Comparing and Improving Current Packet Capturing Solutions based on Commodity Hardware

Lothar Braun, Alexander Didebulidze, Nils Kammenhuber, Georg Carle
Technische Universität München
Institute for Informatics
Chair for Network Architectures and Services
{braun,didebuli,kammenhuber,carle}@net.in.tum.de

ABSTRACT

Capturing network traffic with commodity hardware has become a feasible task: Advances in hardware as well as software have boosted off-the-shelf hardware to performance levels that some years ago were the domain of expensive special-purpose hardware. However, the capturing hardware still needs to be driven by a well-performing software stack in order to minimise or avoid packet loss. Improving the capturing stack of Linux and FreeBSD has been an extensively covered research topic in the past years. Although the majority of the proposed enhancements have been backed by evaluations, these have mostly been conducted on different hardware platforms and software versions, which renders a comparative assessment of the various approaches difficult, if not impossible.

This paper summarises and evaluates the performance of current packet capturing solutions based on commodity hardware. We identify bottlenecks and pitfalls within the capturing stack of FreeBSD and Linux, and give explanations for the observed effects. Based on our experiments, we provide guidelines for users on how to configure their capturing systems for optimal performance and we also give hints on debugging bad performance. Furthermore, we propose improvements to the operating system's capturing processes that reduce packet loss, and evaluate their impact on capturing performance.

Categories and Subject Descriptors

C.2.3 [Network Operation]: Network Monitoring

General Terms

Measurement, Performance

1. INTRODUCTION

Packet capture is an essential part of most network monitoring and analysing systems. A few years ago, using specialised hardware—e.g., network monitoring cards manufactured by Endace [1]—was mandatory for capturing Gigabit or Multi-gigabit network traffic, if little or no packet loss was a requirement. With recent development progresses in bus systems, multi-core CPUs and commodity network cards, nowadays off-the-shelf hardware can be used to capture network traffic at near wire-speed with little or no packet loss in 1 GE networks, too [2, 3]. People are even building monitoring devices based on commodity hardware that can be used to capture traffic in 10 GE networks [4, 5]

However, this is not an easy task, since it requires careful configuration and optimization of the hardware and software components involved—even the best hardware will suffer packet loss if its driving software stack is not able to handle the huge amount of network packets. Several subsystems including the network card driver, the capturing stack of the operating system and the monitoring application are involved in packet processing. If only one of these subsystems faces performance problems, packet loss will occur, and the whole process of packet capturing will yield bad results.

Previous work analysed [2, 4, 6] and improved [5, 7, 8] packet capturing solutions. Comparing these work is quite difficult because the evaluations have been performed on different hardware platforms and with different software versions. In addition, operating systems like Linux and FreeBSD are subject to constant changes and improvements. Comparisons that have been performed years ago therefore might today not be valid any longer. In fact, when we started our capturing experiments, we were not able to reproduce the results presented in several papers. When we dug deeper into the operating systems' capturing processes, we found that some of our results can be explained by improved drivers and general operating systems improvements. Other differences can be explained by the type of traffic we analysed and by the way our capturing software works on the application layer. While it is not a problem to find comparisons that state the superiority of a specific capturing solution, we had difficulties to find statements on why one solution is superior to another solution. Information about this topic is scattered throughout different papers and web pages. Worse yet, some information proved to be highly inconsistent, especially when from Web sources outside academia. We therefore encountered many difficulties when debugging the performance problems we ran into.

This paper tries to fill the gap that we needed to step over when we set up our packet capturing environment with Linux and FreeBSD. We evaluate and compare different capturing solutions for both operating systems, and try to summarise the pitfalls that can lead to bad capturing perfor-

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
IMC'10, November 1–3, 2010, Melbourne, Australia.
Copyright 2010 ACM 978-1-4503-0057-5/10/11 ...$10.00.

mance. The paper aims at future developers and users of capture systems and serves as a resource helping them not to repeat the pitfalls we and other researchers have encountered.

