether[10]=0x00 and not tcp
  and not ip src 10.11.12.13
  and not ip src 20.11.12.14
  and not ip src 30.11.12.15
5  and not ip src 40.11.12.16
  and not ip src 50.11.12.17
  and not ip src 60.11.12.18
  and not ip src 70.11.12.19
  and not ip src 80.11.12.20
10 and not ip src 90.11.12.21
   and not ip src 100.11.12.22
   and not ip src 110.11.12.23
   and not ip src 120.11.12.24
   and not ip src 130.11.12.25
15 and not ip src 140.11.12.26
   and not ip src 150.11.12.27
   and not ip src 160.11.12.28
   and not ip src 170.11.12.29
   and not ip src 180.11.12.30
20 and not ip src 190.11.12.31
   and not ip dst 10.99.12.13
   and not ip dst 20.99.12.14
   and not ip dst 30.99.12.15
   and not ip dst 40.99.12.16
25 and not ip dst 50.99.12.17
   and not ip dst 60.99.12.18
   and not ip dst 70.99.12.19
   and not ip dst 80.99.12.20
   and not ip dst 90.99.12.21
30 and not ip dst 100.99.12.22
   and not ip dst 990.99.12.23
   and not ip dst 120.99.12.24
   and not ip dst 130.99.12.25
   and not ip dst 140.99.12.26
35 and not ip dst 150.99.12.27
   and not ip dst 160.99.12.28
   and not ip dst 170.99.12.29
   and not ip dst 180.99.12.30
   and not ip dst 190.99.12.31
```

Figure 6.5: This listing shows the BPF expression which is used for the measurements evaluating the impact of installing a packet filter for the capturing process.



(a) single processor mode



(b) dual processor mode

Figure 6.6: Linespoints plots of the performance of the sniffers with additional performed filtering. The capturing rate and the CPU usage are plotted against the increasing data rate.

When comparing Figure 6.3(a) and Figure 6.6(a) only slight differences can be perceived. In the case of the dual processor measurements (Figure 6.3(b) vs. Figure 6.6(b)), Linux drops a few more packets (up to 10%) at the highest data rates when performing additional filtering. On that account using BPF filters is cheap with respect to the possible benefit of filtering out unwanted packets, which is reducing the load due to saved copy operations and cuts down data analysis costs.

6.3.3 Running Multiple Capturing Applications

Hitherto existing measurements have shown that utilizing the second processor is beneficial. For a further investigation of the capabilities of a dual processor machine (with SMP activated) multiple instances of the capturing application are started.

Since it can happen that different applications capture different fractions of the generated packets, the plots have to be enhanced to display these differences. Therefore, the capturing rate of every machine is now represented by three lines, which may superpose each other (especially observable for the FreeBSD systems): (i) one for the application with the worst capturing rate, (ii) one for the average capturing rate of all applications running at a time, and (iii) one for the application with the best capturing rate. These three different lines are indicated by different symbols for the measurement points.

Recalling the results of the project thesis (see [Sch04]) a large variability of the performance of separate applications running at the same time under Linux is expected to be seen. But as Figures 6.7, 6.8, and 6.9 show this effect cannot be recognized as strong as expected. The divergence of the different applications is still larger than under FreeBSD.

While all systems can run two capturing applications at the same time with minor performance reduction compared to Figure 6.3(b), adding further capturing applications worsens the situation: Linux captures nearly no packet when a certain data rate threshold is reached. Although the capturing rate of FreeBSD does not deteriorate as steep as the one of Linux, the additional applications have an impact.

Regarding Figures 6.8 and 6.9, it is significant that Linux begins massively losing packets when the load exceeds the CPU capabilities. This behavior may result from the fact that Linux uses reference counting for the packets in kernel memory. If any application does not release the claim for a packet this packet is kept forever, blocking kernel memory. Once the kernel memory buffer is full, every further incoming packet will be dropped.

When interpreting the results shown in Figures 6.7–6.9, it is obvious that one should avoid using multiple capturing applications simultaneously.

Figure 6.7: Linespoints plots of the performance of the sniffers with *two* capturing applications. The capturing rate and the CPU usage are plotted against the increasing data rate.

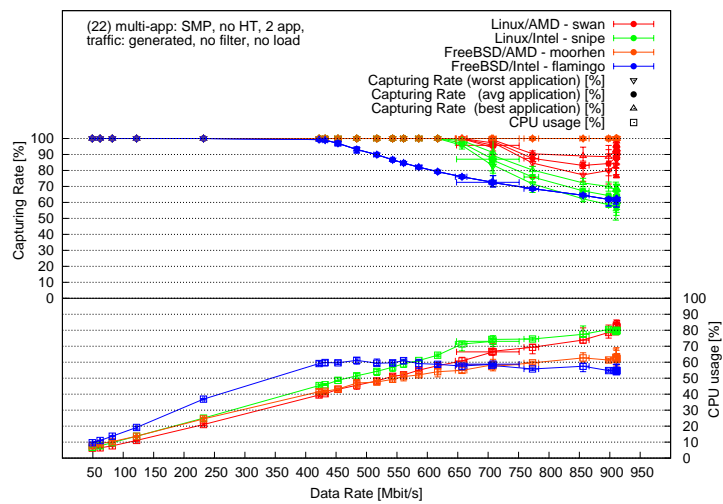


Figure 6.8: Linespoints plots of the performance of the sniffers with *four* capturing applications. The capturing rate and the CPU usage are plotted against the increasing data rate.

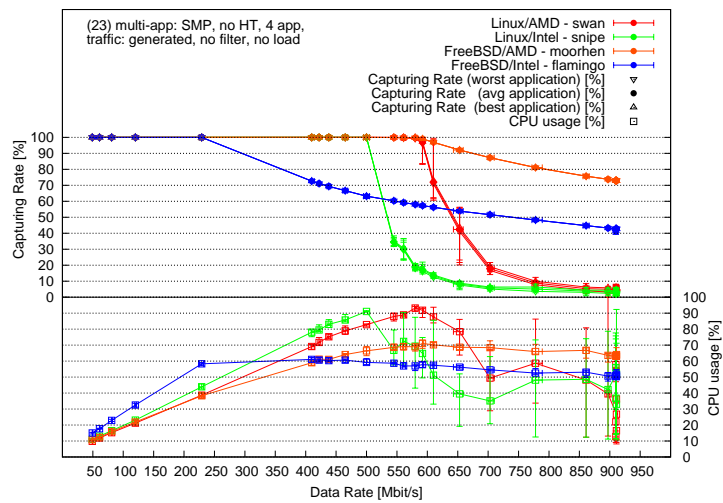
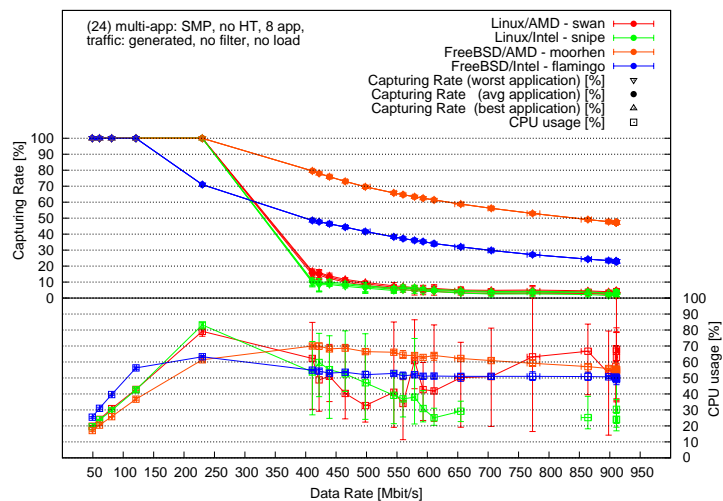


Figure 6.9: Linespoints plots of the performance of the sniffers with *eight* capturing applications. The capturing rate and the CPU usage are plotted against the increasing data rate.



6.3.4 Adding Packet Based Load

The goal of the following measurements is to simulate traffic analysis tools. To achieve this goal, two different approaches are used. The first tries to add supplementary load by performing packet copy operations after the packet was successfully captured. The second idea is to perform a deflating operation from the `zlib` [GAR05], a popular C library for data compression.

Additional Copy Operations

The additional copy operations are implemented via the `memcpy` (see manpage `memcpy(3)` for details) function which is called multiple times. The number of copy operations is fully user configurable, thus allowing to easily explore different settings.

The measurement corresponding with Figure 6.10 was setup with 50 sequentially performed copy operations. Another measurement with 25 copy operation has been carried out. The results are shown in Figure B.2 in the appendix. Returning to Figure 6.10, once again `moorhen` performs best. In the single processor plot (Figure 6.10(a)) even at moderate data rates the CPU usage reaches 100 % which results in packet drops when the data rate is further increased.

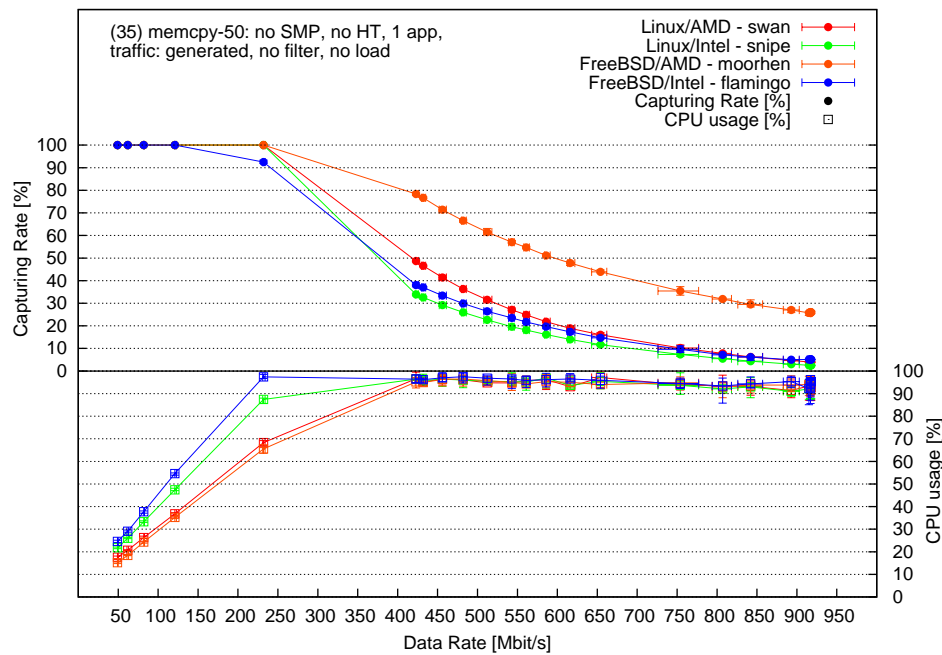
Whereas in single processor mode both Opteron machines perform best, in dual processor mode both FreeBSD systems are a notch above the Linux systems. This may result from a higher CPU utilization, where Linux has nearly a free second processor.

Additional data compression

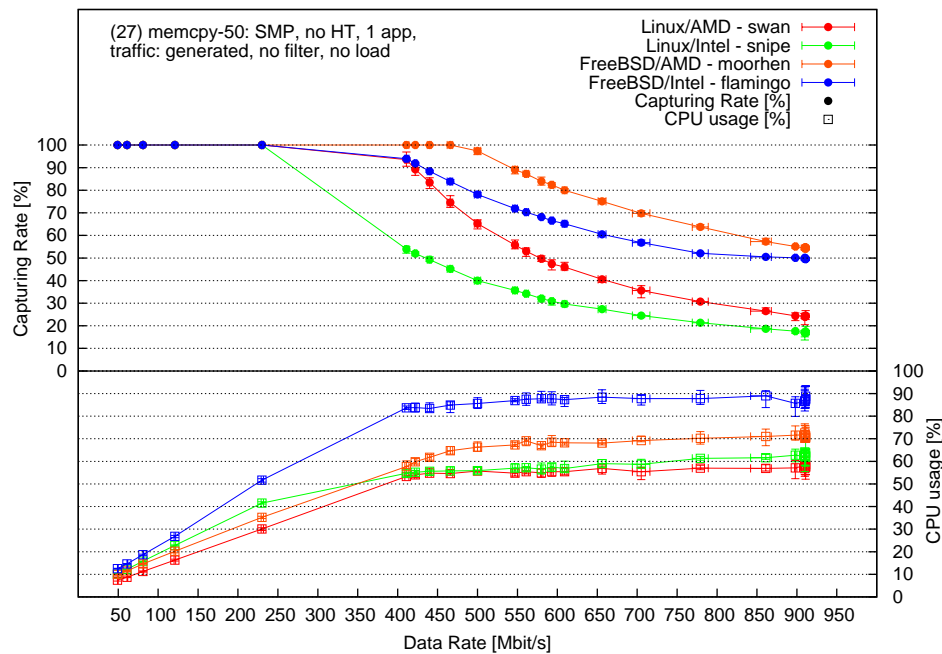
Since copying packets is primarily memory throughput limited and tests the caching mechanisms, it is furthermore of interest to see what happens if the processor is utilized by packet processing. Therefore, the capturing application was enhanced to perform compression of the packet data. This was implemented using the `zlib`.

When the program is started, `gzopen()` is called with the desired compression level and `/dev/null` as filename. So, actually no output has to be written to the filesystem. This prepares the compression and provides a file handle for further processing. Within the processing of each packet the function `gzwrite()` is called to compress the packet data to the preset file handle. When the program is done, `gzclose()` is called to cleanly finish the compression and close the file.

Different CPU loads can be achieved by setting different compression levels. These reach from 0 (no compression, fast) to 9 (maximum compression, slow). The first measurement was done with zip level 9. But with this high compression rate all systems suffer under overload, forcing them to drop nearly all packets. The results are shown in Figure B.3 in the appendix.

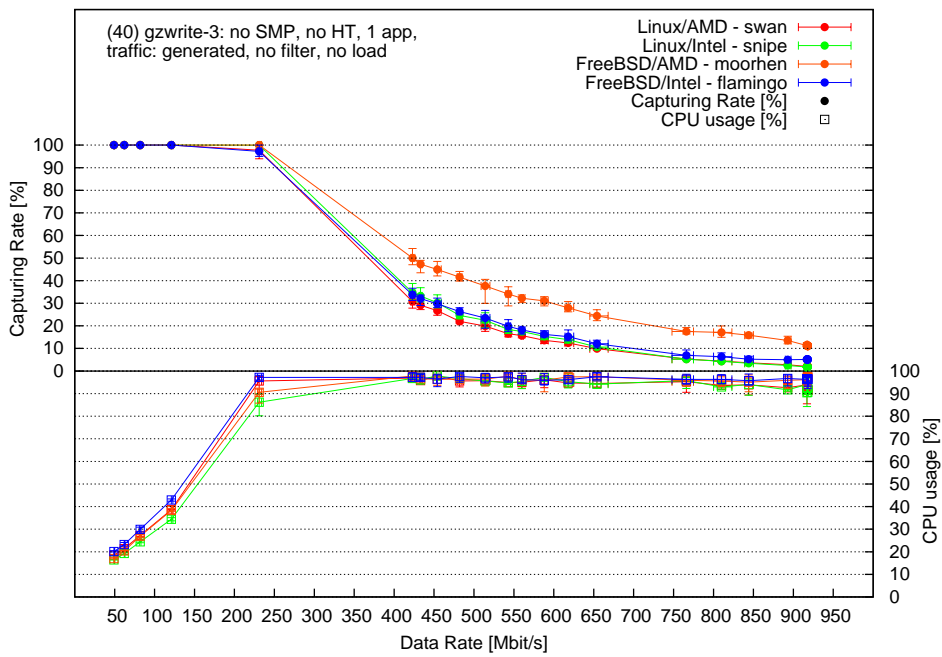


(a) single processor mode

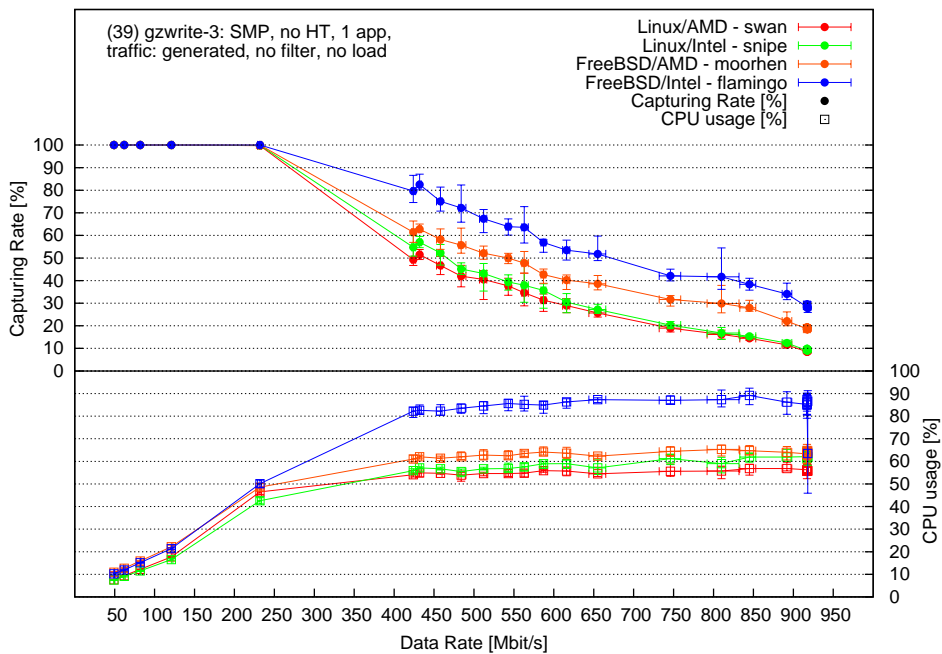


(b) dual processor mode

Figure 6.10: Linespoints plots of the performance of the sniffers with 50 additional packet copies. The capturing rate and the CPU usage are plotted against the increasing data rate.



(a) single processor mode



(b) dual processor mode

Figure 6.11: Linespoints plots of the performance of the sniffers with additional zlib compression (level 3).

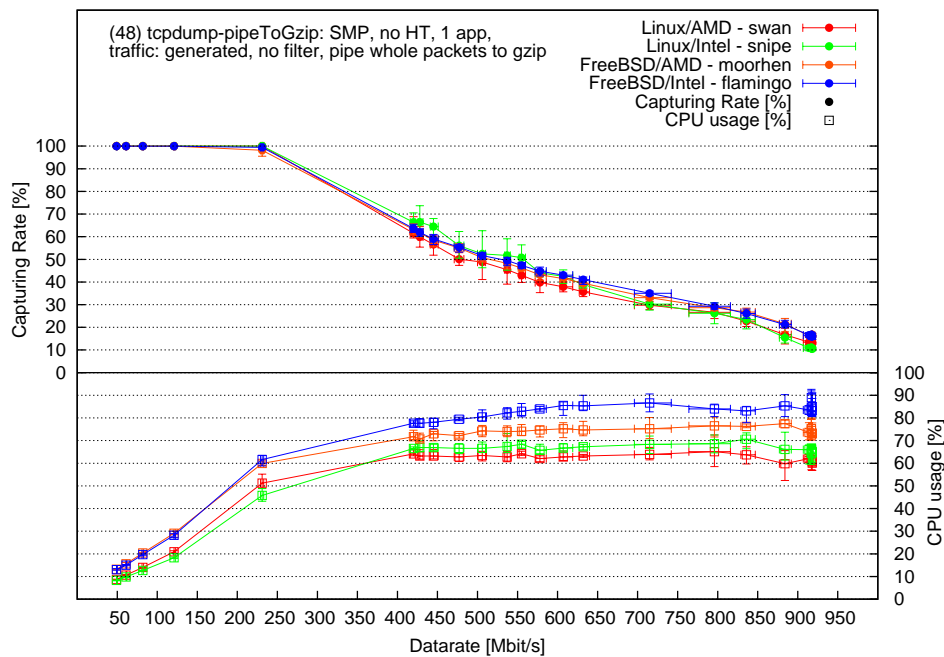


Figure 6.12: Linespoints plots of the performance of the sniffers performing additional compression via a pipe to gzip. Dual processor mode. The capturing rate and the CPU usage are plotted against the increasing data rate.

A second measurement with compression level 3 was more enlightening (refer to Figure 6.11). Due to the fact that the overall load has not been as high as in the first measurement, the differences emerge more clearly. Comparable to the results from Figure 6.10(b) the FreeBSD systems perform better than the Linux systems as shown in Figure 6.11(b). But this time each of the Intel systems performs better than the corresponding AMD system. This is a novelty among the measurements. The Intel processors seem to be much more efficient for the special task of compression as implemented in the `zlib`. But nevertheless, the effects of additional load are definitely noticeable.

Piping to gzip

Analyzing Figure 6.11(b) it is eye-catching that the processors are not fully utilized. Thus, another measurement was set up to see if the performance can be increased by separating the process of capturing from the process performing the compression. Therefore, the mechanism of pipes was used (see `man mkfifo` for details):