A special focus of this paper is on explaining our findings. We try to identify the major factors that influence the performance of a capturing solution, providing users of packet capture systems with guidelines on where to search for performance problems in their systems. Our paper also targets developers of capturing and monitoring solutions: We identify potential bottlenecks that can lead to performance bottlenecks and thus packet loss. Finally, we propose a modification that can be applied to popular capturing solutions. It improves capturing performance in a number of situations.

The remainder of this paper is organised as follows: Section 2 introduces the capturing mechanisms in Linux and FreeBSD that are in use when performing network monitoring, and presents related improvements and evaluations of packet capturing systems. Section 3 presents the test setup that we used for our evaluation in Section 4. Our capturing analysis covers scheduling issues in Section 4.1 and focuses on the application and operating system layer with low application load in Section 4.2. Subsequently, we analyse application scenarios that pose higher load on the system in Section 4.3, where we furthermore present our modifications to the capturing processes and evaluate their influence on capturing performance. In Section 4.4, we move downwards within the capturing stack and discuss driver issues. Our experiments result in recommendations for developers and users of capturing solutions, which are presented in Section 5. Finally, Section 6 concludes the paper with a summary of our findings.

2. BACKGROUND AND RELATED WORK

Various mechanisms are involved in the process of network packet capturing. The performance of a capturing process thus depends on each of them to some extent. On the one hand, there is the hardware that needs to capture and copy all packets from the network to memory before the analysis can start. On the other hand, there is the driver, the operating system and monitoring applications that need to carefully handle the available hardware in order to achieve the best possible packet capturing performance. In the following, we will introduce popular capturing solutions on Linux and FreeBSD in 2.1. Afterwards, we will summarise comparisons and evaluations that have been performed on the different solutions in Section 2.2.

2.1 Solutions on Linux and FreeBSD

Advances made in hardware development in recent years such as high speed bus systems, multi-core systems or network cards with multiple independent reception (RX) queues offer performance that has only been offered by special purpose hardware some years ago. Meanwhile, operating systems and hardware drivers have come to take advantage of these new technologies, thus allowing higher capturing rates.

Hardware:

The importance of carefully selecting suitable capturing hardware is well-known, as research showed that different hardware platforms can lead to different capturing performance.

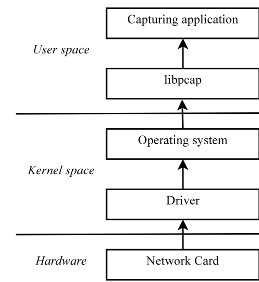

Figure 1: Subsystems involved in the capturing process

Schneider et al. [4] compared capturing hardware based on Intel Xeon and AMD Opteron CPUs with otherwise similar components. Assessing an AMD and an Intel platform of comparable computing power, they found the AMD platform to yield better capturing results. AMD's superior memory management and bus contention handling mechanism was identified to be the most reasonable explanation. Since then, Intel has introduced Quick Path Interconnect [9] in its recent processor families, which has improved the performance of the Intel platform; however, we are not able to compare new AMD and Intel platforms at this time due to lack of hardware. In any case, users of packet capturing solutions should carefully choose the CPU platform, and should conduct performance tests before buying a particular hardware platform.

Apart from the CPU, another important hardware aspect is the speed of the bus system and the used memory. Current PCI-E buses and current memory banks allow high-speed transfer of packets from the capturing network card into the memory and the analysing CPUs. These hardware advances thus have shifted the bottlenecks, which were previously located at the hardware layer, into the software stacks.

Software stack:

There are several software subsystems involved in packet capture, as shown in Figure 1. Passing data between and within the involved subsystems can be a very important performance bottleneck that can impair capturing performance and thus lead to packet loss during capturing. We will discuss and analyse this topic in Section 4.3.