```
mkfifo sniffer_pipe
gzip -c -v -3 2>> <logfile> > <tracefile> < sniffer_pipe &
/usr/local/sbin/tcpdump -s 1515 -w sniffer_pipe -i if0 -n && <dumpfile> &
```

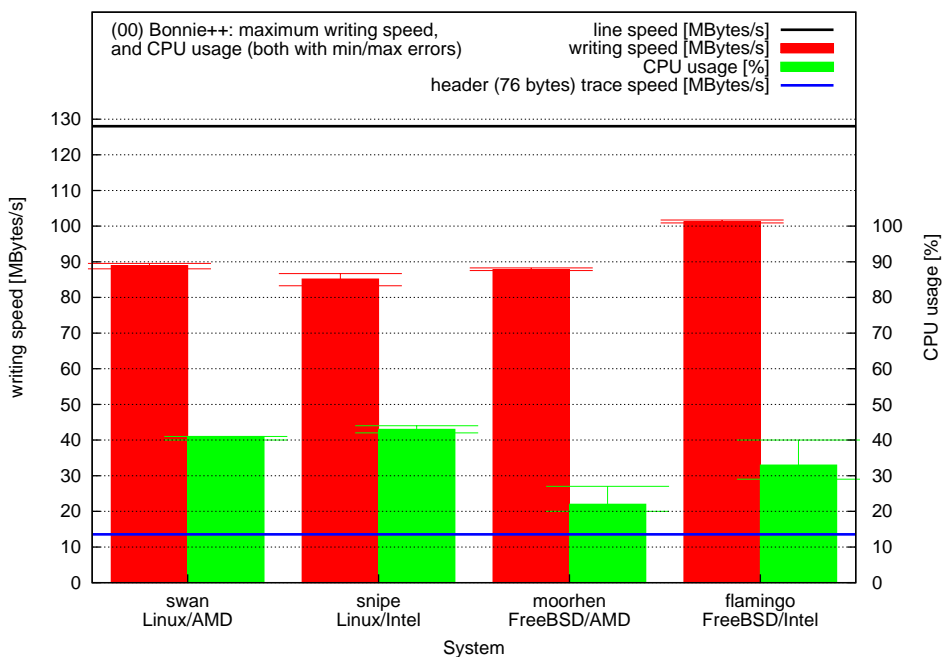


Figure 6.13: Histogram of the maximum writing speed and the CPU usage of the machines when writing data to disk.

As shown in Figure 6.12, all systems are very close to each other in terms of capturing performance. *moorhen* shows the same performance as in Figure 6.11(b), and the other systems improve or worsen in direction to *moorhen*. For all systems the CPU utilization is higher. Thus, the piping mechanism seems to present a new bottleneck for the capturing process.

6.3.5 Writing to Disk

Before trying to write the captured data to disk, the maximum writing speed of the different systems is determined using the *bonnie++* [Cok] toolkit. Figure 6.13 depicts the speed at which the systems are able to write onto their RAID sets and the CPU load caused by this writing. The black line indicates the throughput which would be required to write packets arriving at line speed to the disk. The results show that none of the systems is able to write data to disk at line speed.

Therefore, we decided to capture full packets but write only the headers to disk. To keep it simple, the first 76 Bytes of every packet shall be written to disk. This produces a theoretical throughput demand of 13.56 Mbytes/s as indicated by the blue line in Figure 6.13.

Writing Headers to Disk

As shown in Figure 6.13, writing data to disk produces additional load in terms of CPU utilization. Supplementary to this increase of the load, writing to disk causes I/O which produces additional interrupts and consumes bandwidth of the PCI-64bit bus which is shared with the network card. Therefore, a decrease in the capturing performance is expected when the headers are written to disk.

Figure 6.14 shows the outcomes of the measurement. Surprisingly, there is only a slight deterioration noticeable at all systems. Hence, the PCI-64bit bus bandwidth is not used completely by the network card leaving enough resources for the disk operation.

It is worth mentioning that the AMD Opteron systems in single processor mode (see Figure 6.14(a)) suffer more under the additional load of writing to disk at the highest rates than the Intel Xeon systems, both with a decrease of about 10%. But both perform still better than the Intel systems.

This is different for the dual processor mode (Figure 6.14(b)) where the FreeBSD systems are performing with no noticeable difference than without writing to disk. But the Linux systems are losing about 10% of their capturing rate. Thus, once again, the FreeBSD/AMD Opteron combination performs best.

In terms of CPU utilization there is no perceivable change in the performance of the four systems under test.

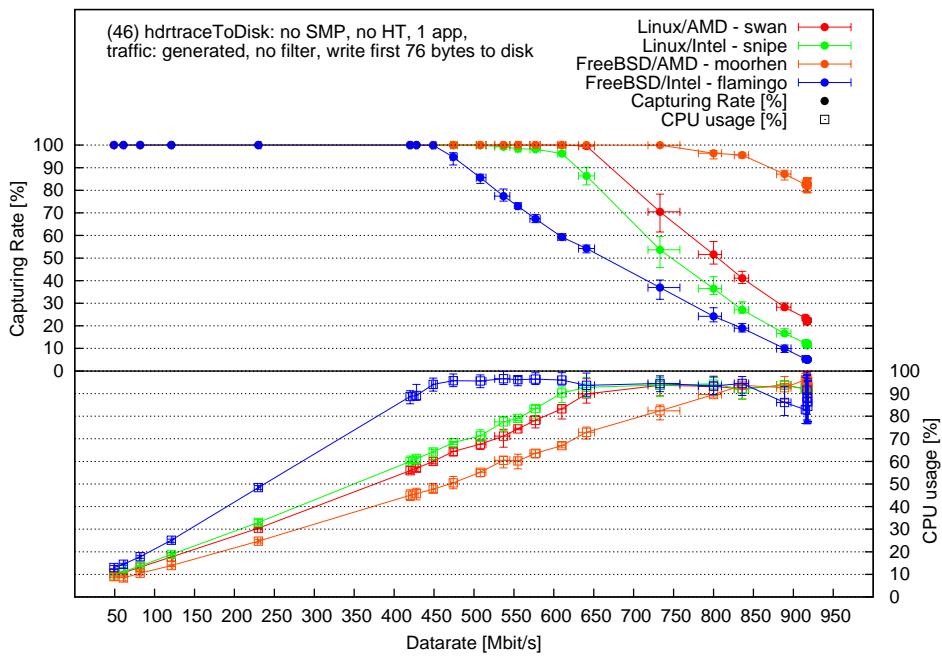
6.3.6 Patches to the Linux Capturing Stack

Now, that the impacts affecting all architecture/operating system pairs have been discussed other factors which will only affect a single operating system are to be researched. This section describes proposed improvements of the capturing capabilities of Linux.

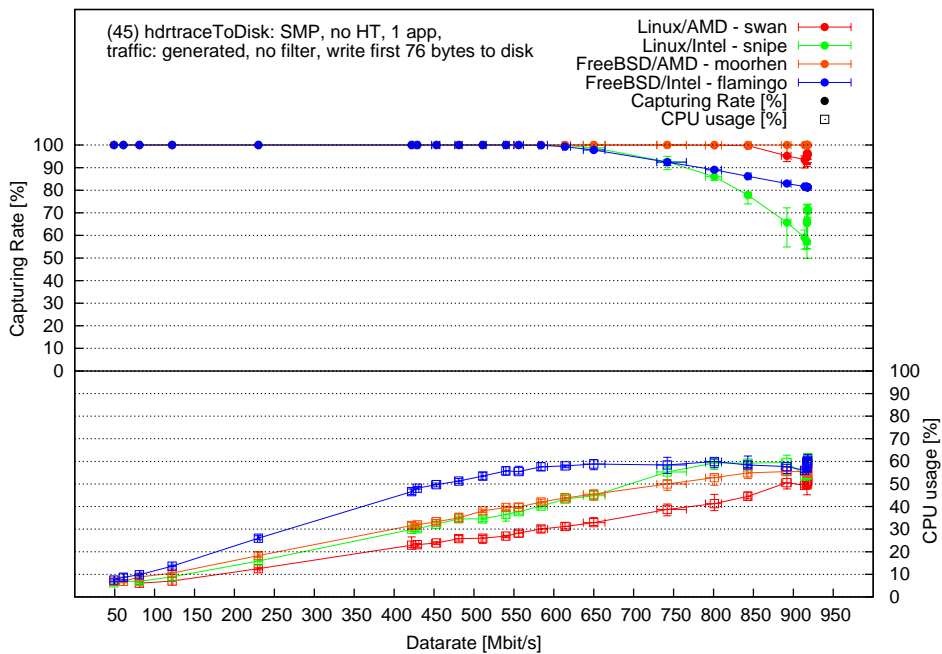
Memory-mapped libpcap

As described in Section 2.2, using memory-mapped buffers helps reducing the copy operations within the capturing stack from three to two. The `mmap` patch for libpcap by Phil Woods [Woo] implements this feature. But since it uses the NAPI [SOK01] features of the Linux kernel, this patch will only work with Linux. It is indeed necessary to activate the `CONFIG_PACKET_MMAP` option when building the kernel which is usually default.

A rigorous performance improvement can be measured if this patch is utilized. Figure 6.15 shows the results. The grey lines repeat the result of the Linux systems from Figure 6.3. Outperforming the unpatched version and the corresponding FreeBSD



(a) single processor mode



(b) dual processor mode

Figure 6.14: Linespoints plots of the performance of the sniffers when writing the header of every packet to disk.

systems the drop rate of the Linux systems is only visible at the highest rate in single processor mode on snipe.

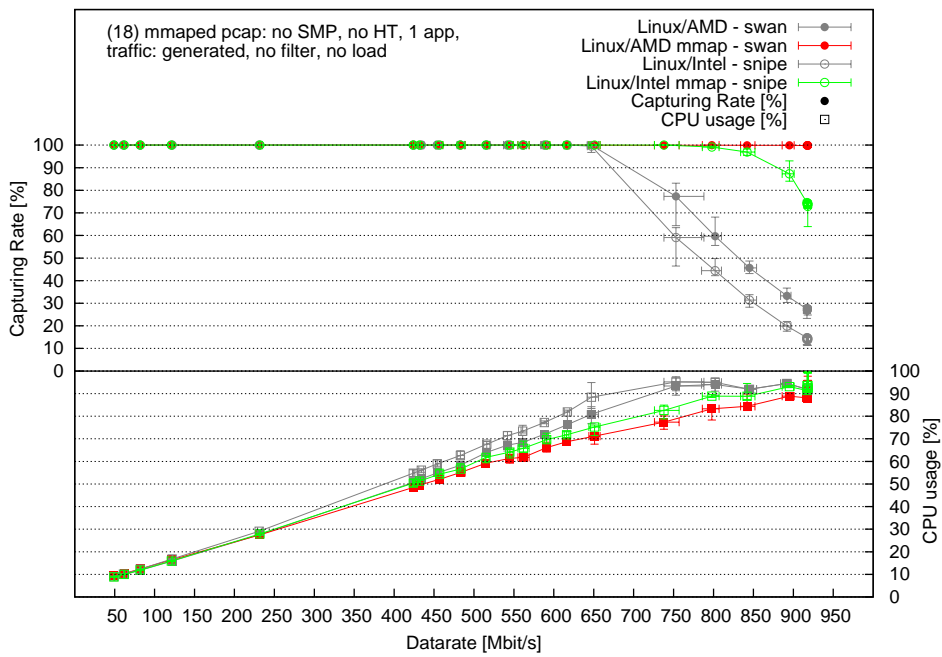
Despite of benefits, this patch brings along problems as well. The non-blocking read mode of libpcap is not supported by now, leading to the disadvantage that applications using this functionality will not work with this patch. Specifying non-blocking mode changes the behavior of the methods used to acquire the next packet from the operating system. Usually, if there are no packets to be fetched, libpcap blocks until further data is ready or a timeout is exceeded. In non-blocking mode, a request for the next packet is answered by returning 0, indicating that no packets are available, or by returning a pointer to the packet.

One of the applications using the non-blocking mode is the Open-Source intrusion detection system Bro [LBN, Pax99], which is subject for research in our group. Since the patch modifies only libpcap, it can be used parallel to the unmodified version. This is achieved through building the two libraries as shared objects and linking the applications dynamically. Then the user can choose the appropriate library by using the `LD_LIBRARY_PATH` environment variable to set the correct path and name of the shared object.

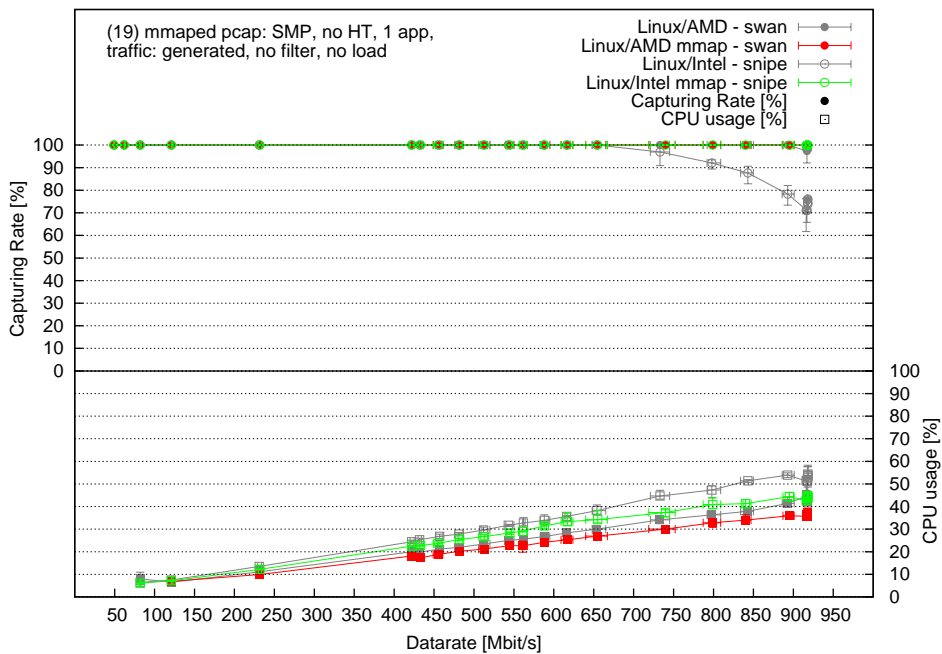
6.3.7 Hyperthreading

At last, the impact of using Hyperthreading, only available for Intel Xeons, was examined. Hyperthreading is a technology Intel introduced to counteract the problems of cache misses and branch misprediction. When data necessary for the next processor operation is not available (cache miss), it has to be loaded. This causes the processor to be idle, wasting computation cycles. Furthermore, modern processors process instructions in pipelines. To do this effectively, the pipelined architecture tries to guess the next instruction when a branch is encountered. Sometimes, this guess is wrong which requires to rebuild the pipeline, again leading to idle cycles. To reduce this idle periods with Hyperthreading, the pipelines and some registers are duplicated (which is much cheaper then duplicating the whole core), and two virtual processors are provided per processor. This makes it possible to switch between virtual processors if one of the virtual processors is idle. See [MBH⁺02] for details.

Therefore, one might expect that turning Hyperthreading on improves the performance of the packet capturing systems. Figure 6.16 compares the performance of the Intel systems with Hyperthreading turned on and off for both operating systems. The grey lines repeat the results of the Intel systems from Figure 6.3(b). There is neither a noticeable amelioration nor deterioration when turning on Hyperthreading. Based on these results it is not possible to conclude an advice for using Hyperthreading or not.



(a) single processor mode



(b) dual processor mode

Figure 6.15: Linespoints plots of the performance of the sniffers with an mmaped libpcap under Linux.

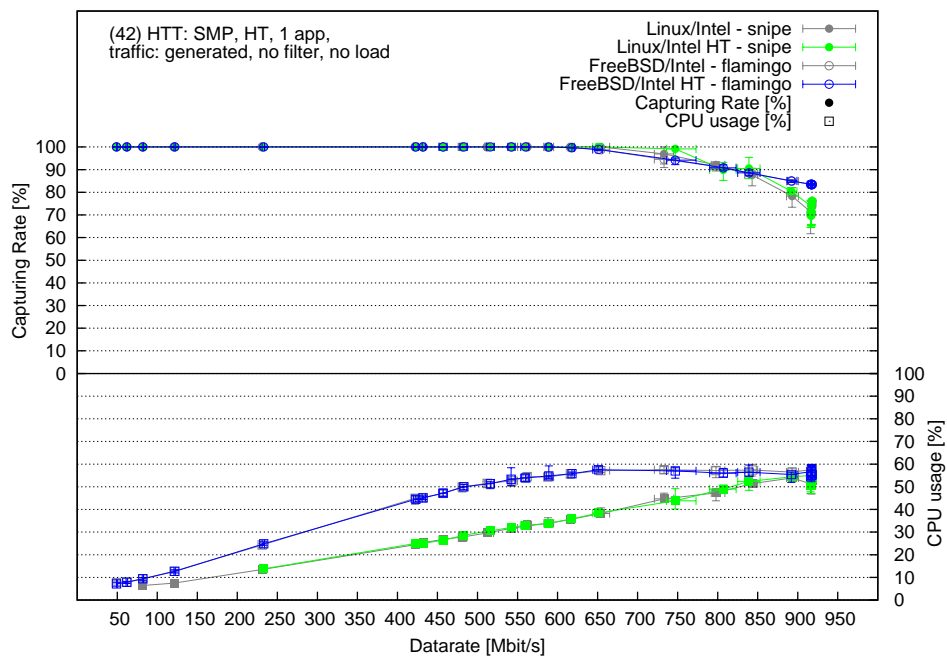


Figure 6.16: Linespoints plots of the performance of the sniffers using Hyperthreading on the Intel Xeon machines. The capturing rate and the CPU usage are plotted against the increasing data rate.

7 Conclusion

This chapter briefly summarizes this work and gives an idea of what might be interesting in the future in the field of capturing with commodity hardware.

7.1 Summary

To answer the question, which operating system/architecture combination performs best in terms of packet capturing performance, it is necessary to find a suitable environment for such a comparison. In this case, we chose to impose generated workload on four comparable up-to-date commodity systems at the same time using an optical splitter. These four systems were equipped with dual Intel Xeon and with dual AMD Opteron processors, two systems each. Identical systems in terms of architecture are then installed with Linux and FreeBSD, thus creating four different combinations.

Using this methodology introduced in Chapter 3 with an appropriate workload source it is possible to evaluate different systems with respect to their abilities to capture high bandwidth network traffic. It has been shown that it is possible to generate the required workload—in this case UDP packets following a given packet size distribution—at line speed in Gigabit Ethernet environments with commodity hardware. Thus, no expensive specialized hardware (or weird experiments as described in [Der04, Der05]) are necessary to test network hard- and software.

Various factors influencing packet capturing performance have been measured and compared to decide which of the examined operating systems/architecture combination performs best. As the first measurements have shown, *moorhen*, the FreeBSD 5.4/AMD Opteron combination, is performing best, losing nearly no packets in single processor mode and no packet at all in dual processor mode. Whereas, *flamingo*, the FreeBSD/Intel Xeon combination, is often losing more packets than the other systems.

Further investigations of the results led to the crucial impact of buffer sizes for the Linux systems. Granting the systems more space to buffer incoming packets, the data rate where packet drops begin to occur can be raised from about 225 Mbit/s to about 650 Mbit/s. But under FreeBSD in single processor mode the capturing rate decreases with higher buffers. Therefore, it is necessary to be careful about arbitrarily increasing buffer sizes. 10 Mbytes for the double buffer of FreeBSD and 128 Mbytes for the Linux packet receive buffer have proven to be a good choice.

When filters are used, the additional load is almost negligible. On that account it is useful to apply a filter if it is needed, because the savings of not processing the filtered packet are reducing the load in any case.

Using multiple applications to capture traffic at the same time is never benefitting, but using two applications on two processors is still acceptable in terms of the capturing rate. If even more applications are capturing at the same time, the internal distribution mechanism of FreeBSD ensures that at least relevant fractions of packets are delivered to the applications, whereas Linux' capturing rate drops nearly to zero when the system is under overload.

Another important impact is the load which is caused by packet analysis. Additional packet copy operations as well as compressing the captured data (where the Intel machines have an advantage) impose high loads on the systems, leading to a large number of packet drops.

The most relevant result for pulling traces of network packets is that writing the headers of the packets to disk is cheap in terms of additional load. There is almost no difference compared to the measurement without this writing. Writing the whole packets to disk could not be measured because the RAID of the capturing systems is not fast enough for line speed.

At least it is worth mentioning that using the newest operating system version (especially the step from FreeBSD 5.2.1 to 5.4; see Figure B.1 in the appendix) and using available patches for the capturing stacks is quite benefitting. On the other hand, utilizing the Hyperthreading feature of the Intel systems is neither ameliorating nor deteriorating the performance of the systems.

Concluding, it is important to choose the right architecture and operating system for a special task. At this time FreeBSD in combination with AMD Opteron hardware is the system of choice. Nevertheless, capturing packets on a Gigabit Ethernet link is a nontrivial task, even with the best commodity hard- and software for this challenge.

7.2 Future Work

Although many question have been answered in this thesis, further questions appeared and new ideas to improve the capturing capabilities were born. This section tries to outline the possible future work in this field of interest.

The most commonly interest would be the evaluation of 10 Gigabit Ethernet with respect to the possibility to capture packets in these environments. The difficulty is the further increased maximum packet and data rate, requiring faster busses and disks. Distributing the analysis of the data might be a chance of conquering the bandwidth. This can either be accomplished by using multiple threads on one

machine to take full advantage of multiprocessor systems—[DV04] describes preliminary implementations of this approach—or by physically distributing the traffic over different machines for analysis. The second approach leads to the problem of synchronizing the different processing units. A means of communication between these units is described in [Som05].

Furthermore, evaluating the possible benefits of 64 bit systems and testing upcoming operating system versions (like FreeBSD 6) might be enlightening. Another patch from Luca Deri [Der05] which realizes a complete new capturing stack for Linux is available and awaits its evaluation.

Another interesting project would be the implementation of a memory-mapped libpcap for FreeBSD as well. Since FreeBSD seems to perform better than Linux in general, this could boost the capturing rates and reduce the CPU load.

A Programs Written for the Thesis

A.1 createDist

This program is used to count packet sizes, calculate distributions from the results of the counting, and produce input for the enhanced Linux Kernel Packet Generator (see Section A.2) for the measurements.

In addition, this application was used as the capturing application for the measurements. For some measurements additional load per packet was needed. Furthermore, the capability of writing the packet data to a file was added.

The sources to this application are available at
<http://www.net.in.tum.de/~schneifa/sources/createDist-0.1.tar.gz>.

A.1.1 Possible Input and Output

The following types of input and output are designated. Figure A.1 shows the interrelationships between the different types. For every use of the application one of the available types has to be chosen as input and one as output.

sizes: This is simply a sequence of numbers representing packet sizes. Same numbers can occur arbitrarily often in the file which can be of any length. **sizes** might be the input from some sort of network data analysis tool like `ipsumdump` or `tcpdump`.

dist: Such a distribution consists of lines like:

```
<pkt_size> <#count>
```

This can be derived from the **sizes** by simply counting the occurrences of the packet sizes.

prodfs: This is exactly the type of distribution which can directly be fed to the enhanced version of the Linux Kernel Packet Generator. It is produced by the calculation in Section 4.2.3 Equation (4.10) and the format is explained in Section A.2.2.

The following additional input types are possible:

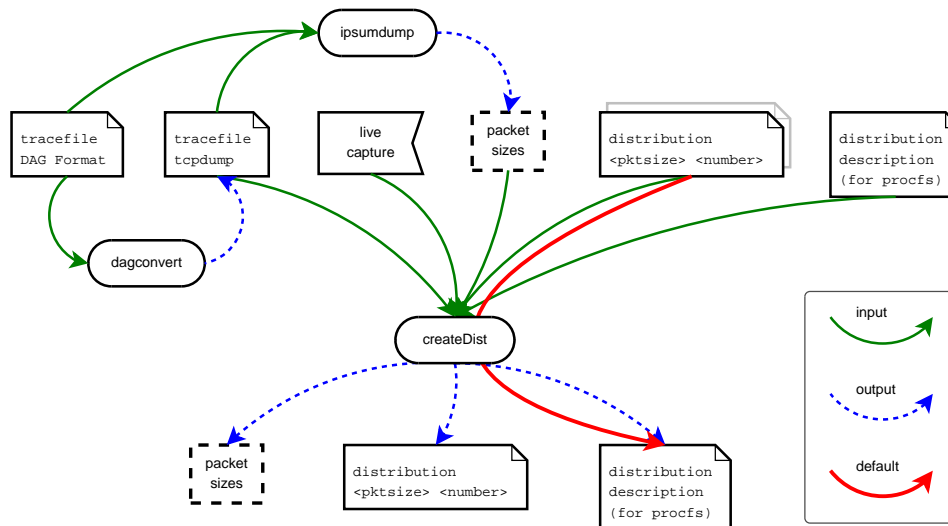


Figure A.1: Flow diagram of the input and output types of the `createDist` application.

trace: The input is now a trace file in pcap format. The size of every IP packet in the stream will be counted. Since there is no library to process DAG¹ trace files, this tool can only read pcap formatted trace files.

live: This means a capturing session is established (root privileges are needed for opening the socket) and the packet sizes of the online captured IP packets are counted.

The input and output type are defined via the `-I` and `-O` options (see Section A.1.3). With type sizes as well as dist as input it is possible to supply multiple files concatenated together as input.

A.1.2 Functionality of `createDist`

Depending on the chosen input, different actions have to be performed. In any case the goal is to obtain the distribution as explained for type `dist`. From that point the distribution can be displayed or transformed to the input needed by the enhanced Linux Kernel Packet Generator. If the requested output type is `sizes`, this program produces packet sizes according to the distribution and acts like the generator. For reading the packet sizes it is either necessary to read in a text or pcap trace file and count the packet sizes or the network layer has to be accessed using `libpcap`. For both `libpcap` based cases—reading trace files and live capturing—a callback

¹ *DAG* is a type of packet capturing hardware build by Endace Measurement Systems Capturing. It also defines a trace file format which is indicated by the `.erf` filename extension.

function² simply discards all non-IP packets³ and increases the counter according to the length of the IP packet.

A.1.3 Command-line Options

a) General Options:

- v *verbose output*. Status information, progress information, warning output, packet counters, pcap version information, and statistics are written to standard error.
- t *file specify trace output file*. With this option it is possible to specify a file into which the raw packet data is written.
- z [*int*] *do additional compression [level] on any packet*.
- c [*int*] *do additional copy operations [number] on any packet*. You can add very fine-grained load per packet with this option.
- ts1 [*int*] *write [bytes] of every packet to file*. When this option is set, only the first *n* bytes of every packet are written to the file specified by the -t option.
- h *show help*.
- n *int number of packet sizes to generate (default: 10 000 000)*. This only applies if sizes is chosen as output type.

b) Input and Output Options:

- i *file specify input file*. Instead of reading from standard in read (the input) from the specified file.
- o *file specify output file*. Instead of writing to standard out write (the output) to the specified file.
- I *dist|procfs|sizes|trace|live specify input type (default: dist)*.
- O *dist|procfs|sizes specify output type (default: procfs)*.
- fs *char specify field separator character (default: space)*. Applies to type dist only.
- if *string specify capturing interface*. This is necessary when doing live capturing.

²The *callback function* is a mechanism of the libpcap. This function is called for every packet. A pointer to the packet data is handed over to this function. The programmer can write this function as he likes, thus providing an interface to process the packets.

³*non-IP packets* in opposition to IP packets, do not have the EtherType field of the Ethernet frame set to 0x0800 (indicating an IP packet).

- s1 int *specify capturing snaplength*. See `man pcap` for details on this option.
- f string *specify capturing filter*. See `man tcpdump` for the filter description language.
- s *surround procfs output by pgset()’s*. Useful when using the output with the script supplied in `pktgen.txt` from the Linux Kernel Packet Generator.

c) Distribution Options: read Section 4.2.2 for explanations of the variables.

- max int *specify maximum packet size (default: 1500)*. Sets the maximum regarded packet size N_{ps} .
- prec int *specify precision/resolution of arrays for procfs (default: 1000)*. The size of the arrays (ρ) used for generating the different packet sizes.
- hwidth int *width of bins (default: 20)*. σ_{bin} .
- outlb double *when over this percentage \rightarrow packet size is an outlier (default: 0.0020)*. $p_{\Omega_{bound}}$.

A.2 Linux Kernel Packet Generator – Packet Size Distribution Enhancement

This patch was only tested on a x86 PC with a v2.6.8 Linux Kernel. It probably will not work with the newer versions of the Linux Kernel. Since Robert Olsson—the author of the Linux Kernel Packet Generator—recently did a major rework of the code as described in [Ols05], the modifications described in this thesis are not compatible with the new original version.

Details on the enhancement are described in Section A.2.3 and the theory of representing the distributions as well as the calculation of the presentation of these distributions for the Linux Kernel Packet Generator are explained in Section 4.2.

A.2.1 How it Works

When a new packet is to be generated a new packet size has to be determined. Therefore, an entry of the so called outliers array is chosen randomly. This array contains packet size values of the packet sizes which appear very often in the distribution which is to be represented. If a `-1` is read in this array, none of these packet sizes is chosen, so it is necessary to choose randomly again. But this time another array—the so called bins array—is used. In this array the entries represent a whole bin of packet sizes. For convenience the smallest packet size in a bin represents it. Additionally, random jitter of maximal the width of such a bin (called `hist_width` below) needs to be added to this obtained packet size.

A.2.2 How to Use the new Enhancements

1. Read the original pktgen.txt.
2. The following three new commands for the /proc interface are added:

dist (e.g. `pgset "dist 1000 20 1500 33 75"`)

This is used to set up the Linux Kernel Packet Generator for excepting the distributions entered by the `outl` and `hist` commands. The syntax is:

```
dist <precision> <hist_width> <max_pkt_size> <#outliers> <#histos>
```

With `<precision>` the size of the array used for generating the different packet sizes is set. This is directly influencing how high the resolution of the different entries is. The `<hist_width>` sets the width of a bin. The `<max_pkt_size>` sets the maximum packet size. The `<#outliers>` and `<#histos>` define how many lines of `outl` and `hist` have to follow until the input distribution is complete.

outl (e.g. `pgset "outl 40 179"`) Syntax:

```
outl <pkt_size> <#cells>
```

This instructs the generator to fill `<#cells>` of the outliers array with the packet size `<pkt_size>`

hist (e.g. `pgset "hist 40 91"`) Syntax:

```
hist <pkt_size> <#cells>
```

This instructs the generator to fill `<#cells>` of the histos array with the packet size `<pkt_size>` to which jitter will be added.

3. To activate the distribution you have to switch the `PKTSIZE_REAL` flag with

```
pgset "flag PKTSIZE_REAL"
```

This will only succeed if the distribution is complete and correct indicated by the `DIST_READY` flag.

A.2.3 Changes Made

To use the packet size distributions it was necessary to change the source code of the Linux Kernel Packet Generator module [Ols02]. [SBP05] was very helpful when performing the coding. Please refer to the original source code to comprehend the following changes:

- A new `struct` for input array entries, consisting of two integers for packet size and number of cells in the arrays used for generation, was added.
- A general `struct` for controlling the distribution and its state was subjoined. The precision ρ , the size of the bins σ_{bin} , the maximum packet size N_{ps} , the number of outliers n_{Ω} , the number of bins n_{bin} as well as indexes to the arrays and the two input arrays themselves are saved in this `struct`.
- Two new flags were introduced: one to indicate that the entered distribution is ready, and one to activate the usage of the packet size distribution.
- The two generation arrays and a byte counter to the existent control `struct` of the packet generator.
- The new function `mod_cur_pktsize()` which is modifying the actual packet size according to the principle shown in Figure 4.3 utilizing the `net_random()` function of the kernel was written.
- A check for each packet to calculate a new packet size if both new flags are set (in function `fill_packet()`) was adjoined.
- Byte counting after the generation of the packet was added. This is necessary because calculating the data rate by multiplying the number of generated packets with the preset packet size is not possible any longer, since the packet size changes per packet now.
- The accounting processing at the end of packet generation was adjusted to get the right results (this was somewhat screwing up my brain, because of wild bit shifts to preserve reasonable accuracy for the output).
- The function `calculate_ra_arrays()` which fills the arrays used for packet size inquiry was implemented. If finished successfully, the flag which indicates that the distribution is ready is set.
- The function `check_dist_complete()` which checks if the entered distribution is complete, and if so calls `calculate_ra_arrays()` was written.
- The whole input stuff for the distributions was added to `proc_write()`.
- The initialization was added to `init()` as well as the clearance to `cleanup()`.

A.2.4 How to Compile and Install the New Module

1. Download the source code from <http://www.net.in.tum.de/~schneifa/sources/pktgen-lkpg-dist-0.1.tar.gz>

2. Untar the archive:

```
tar -xvzf pktgen-lkpg-dist-0.1.tar.gz
```

3. Copy the new pktgen.c over the old:

```
cp pktgen-lkpg-dist-0.1/pktgen.c \  
  /usr/src/linux/net/core/pktgen.c
```

4. Compile the new pktgen.c:

```
cd /usr/src/linux/net/core  
make -C /usr/src/linux SUBDIRS=$PWD modules
```

5. Install the new module:

```
cd /usr/src/linux  
make modules_install
```

A.3 cpusage

This tool reads out the CPU states ticks counter of the operating system and calculates the percentages spend in each `CPU_STATE`. This is done each half second. The output is send to standard out. Supplementary the minimum, maximum, and average percentages are displayed. The recording for the average measuring can be limited to start only if the idle value drops under a certain limit (set with the `-l` option) and stops if the idle value returns over that limit again.

A.3.1 Command-line Options

- `-o` Format the output machine-readable. No CPU state names are printed and only colons separate the values.
- `-l int` Set limits for average usage snapping.
- `-a` Build average over the whole time, same as `-l 100`.
- `-h` Prints the help

A.3.2 Building and Installing

1. Download the tarball from
<http://www.net.in.tum.de/~schneifa/sources/cpusage-0.1.tar.gz>
2. Untar the archive:

```
tar -xvzf cpusage-0.1.tar.gz
```
3. Build the program:

```
cd cpusage-0.1
make (please use GNU make if available)
```
4. Install the program:

```
cp cpusage <target>
```

A.4 trimusage

The awk script `trimusage.awk` (refer to [RCSR01] for an introduction to awk) is used to postprocess and correct the output of `cpusage`. This script was applied utilizing the `for i in ... sh-functionality` on all `cpusage` output files of a measurement:

```
for i in *.usage.out; do awk -f trimusage.awk $i > $i.trimmed; done
```

A sketch of the procedure of this script is given in Section 5.2 and a listing of the code follows:

```
1  # trimusage.awk
   # by Fabian Schneider (fabian@net.in.tum.de)

   BEGIN { limit = 95          # limit for the idle value
5      FS = ":"                # field separator
        set = 0                # current set of lines under limit
        longestset = 0        # longest set in file
        last = 0              # lines under limit of the current set
        longest = 0           # highest last counter in file
10     }