A packet's journey through the capturing system begins at the network interface card (NIC). Modern cards copy the packets into the operating systems kernel memory using Direct Memory Access (DMA), which reduces the work the driver and thus the CPU has to perform in order to transfer the data into memory. The driver is responsible for allocating and assigning memory pages to the card that can be used for DMA transfer. After the card has copied the captured packets into memory, the driver has to be informed about the new packets through an hardware interrupt. Raising an interrupt for each incoming packet will result in packet loss, as the system gets busy handling the interrupts (also known as an *interrupt storm*). This well-known issue has lead to the development of techniques like interrupt moderation or device polling, which have been proposed several years ago [7, 10, 11]. However, even today hardware interrupts can be a problem because some drivers are not able to use the hardware features or do not use polling—actually,

when we used the `igb` driver in FreeBSD 8.0, which was released in late 2009, we experienced bad performance due to interrupt storms. Hence, bad capturing performance can be explained by bad drivers; therefore, users should check the number of generated interrupts if high packet loss rates are observed.[1]

The driver's hardware interrupt handler is called immediately upon the reception of an interrupt, which interrupts the normal operation of the system. An interrupt handler is supposed to fulfill its tasks as fast as possible. It therefore usually doesn't pass on the captured packets to the operating systems capturing stack by himself, because this operation would take to long. Instead, the packet handling is deferred by the interrupt handler. In order to do this, a kernel thread is scheduled to perform the packet handling in a later point in time. The system scheduler chooses a kernel thread to perform the further processing of the captured packets according to the system scheduling rules. Packet processing is deferred until there is a free thread that can continue the packet handling.

As soon as the chosen kernel thread is running, it passes the received packets into the network stack of the operating system. From there on, packets need to be passed to the monitoring application that wants to perform some kind of analysis. The standard Linux capturing path leads to a subsystem called PF_PACKET; the corresponding system in FreeBSD is called BPF (Berkeley Packet Filter). Improvements for both subsystems have been proposed.

Software improvements:

The most prominent replacement for PF_PACKET on Linux is called PF_RING and was introduced in 2004 by Luca Deri [8]. Deri found that the standard Linux networking stack at that time introduced some bottlenecks, which lead to packet loss during packet capture. His capturing infrastructure was developed to remove these bottlenecks. He showed to achieve a higher capturing rate with PF_RING when small packets are to be captured. PF_RING ships with several modified drivers. These are modified to directly copy packets into PF_RING and therefore completely circumvent the standard Linux networking stack. This modification further boosts the performance for network cards with a modified driver.

Figure 2 shows the difference between the two capturing systems. One important feature of PF_RING is the way it exchanges packets between user space and kernel: Monitoring applications usually access a library like `libpcap` [12] to retrieve captured packets from the kernel. Libpcap is an abstraction from the operating systems' capturing mechanisms and allows to run a capturing application on several operating systems without porting it to the special capturing architecture. Back in 2004, the then current libpcap version 0.9.8 used a copy operation to pass packets from the kernel to the user space on Linux. An unofficial patch against that libpcap version from Phil Woods existed, which replaced the copy operation by a shared memory area that was used to exchange packets between kernel and application [13]. This modification will be called MMAP throughout the rest of the paper. PF_RING uses a similar structure to exchange packets by default. Libpcap version 1.0.0, which was re-