    /---/ { next }             # ignore lines with "---"
    /Min/ { next }            # ignore lines with "Min"
    /Max/ { next }            # ignore lines with "Max"
15   /Avg/ { next }           # ignore lines with "Avg"

    { # determin number of cpustates (7 for Linux, 5 for FreeBSD)
      if ( NR == 1 ) {        # only in first line
```

```

20         cpustates = NF           # number of fields in first line
        for ( i=1; i <= cpustates; i++ ) {
            min[i] = 100           # prepare min values
            max[i] = 0             # prepare max values
        }
    }
25
    print $0 # print out the line currently read

    # find min and max values.
    for ( i=1; i <= cpustates; i++ ) {
30        if ( $(i) > max[i] )
            max[i] = $(i)
        if ( $(i) < min[i] )
            min[i] = $(i)
    }
35
    # find the sets
    if ( $4 < limit ) {           # idle value under limit
        if ( last == 0 ) {        # new shortfall -> new set
            last ++               # increment last counter, now 1
40            set ++              # increment set number
            # save values from line
            for ( i=1; i <= cpustates; i++)
                values[set, last, i] = $(i)
        } else {                 # last line was under limit, too
45            last ++             # increment last counter
            # save values from line
            for ( i=1; i <= cpustates; i++)
                values[set, last, i] = $(i)
            # update longest and longestset if this set
            # is the longest
50            if ( last > longest ) {
                longest = last
                longestset = set
            }
55        }
    } else {                     # idle value over limit
        if ( last != 0 )         # last line was under limit
            last = 0             # reset last counter to 0
    }
60    }

    END { print "---Summary----"
          printf( "Min" )
          for ( i=1; i <= cpustates; i++)
65            printf( ":%.1f", min[i] )

          printf( "\nMax" )

```

```
    for ( i=1; i <= cpustates; i++)  
        printf( ":%.1f", max[i] )  
70  
  
    # sum up all values from longestset in sum[i]  
    for (l=1; l <= longest; l++) {  
        for (i=1; i <= cpustates; i++) {  
            sum[i] += values[longestset, l, i]  
75        }  
    }  
  