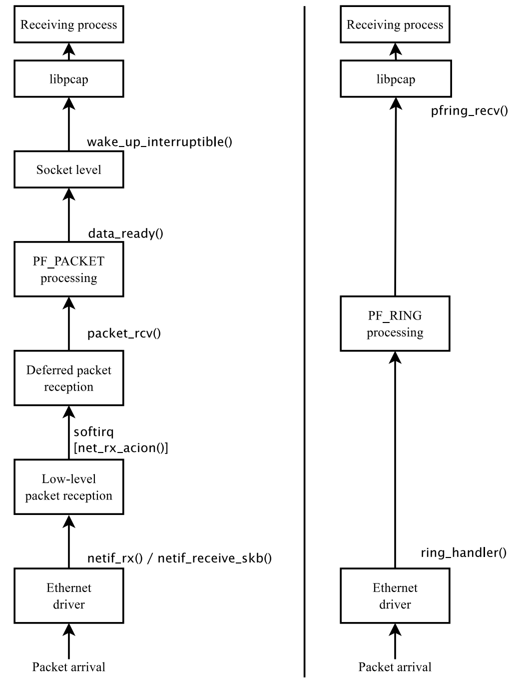

Figure 2: PF_RING and PF_PACKET under Linux

leased in late 2008, is the first version that ships built-in shared memory (SHM) exchange support; hence the patch from Phil Woods is not longer necessary. We will analyse the performance of these different solutions in Section 4.2.

All capturing mechanisms on Linux have something in common: They handle individual packets, meaning that each operation between user space and kernel is performed on a per-packet basis. FreeBSD packet handling differs in this point by exchanging buffers containing potentially several packets, as shown in Figure 3.

Both BPF as well as its improvement Zero-Copy BPF (ZCBPF) use buffers that contain multiple packets for storing and exchanging packets between kernel and monitoring application. BPF and ZCBPF use two buffers: The first, called HOLD buffer, is used by the application to read packets, usually via libpcap. The other buffer, the STORE buffer, is used by the kernel to store new incoming packets. If the application (i.e., via libpcap) has emptied the HOLD buffer, the buffers are switched, and the STORE buffer is copied into user space. BPF uses a copy operation to switch the buffers whereas ZCBPF has both buffers memory-mapped between the application and the kernel. Zero-Copy BPF is expected to perform better than BPF as it removes the copy operation between kernel and application. However, as there are fewer copy operations in FreeBSD than in non-shared-memory packet exchange on Linux, the benefits between ZCBPF and normal BPF in FreeBSD are expected to be smaller than in Linux with shared memory support.

TNAPI is very recent development by Luca Deri [5]. It improves standard Linux network card drivers and can be used in conjunction with PF_RING. Usually, Linux drivers assign new memory pages to network cards for DMA after the card copied new packets to old memory pages. The driver allo-

[1]FreeBSD will report interrupt storms via kernel messages. Linux exposes the number of interrupts via the proc file system in */proc/interrupts*.

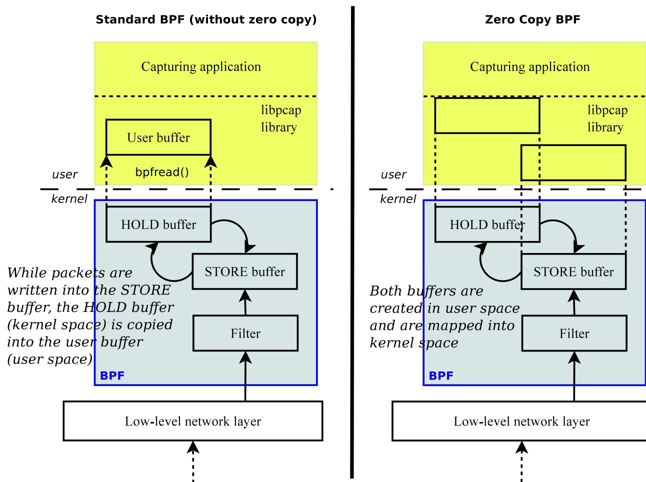

Figure 3: Berkeley Packet Filter and Zero Copy Berkeley Packet Filter

cates new pages and assign them to the card for DMA. Deri changed this behaviour in two ways: Non-TNAPI drivers receive an interrupt for received packets and schedule a kernel thread to perform the packet processing. The processing is performed by a kernel thread that is not busy doing other work and is chosen according to the scheduling rules of the operating system. Deri's driver creates a separate kernel thread that is only used to perform this processing. Hence, there is always a free kernel thread that can continue, and packet processing can almost immediately start.