    # calculate new average in avg[i]  
    for (i=1; i <= cpustates; i++) {  
80        avg[i] = sum[i] / longest  
    }  
  
    printf( "\nAvg" )  
    for ( i=1; i <= cpustates; i++)  
85        printf( ":%.1f", avg[i] )  
}
```

B Plots

Arriving at the measurements and the results presented in Chapter 6 was not straight forward. Multiple measurements had to be repeated, due to measurement interference of other applications and/or of false cabling. Furthermore, some insights had to be recognized. This chapter shows some but not all of the results of the measurements which have not yet been described in full detail.

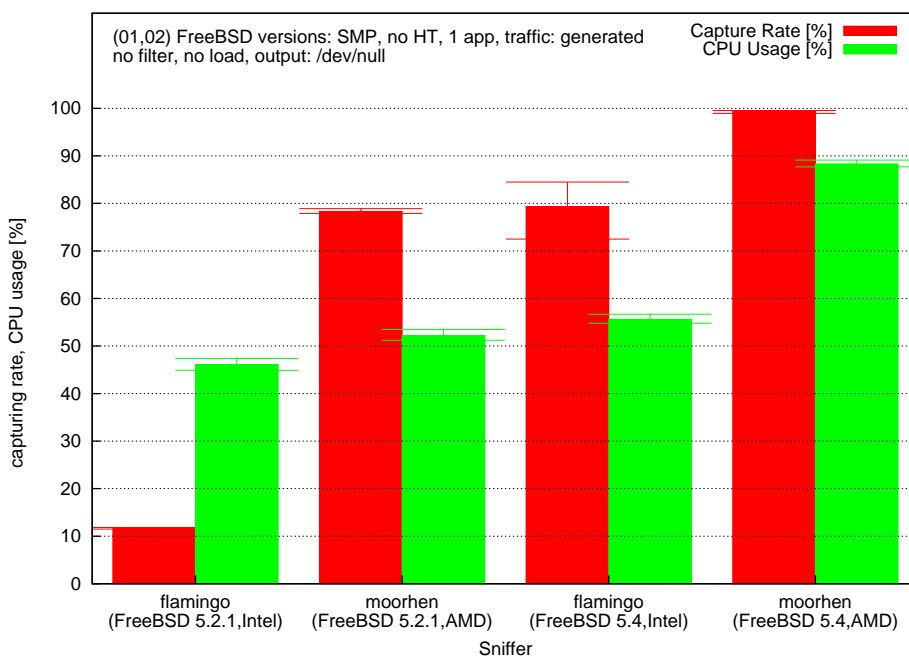
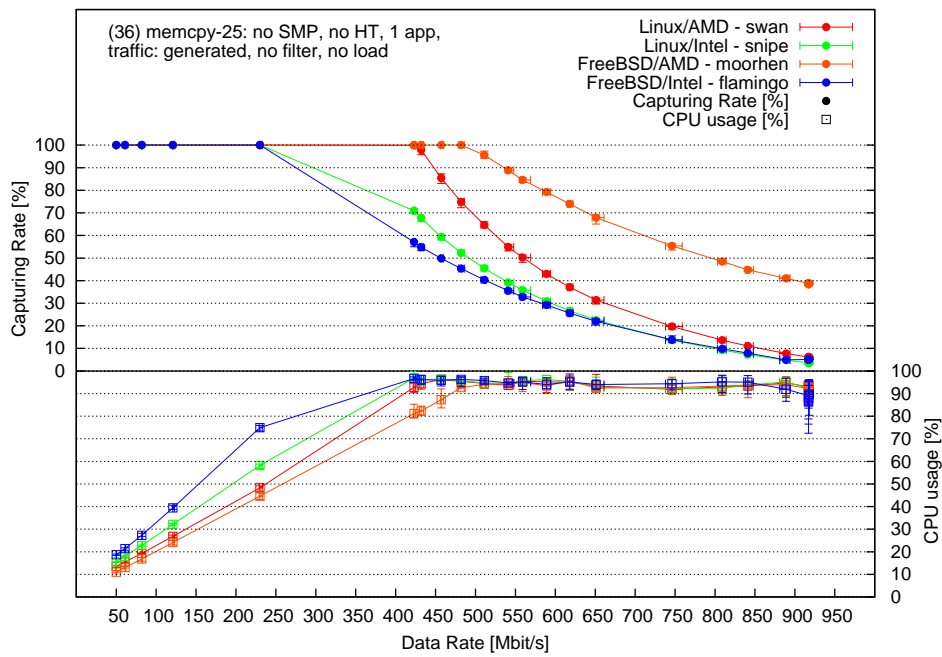
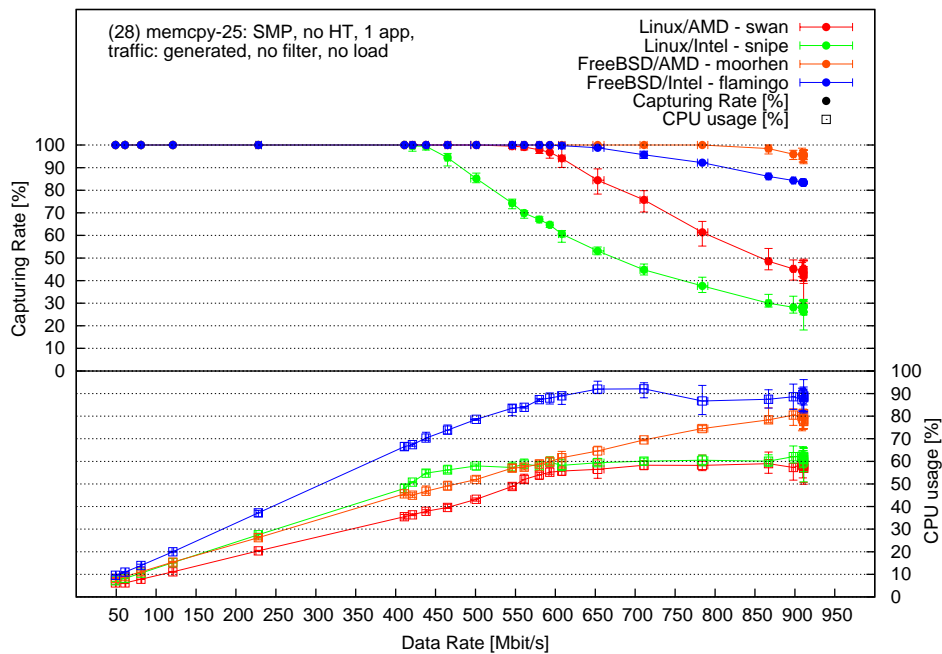


Figure B.1: Histogram of the performance of the sniffers using different FreeBSD versions measured at maximum generation rate. A relevant increase in the data rate can be observed when comparing version 5.2.1 with 5.4.

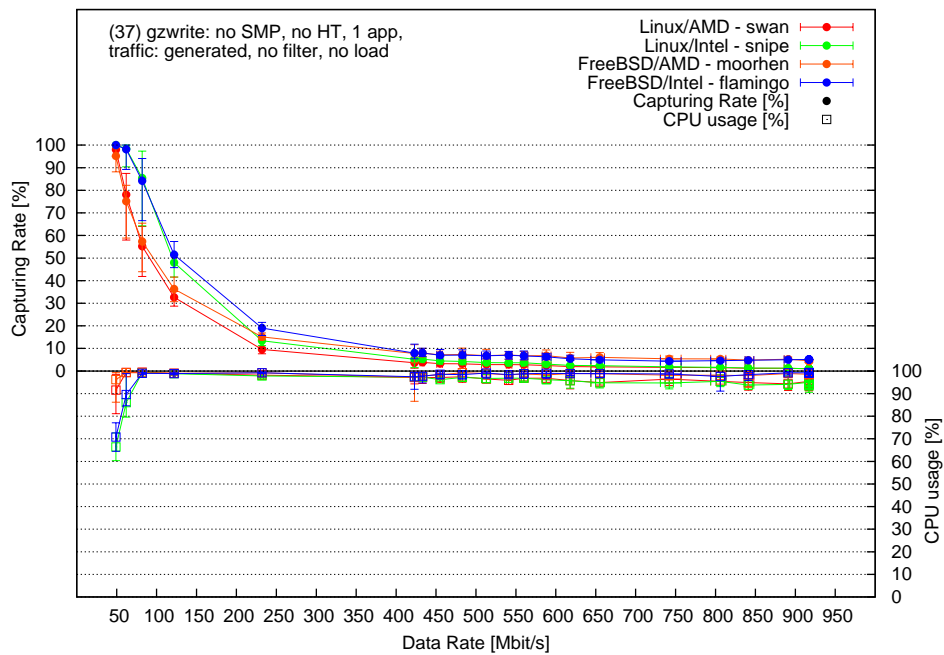


(a) single processor mode

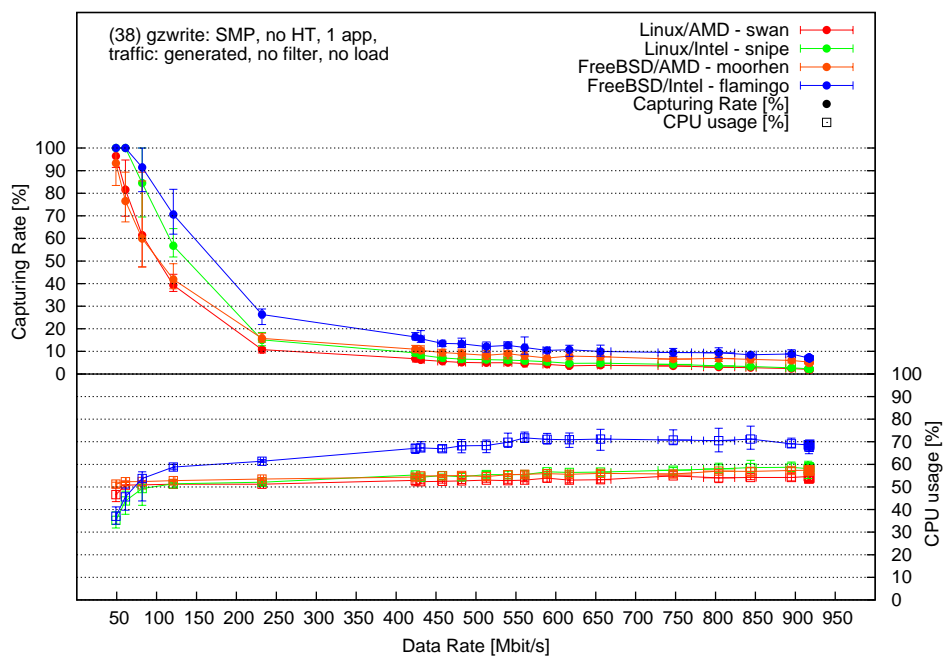


(b) dual processor mode

Figure B.2: Linespoints plots of the performance of the sniffers with 25 additional packet copies. The capturing rate and the CPU usage are plotted against the increasing data rate.



(a) single processor mode



(b) dual processor mode

Figure B.3: Linespoints plots of the performance of the sniffers with additional zlib compression (level 9).

C Acknowledgements

First of all, I want to thank the whole research group of professor Anja Feldmann (Technische Universität München (TUM), Department of Informatics) for their helpfulness and patience. Many questions discussed casually were helpful getting a picture of the coherences. Likewise, they helped coming down to earth when I was screwed up in non-trivial thoughts.

Then, I want to thank the authors of GNU awk [RCSR01] for their “*I need a fast result with less effort*”-tool, the authors of GNU make [SMS02] for being able to use automatisms compiling programs and documents, the authors of cvs [Ced05] for their version control and storage system, and, of course, the authors of gnuplot [WK04] for their plotting tool I used for all my plots.

It was also useful to read Vern Paxsons “Strategies for sound internet measurement” [Pax04] and some RFC’s concerning terminology ([Bra91], [BM99] and [MP00]).

In almost the same manner, I want to thank Donald Knuth and Leslie Lamport for \TeX and \LaTeX . The \LaTeX Companion [GMS00] and different Internet manuals have really been viable for some typesetting problems. Futhermore, I want to thank the authors of KOMA-Script [NKKM04] with which this thesis is typeset.

My special thanks go to Britta Liebscher for having practical advices when typesetting this thesis with \LaTeX .

List of Figures

2.1	Block diagram: BPF	6
2.2	Block diagram: LSF	6
2.3	Network diagram: Regular sniffer setup	11
2.4	Tabular: Sniffer differences	12
2.5	Block diagrams: Comparison of Xeon and Opteron design	13
3.1	Network diagram: Measurement sniffer setup	19
3.2	Sequence diagram: Measurement cycle	20
4.1	Scatterplot: Example packet size distribution	25
4.2	Histogram plot: Percentages of packets grouped by size	25
4.3	Flow diagram: Selecting a new packet size	26
4.4	Plots: Comparing generated vs. original distribution	29
4.5	Histogram: Output (data and packet rate) of the LKPG	30
5.1	Sample output: <code>cpusage</code>	32
6.1	Line plot: data/packet rate against inter-packet gaps	37
6.2	Linespoint plots: Performance of the sniffers (no improvements)	39
6.3	Linespoint plots: Performance of the sniffers (with buffers)	41
6.4	Linespoint plots: Performance of the sniffers (increasing buffer size)	43
6.5	Listing: BPF expression used for filter measurements	45
6.6	Linespoint plots: Performance of the sniffers (with filtering)	46
6.7	Linespoint plots: Performance of the sniffers (two applications)	48
6.8	Linespoint plots: Performance of the sniffers (four applications)	48
6.9	Linespoint plots: Performance of the sniffers (eight applications)	48
6.10	Linespoint plots: Performance of the sniffers (with additional copy operations)	50
6.11	Linespoint plots: Performance of the sniffers (with additional compression – level 3)	51
6.12	Linespoint plots: Performance of the sniffers (additional compression via pipes)	52
6.13	Histogram: maximum speed writing to disk	53
6.14	Linespoint plots: Performance of the sniffers (writing packet header to disk)	55
6.15	Linespoint plots: Performance of the sniffers (mmaped libpcap)	57
6.16	Linespoint plot: Performance of the sniffers (using Hyperthreading)	58

A.1	Flow diagram: Input and Output Types of <code>createDist</code>	64
B.1	Histogram: Performance of the sniffers (different FreeBSD versions)	73
B.2	Linespoint plots: Performance of the sniffers (with additional copy operations)	74
B.3	Linespoint plots: Performance of the sniffers (with additional compression – level 9)	75

Bibliography

For easy looking up citations an Internet link to the location where the appropriate document or the appropriate application was found is supplied. Please be aware of the fact that the cited Internet contents may change. Therefore, the date in parentheses tells the point in time on which the URL was accessed at last.

- [AGK⁺02] Luca Abeni, Ashvin Goel, Charles Krasnic, Jim Snow, and Jonathan Walpole. A measurement-based analysis of the real-time performance of Linux. In *Proceedings of the Eighth IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS 2002)*, page 133 et seqq., San Jose, CA, USA, September 2002. IEEE Computer Society. <http://syn.cs.pdx.edu/~jsnow/publications/rtas2002.pdf> (on 30 Sep 2005).
- [BM99] S. Bradner and J. McQuaid. RFC 2544: Benchmarking methodology for network interconnection devices, March 1999. <http://www.ietf.org/rfc/rfc2544.txt> (on 05 Oct 2005).
- [Bra91] S. Bradner. RFC 1242: Benchmarking terminology for network interconnection devices, July 1991. <http://www.ietf.org/rfc/rfc1242.txt> (on 05 Oct 2005).
- [CB96] Mark Crovella and Azer Bestavros. Self-Similarity in World Wide Web Traffic: Evidence and Possible Causes. In *Proceedings of the ACM International Conference on Measurement and Modeling of Computer Systems (SIGMETRICS 1996)*, pages 160–169, Philadelphia, PA, USA, May 1996. <http://www.cs.bu.edu/fac/best/res/papers/sigmetrics96.ps> (on 30 Sep 2005).
- [Ced05] Per Cederqvist. *Version Management with CVS*, 2005. <http://ximbiot.com/cvs/manual/> (on 30 Sep 2005).
- [CHC⁺04] Yu-Chung Cheng, Urs Hölzle, Neal Cardwell, Stefan Savage, and Geoffrey M. Voelker. Monkey see, monkey do: A tool for TCP tracing and replaying. In *Proceedings of the USENIX Annual Technical Conference, General Track (USENIX 2004)*, pages 87–98, Boston, MA, USA, June 2004. USENIX. <http://www.usenix.org/publications/library/proceedings/usenix04/tech/general/cheng.html> (on 30 Sep 2005).
- [CLRS03] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms*. MIT Press, Cambridge, MA, USA, second edition, 2003.
- [CLS00] William S. Cleveland, Dong Lin, and Don X. Sun. IP packet generation: statistical models for TCP start times based on connection-rate superposition. In *Proceedings of the ACM International Conference on Measurement and Modeling of Computer Systems (SIGMETRICS 2000)*, pages 166–177, Santa Clara, CA, USA, June 2000. <http://citeseer.ist.psu.edu/cleveland00ip.html> (on 30 Sep 2005).

- [CMT98] K. Claffy, Greg Miller, and Kevin Thompson. the nature of the beast: recent traffic measurements from an internet backbone. In *Proceedings of the Internet Society's Eighth Annual Networking Conference (INET 1998)*, Geneva, Switzerland, July 1998. Internet Society. <http://www.caida.org/outreach/papers/1998/Inet98/Inet98.pdf> (on 03 Oct 2005).
- [Cok] Russell Coker. *bonnie++*, man page. <http://www.coker.com.au/bonnie++/> (on 02 Nov 2005).
- [Der03] Luca Deri. PF_RING: Passive packet capture on Linux at high speeds. http://www.ntop.org/PF_RING.html (on 30 Sep 2005), April 2003. Patches see www.ntop.org.
- [Der04] Luca Deri. Improving passive packet capture: Beyond device polling. In *Proceedings of the Fourth International System Administration and Network Engineering Conference (SANE 2004)*, Amsterdam, The Netherlands, September 2004. <http://luca.ntop.org/Ring.pdf> (on 30 Sep 2005).
- [Der05] Luca Deri. nCap: Wire-speed packet capture and transmission. In *Proceedings of the IEEE/IFIP Workshop on End-to-End Monitoring Techniques and Services (IM 2005, E2EMON)*. IEEE Computer Society, May 2005. <http://http://luca.ntop.org/nCap.pdf> (on 30 Sep 2005).
- [DFN] Deutsches Forschungsnetz, Welcome to DFN. <http://www.dfn.de/content/en/enhome/index.html> (on 30 Sep 2005).
- [DV04] Loris Degioanni and Gianluca Varenni. Introducing scalability in network measurement: toward 10 Gbps with commodity hardware. In *Proceedings of the Fourth ACM SIGCOMM Conference on Internet Measurement (IMC 2004)*, pages 233–238, Taormina, Italy, 2004. ACM Press. <http://www.imconf.net/imc-2004/papers/p233-degioanni.pdf> (on 30 Sep 2005).
- [FGB⁺03] Wu-Chang Feng, Ashvin Goel, Abdelmajid Bezzaz, Wu-Chi Feng, and Jonathan Walpole. TCPivo: a high-performance packet replay engine. In *Proceedings of the ACM Workshop on Models, Methods and Tools for Reproducible Network Research (SIGCOMM 2003, MoMe Tools)*, pages 57–64, Karlsruhe, Germany, August 2003. ACM Press. http://www.thefengs.com/wuchang/work/tcpivo/TCPivo_SIGCOMM_MoMeTools_2003.pdf (on 30 Sep 2005).
- [FP01] Sally Floyd and Vern Paxson. Difficulties in simulating the internet. *IEEE/ACM Transactions on Networking (TON)*, 9(4):392–403, August 2001. http://www.icir.org/floyd/papers/simulate_2001.pdf (on 30 Sep 2005).
- [GAK⁺02] Ashvin Goel, Luca Abeni, Charles Krasic, Jim Snow, and Jonathan Walpole. Supporting time-sensitive applications on a commodity OS. *ACM SIGOPS Operating Systems Review*, 36(SI):165–180, December 2002. <http://syn.cs.pdx.edu/~jsnow/publications/osdi2002.pdf> (on 30 Sep 2005).
- [GAR05] Jean-loup Gailly, Mark Adler, and Greg Roelofs. zlib – home page. especially the manual and the usage example, 2005. <http://www.zlib.net/> (on 12 Oct 2005).

- [GMS00] Michel Goossens, Frank Mittelbach, and Alexander Samarin. *Der L^AT_EX Begleiter*. Addison Wesley Company, Munich, Germany, first edition, 2000. The title of the english original is *The L^AT_EX Companion*.
- [Ins01] Gianluca Insolvibile. The Linux Socket Filter: Sniffing bytes over the network. *Linux Journal*, 86, June 2001. <http://www.linuxjournal.com/article/4659> (on 30 Sep 2005).
- [Ins02a] Gianluca Insolvibile. Kernel Korner: Inside the Linux Packet Filter. *Linux Journal*, 94, February 2002. <http://www.linuxjournal.com/article/4852> (on 30 Sep 2005).
- [Ins02b] Gianluca Insolvibile. Kernel Korner: Inside the Linux Packet Filter, part 2. *Linux Journal*, 95, March 2002. <http://www.linuxjournal.com/article/5617> (on 30 Sep 2005).
- [JLM94] Van Jacobson, Craig Leres, and Steven McCanne. libpcap, Initial public release 1994. Currently (30 Sep 2005) available at <http://www.tcpdump.org>.
- [Koh] Eddie Kohler. ipsumdump. <http://www.cs.ucla.edu/~kohler/ipsumdump/> (on 30 Sep 2005).
- [Kor05] Stefan Kornexl. High-performance packet recording for network intrusion detection. Diplomarbeit, Technische Universität München, Munich, Germany, January 2005. <http://www.net.in.tum.de/teaching/projects/docs/kornexl.pdf>.
- [KPD⁺05] Stefan Kornexl, Vern Paxson, Holger Dreger, Anja Feldmann, and Robin Sommer. Building a time machine for efficient recording and retrieval of high-volume network traffic. Technical Report TUM-I0509, Technische Universität München, Fakultät für Informatik, June 2005. <http://wwwbib.informatik.tu-muenchen.de/infberichte/2005/TUM-I0509.pdf> (on 30 Sep 2005).
- [Kuh04] Bernhard Kuhn. The Linux real time interrupt patch, January 2004. <http://linuxdevices.com/articles/AT6105045931.html> (on 30 Sep 2005).
- [Köh05] Axel Köhler. Opteron Technologie. Slides of a talk at the Introduction of the Linux cluster at computing center in Cologne, February 2005. http://www.uni-koeln.de/rrzk/server/clio/intro_talk/sun_opteron.pdf (on 01 Oct 2005).
- [Lan04] Sebastian Lange. Analyse und Performancetests von Paket Replay Programmen. Ausarbeitung zum Systementwicklungsprojekt, Technische Universität München, Munich, Germany, November 2004. http://www.net.in.tum.de/teaching/projects/docs/lange_SEP_nov2004.pdf (on 07 Oct 2005).
- [LBN] Lawrence Berkeley National Laboratory, Bro Intrusion Detection System. <http://www.bro-ids.org/> (on 30 Sep 2005).
- [LRZ] Leibniz Rechenzentrum, The Leibniz Computing Centre. <http://www.lrz-muenchen.de/wir/intro/en/> (on 30 Sep 2005).

- [LTWW93] Will E. Leland, Murad S. Taqqu, Walter Willinger, and Daniel V. Wilson. On the self-similar nature of Ethernet traffic. In *Proceedings of the ACM Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication (SIGCOMM 1993)*, pages 183–193, San Francisco, CA, USA, March 1993. ACM. <http://citeseer.ist.psu.edu/leland93selfsimilar.html> (on 30 Sep 2005).
- [MBH⁺02] Deborah T. Marr, Frank Binns, David L. Hill, Glenn Hinton, David A. Koufaty, J. Alan Miller, and Michael Upton. Hyper-threading technology architecture and microarchitecture. *Intel Technology Journal (ITJ)*, 06(Issue 01):4–15, February 2002. ftp://download.intel.com/technology/itj/2002/volume06issue01/vol6iss1_hyper_threading_technology.pdf (on 10 Oct 2005).
- [MJ93] Steven McCanne and Van Jacobson. The BSD packet filter: A new architecture for user-level packet capture. In *Proceedings of the Usenix Technical Conference Winter 1993, (USENIX 1993 (Winter))*, pages 259–270, San Diego, CA, USA, January 1993. <http://www.tcpdump.org/papers/bpf-usenix93.pdf> (on 30 Sep 2005).
- [MP00] R. Mandeville and J. Perser. RFC 2889: Benchmarking methodology for LAN switching devices, August 2000. <http://www.ietf.org/rfc/rfc2889.txt>.
- [MR97] Jeffrey C. Mogul and K.K. Ramakrishnan. Eliminating receive live-lock in an interrupt-driven kernel. *ACM Transactions on Computer Systems*, 15(3):217–252, August 1997. <http://www.stanford.edu/class/cs240/readings/mogul.pdf> (on 04 Oct 2005).
- [NKKM04] Frank Neukam, Markus Kohm, Axel Kielhorn, and Jens-Uwe Morawski. *Das KOMA-Script Paket*, September 2004. also available in english: *The KOMA-Script bundle*, <ftp://ftp.ctan.org/tex-archive/macros/latex/contrib/koma-script/scrguien.pdf> (on 04 Oct 2005).
- [Ols02] Robert Olsson. Linux kernel packet generator. Linux Kernel sources, v2.6.8, documentation in kernel tree: <Documentation/networking/pktgen.txt>, last changes 2002. Uppsala University, Sweden.
- [Ols05] Robert Olsson. pktgen the Linux packet generator. In *Proceedings of the Linux Symposium – Volume Two*, pages 11–24, Ottawa, Canada, July 2005. http://www.linuxsymposium.org/2005/linuxsymposium_procv2.pdf (on 02 Oct 2005).
- [Pax99] Vern Paxson. Bro: a system for detecting network intruders in real-time. *Computer Networks*, 31(23-24):2435–2463, December 1999. <ftp://ftp.ee.lbl.gov/papers/bro-CN99.ps.gz> (on 30 Sep 2005).
- [Pax04] Vern Paxson. Strategies for sound internet measurement. In *Proceedings of the Fourth ACM SIGCOMM Conference on Internet Measurement (IMC 2004)*, pages 263–271, Taormina, Sicily, Italy, 2004. ACM Press. <http://www.icir.org/vern/papers/meas-strategies-imc04.pdf> (on 30 Sep 2005).
- [PF95] Vern Paxson and Sally Floyd. Wide area traffic: the failure of Poisson modeling. *IEEE/ACM Transactions on Networking (TON)*, 3(3):226–244, June 1995. <http://citeseer.ist.psu.edu/paxson95widearea.html> (on 30 Sep 2005).

- [RCSR01] Arnold Robbins, Diane Close, Richard Stallman, and Paul Rubin. *The GNU Awk User's Guide*. GNU, FSF, 3 edition, March 2001. <http://www.gnu.org/software/gawk/manual/gawk.html> (on 30 Sep 2005).
- [RD01] Fulvio Risso and Loris Degioanni. Winpcap – an architecture for high performance network analysis. In *Proceedings of the Sixth IEEE International Symposium on Computers and Communications (ISCC 2001)*, pages 686–693, Hammamet, Tunisia, July 2001. IEEE Computer Society. <http://www.winpcap.org/docs/iscc01-wpcap.pdf> (on 30 Sep 2005).
- [Riz01] Luigi Rizzo. Device polling support for FreeBSD. In *Proceedings of the Main European BSD Conference (EuroBSDCon 2001)*, Brighton, UK, 2001. <http://info.iet.unipi.it/~luigi/polling/> (on 30 Sep 2005).
- [SB04] Joel Sommers and Paul Barford. Self-configuring network traffic generation. In *Proceedings of the Fourth ACM SIGCOMM Conference on Internet Measurement (IMC 2004)*, pages 68–81, Taormina, Sicily, Italy, October 2004. ACM Press. <http://www.cs.wisc.edu/~jsommers/pubs/p68-sommers.pdf> (on 30 Sep 2005).
- [SBP05] Peter Jay Salzman, Michael Burian, and Ori Pomerantz. *The Linux kernel module programming guide*. TLDP, v2.6.1 edition, January 2005. <http://www.tldp.org/LDP/lkmpg/2.6/lkmpg.pdf> (on 30 Sep 2005).
- [Sch04] Fabian Schneider. Analyse der Leistung von BPF und libpcap in Gigabit-Ethernet Umgebungen. Ausarbeitung zum Systementwicklungsprojekt, Technische Universität München, Munich, Germany, October 2004. <http://www.net.in.tum.de/teaching/projects/docs/schneider.ps> (on 30 Sep 2005).
- [SMS02] Richard Stallman, Roland McGrath, and Paul Smith. *GNU Make Documentation*. GNU, FSF, December 2002. <http://www.gnu.org/software/make/manual/make.html> (on 30 Sep 2005).
- [SOK01] Jamal Hadi Salim, Robert Olsson, and Alexey Kuznetsov. Beyond softnet. In *Proceedings of the Fifth Annual Linux Showcase & Conference*, Oakland, CA, November 2001. http://www.linuxshowcase.org/2001/full_papers/jamal/jamal.pdf (on 30 Sep 2005).
- [Som05] Robin Sommer. *Viable Network Intrusion Detection in High-Performance Environments*. PhD thesis, Technische Universität München, June 2005. <http://www.icir.org/robin/papers/thesis.pdf> (on 03 Nov 2005).
- [Tan01] Andrew S. Tanenbaum. *Modern Operating Systems*. Prentice Hall, Upper Saddle River, NJ, USA, second edition, 2001.
- [Tan03] Andrew S. Tanenbaum. *Computer Networks*. Prentice Hall Professional Technical Reference, Upper Saddle River, NJ, USA, fourth edition, 2003.
- [Tur04] Aaron Turner. Tcpreplay: Pcap editing and replay tools for *NIX, version 2.x, May 2004. <http://tcpreplay.sourceforge.net> (on 17 Oct 2005).
- [Vil02] Christian Vilsbeck. Alle Details zur AMD64-Architektur. *tecCHANNEL (www.tecchannel.de)*, September 2002. Available only online: <http://www.tecchannel.de/server/hardware/401971/index.html> (on 01 Oct 2005).

- [Vil03] Christian Vilsbeck. Test: Xeon 3,06 GHz L3 vs. Opteron 244. *tecCHANNEL* (www.tecchannel.de), May 2003. Available only online: <http://www.tecchannel.de/server/hardware/402096/index.html> (on 01 Oct 2005).
- [War02] James C. Warner. Sources for `top`. `top` for Linux, e.g. Source package `procps` from Debian GNU Linux, 2002.
- [WK04] Thomas Williams and Colin Kelley. *GNUPLOT: An Interactive Plotting Program, version 4.0*, 2004. <http://www.gnuplot.info/docs/gnuplot.html> (on 30 Sep 2005).
- [Woo] Phil Wood. `libpcap-mmap`. Los Alamos National Lab. <http://public.lanl.gov/cpw/see2>. (on 30 Sep 2005).
- [WTSW97] Walter Willinger, Murad S. Taqqu, Robert Sherman, and Daniel V. Wilson. Self-similarity through high-variability: statistical analysis of Ethernet LAN traffic at the source level. *IEEE/ACM Transactions on Networking (TON)*, 5(1):71–86, April 1997. <http://citeseer.ist.psu.edu/willinger97selfsimilarity.html> (on 30 Sep 2005).