The second major change is the memory handling within the driver: As soon as the thread is notified about a new packet arrival, the new packet is copied from the DMA memory area into the PF_RING ring buffer. Usually, network drivers would allocate new memory for DMA transfer for new packets. Deri's drivers tell the card to reuse the old memory page, thus eliminating the necessity of allocating and assigning new memory.

Furthermore, his drivers may take advantage of multiple RX queues and multi-core systems. This is done by creating a kernel thread for each RX queue of the card. Each kernel thread is bound to a specific CPU core in the case of multi-core systems. The RX queues are made accessible to the user space, so that users can run monitoring applications that read from the RX queues in parallel.

A rather non-standard solution for wire-speed packet capture is *ncap*. Instead of using the operating system's standard packet processing software, it uses special drivers and a special capturing library. The library allows an application to read packets from the network card directly [3].

2.2 Previous Comparisons

Previous work compared the different approaches to each other. We will now summarise the previous findings and determine which experiments we need to repeat with our hardware platforms and new software versions. Results from related work can also give hints on further experiments we need to conduct in order to achieve a better understanding of the involved processes.

Capturing traffic in 1 GE networks is seen as something that today's off-the-shelf hardware is able to do, whereas it remains a very challenging task in 10 GE networks [4]. Apart from the ten-fold increased throughput, the difficulties also lie in the ten-fold increased packet rate, as the number of packets to be captured per time unit is a factor that is even more important than the overall bandwidth. As an example, capturing a 1 GE stream that consists entirely of large packets, e.g., 1500 bytes, is easy; whereas capturing a 1 GE stream consisting entirely of small packets, e.g., 64 bytes, is a very difficult task [8, 4]. This is due to the fact that each packet, regardless of its size, introduces a significant handling overhead.

Driver issues that arise with a high number of packets have been studied very well [7, 10, 11]. Problems concerning interrupt storms are well understood and most network cards and drivers support mechanisms to avoid them. Such mechanisms include polling or interrupt moderation.

In 2007, Schneider et al. compared FreeBSD and Linux on Intel and AMD platforms [4]. They determined that device polling on Linux reduces the CPU cycles within the kernel and therefore helps to improve capturing performance. On FreeBSD however, device polling actually reduced the performance of the capturing and furthermore reduced the stability of the system. Hence, they recommend using the interrupt moderation facilities of the cards instead of polling on FreeBSD. In their comparison, FreeBSD using BPF and no device polling had a better capturing performance than Linux with PF_PACKET and device polling. This trend is enforced if multiple capturing processes are simultaneously deployed; in this case, the performance of Linux drops dramatically due to the additional load. Schneider et al. find that capturing 1 GE in their setup is possible with the standard facilities because they capture traffic that contains packets originated from a realistic size distribution. However, they do not capture the maximum possible packet rate, which is about 1.488 million packets per second with 64 bytes packets [5]. Another important aspect about the workload of Schneider et al. is that during each measurement, they send only one million packets (repeating each measurement for 7 times). This is a very low number of packets, considering that using 64 byte packets, it is possible to send 1.488 million packets within one second. Some of the effects we could observe are only visible if more packets are captured. Based on their measurement results, they recommend a huge buffer size in the kernel buffers, e.g., for the HOLD and STORE buffers in FreeBSD, to achieve good capturing performance. We can see a clear correlation between their recommendation and the number of packets they send per measurement and will come back to this in Section 4.2.

Deri validated his PF_RING capturing system against the standard Linux capturing PF_PACKET in 2004 [8]. He finds PF_RING to really improve the capturing of small (64 bytes) and medium (512 bytes) packets compared to the capturing with PF_PACKET and libpcap-mmap. His findings were reproduced by Cascallana and Lizarrondo in 2006 [6] who also found significant performance improvements with PF_RING. In contrast to Schneider et al., neither of the PF_RING comparisons consider more than one capturing process.

Using TNAPI and the PF_RING extensions, Deri claims to be able to capture 1 GE packet streams with small packet sizes at wire-speed (1.488 million packets) [5].

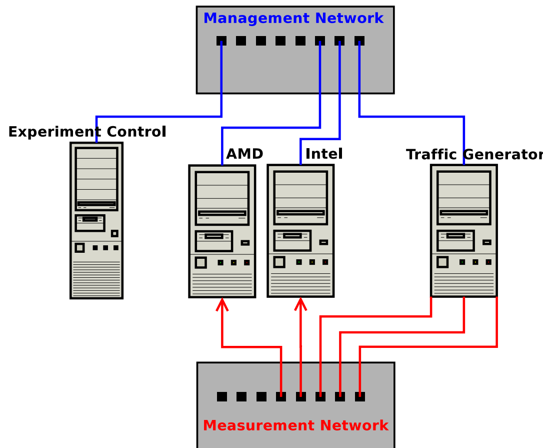

Figure 4: Evaluation Setup

3. TEST SETUP

In this section, we describe the hardware and software setup that we used for our evaluation. Our test setup (see Figure 4) consists of three PCs, one for traffic generation, the other two for capturing.

The traffic generator is equipped with several network interface cards (NICs) and uses the Linux Kernel Packet Generator [14] to generate a uniform 1 GE packet stream. It was necessary to generate the traffic using several NICs simultaneously because the deployed cards were not able to send out packets at maximum speed when generating small packets. Even this setup was only able to generate 1.27 million packets per seconds (pps) with the smallest packet size. However, this packet rate was sufficient to show the bottlenecks of all analysed systems. We had to deploy a second generator for our experiments in Section 4.4, where we actually needed wire-speed packet generation (1.488 million pps).

In contrast to [4], we where did not produce a packet stream with a packet size distribution that is common in real networks. Our goal is explicitly to study the behaviour of the capturing software stacks at high packet rates. As we do not have 10 GE hardware for our tests available, we have to create 64 bytes packets in order to achieve a high packet rate for our 1 GE setup.

Each of our test runs is configured to produce a packet stream with a fixed number of packets per seconds and runs over a time period of 100 seconds and repeated for five times. As our traffic generator is software based and has to handle several NICs, the number of packets per second is not completely stable and can vary to some extend. Hence, we measure the variations in our measurements and plot them where appropriate.

Capturing is performed with two different machines with different hardware in order to check whether we can reproduce any special events we may observe with different processor, bus systems and network card, too. The first capturing PC is operated by two Intel Xeon CPUs with 2.8 GHz each. It has several network cards including an Intel Pro/1000 (82540EM), an Intel Pro /1000 (82546EB) and a Broadcom BCM5701 all connected via a PCI-X bus. In our experiments, we only use one NIC at a time.

The second capturing PC has an AMD Athlon 64 X2 5200+ CPU and is also equipped with several network cards, including an Intel Pro/1000 (82541PI) and a nVidia Corporation CK804 onboard controller. Both cards are connected via a slow PCI bus to the system. Furthermore, there are two PCI-E based network cards connected to the system. One is an Intel Pro/1000 PT (82572EI), the other is a Dual Port Intel Pro/1000 ET(82576). It should be noted that the AMD platform is significantly faster than the Xeon platform.

Using different network cards and different architectures, we can check if an observed phenomenon emerges due to a general software problem or if it is caused by a specific hardware or driver. Both machines are built from a few years old and therefore cheap hardware, thus our machines are not high end systems. We decided not to use more modern hardware for our testing because of the following reasons:

- We did not have 10 GE hardware.
- We want to identify problems in the software stack that appear when the hardware is fully utilised. This is quite difficult to achieve with modern hardware on a 1 GE stream.
- Software problems that exist on old hardware which monitors 1 GE packet streams still exist on newer hardware that monitors a 10 GE packet stream.

Both machines are installed with Ubuntu Jaunty Jackalope (9.04) with a vanilla Linux kernel version 2.6.32. Additionally, they have an installation of FreeBSD 8.0-RELEASE for comparison with Linux.

We perform tests with varying load at the capturing machines' application layer in order to simulate the CPU load of different capturing applications during monitoring. Two tools are used for our experiments:

First, we use tcpdump 4.0.0 [15] for capturing packets and writing them to `/dev/null`. This scenario emulates a simple one-threaded capturing process with very simple computations, which thus poses almost no load on the application layer. Similar load can be expected on the capturing thread of multi-threaded applications that have a separate thread that performs capturing only. Examples for such multi-threaded applications are the Time Machine [16, 17] or the network monitor VERMONT [18].

The second application was developed by us and is called *packzip*. It poses variable load onto the thread that performs the capturing. Every captured packet is copied once within the application and is then passed to `libz` [19] for compression. The compression mode can be configured by the user from 0 (no compression) to 9 (highest compression level). Increasing the compression level increases CPU usage and thus can be used to emulate an increased CPU load for the application processing the packets. Such packet handling has been performed before in [2] and [4]. We used this tool in order to make our results comparable to the results presented in this related work.

4. EVALUATION

This section presents our analysis results of various packet capture setups involving Linux and FreeBSD, including a performance analysis of our own proposed improvements

to the Linux capturing stack. As multi-core and multi-processor architectures are common trends, we focus in particular on this kind of architecture. On these hardware platforms, scheduling issues arise when multiple processes involved in packed capturing need to be distributed over several processors or cores. We discuss this topic in Section 4.1. Afterwards, we focus on capturing with low application load in Section 4.2 and see if we can reproduce the effects that have been observed in the related work. In Section 4.3, we proceed to capturing with higher application load. We identify bottlenecks and provide solutions that lead to improvements to the capturing process. Finally, in Section 4.4 we present issues and improvements at the driver level and discuss how driver improvements can influence and improve the capturing performance.

4.1 Scheduling and packet capturing performance

On multi-core systems, scheduling is an important factor for packet capturing: If several threads, such as kernel threads of the operating system and a multi-threaded capturing application in user space are involved, distributing them among the available cores in a clever way is crucial for the performance.

Obviously, if two processes are scheduled to run on a single core, they have to share the available CPU time. This can lead to shortage of CPU time in one or both of the processes, and results in packet loss if one of them cannot cope with the network speed. Additionally, the processes then do not run simultaneously but alternately. As Schneider et al. [4] already found in their analysis, packet loss occurs if the CPU processing limit is reached. If the kernel capturing and user space analysis are performed on the same CPU, the following effect can be observed: The kernel thread that handles the network card dominates the user space application because it has a higher priority. The application has therefore only little time to process the packets; this leads to the kernel buffers filling up. If the buffers are filled, the kernel thread will take captured packets from the card and will throw them away because there is no more space to store them. Hence, the CPU gets busy capturing packets that will be immediately thrown away instead of being busy processing the already captured packets, which would empty the buffers.

Conversely, if the processes are scheduled to run on two cores, they have more available CPU power to each one of them and furthermore can truly run in parallel, instead of interleaving the CPU. However, sharing data between two cores or CPUs requires memory synchronisation between both of the threads, which can lead to severe performance penalties.

Scheduling can be done in two ways: Processes can either be scheduled dynamically by the operating system's scheduler, or they can be statically pinned to a specific core by the user. Manual pinning involves two necessary operations, as described in [5, 20]:

- Interrupt affinity of the network card interrupts have to be bound to one core.
- The application process must be bound to another core.

We check the influence of automatic vs. manually pinned scheduling in nearly all our experiments.

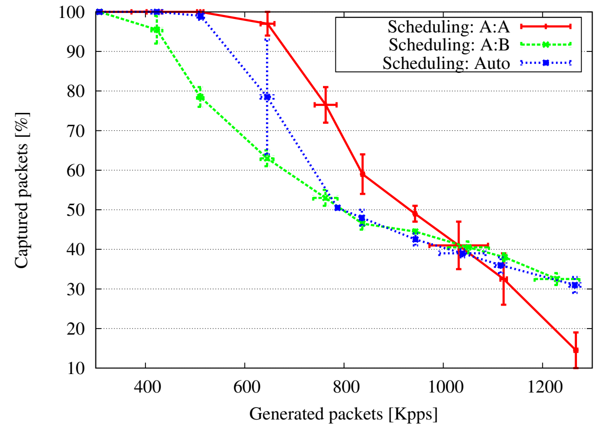

Figure 5: Scheduling effects

Figure 5 presents a measurement with 64 byte packets with varying numbers of packets per seconds and low application load.

Experiments were run where the kernel and user space application are processed on the same core (A:A), are pinned to run on different cores (A:B), or are scheduled automatically by the operating system's scheduler. Figure 5 shows that scheduling both processes on a single CPU results in more captured packets compared to running both processes on different cores, when packet rates are low. This can be explained by the small application and kernel load at these packet rates. Here, the penalties of cache invalidation and synchronisation are worse than the penalties of the involved threads being run alternately instead of parallel. With increasing packet rate, the capturing performance in case A:A drops significantly below that of A:B.

Another interesting observation can be made if automatic scheduling is used. One would expect the scheduler to place a process on the core where it performs best, depending on system load and other factors. However, the scheduler is not informed about the load on the application and is therefore not able to make the right decision. As can be seen on the error bars in Figure 5, the decision is not consistent over the repetitions of our experiments in all cases, as the scheduler tries to move the processes to different cores and sometimes sticks with the wrong decision whereas sometimes it makes a good decision.

Real capturing scenarios will almost always have a higher application load than the ones we have shown so far. In our experiments with higher application load which we will show in Section 4.3, we can almost always see that running the processes in A:B configuration results in better capturing performance. Static pinning always outperformed automatic scheduling as the schedulers on Linux and FreeBSD almost make wrong decisions very frequently.

4.2 Comparing packet capture setups under low application load

Applications that capture network traffic usually build on `libpcap` [12] as presented in Section 2.1. FreeBSD ships libpcap version 1.0.0 in its base system, and most Linux

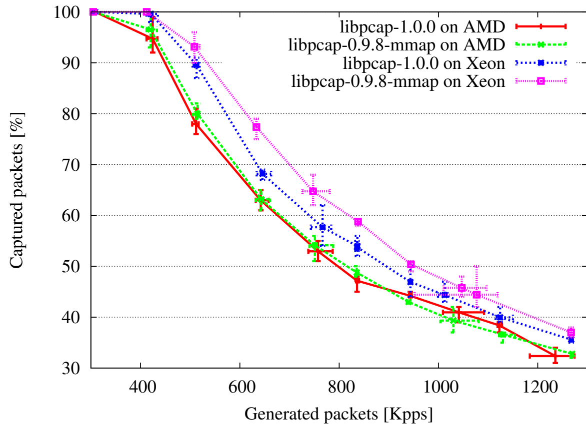

Figure 6: Comparison of different libpcap versions on Linux

distributions use the same version today[2]. Not long ago, libpcap version 0.9.8 was the commonly used version on Linux based systems and FreeBSD. As already explained in Section 2, libpcap-0.9.8 or libpcap-0.9.8-mmap were used in most earlier evaluations.

In this paper, we want to use the standard libpcap-1.0.0, which ships with a shared-memory support. As the shared-memory extensions in libpcap 0.9.8 and 1.0.0 where developed by different people, we first want to compare both versions. The results of this comparison are plotted in Figure 6 for the AMD and Xeon platforms. We can see that libpcap-0.9.8 performs slightly better on Xeon than libpcap 1.0.0 while 1.0.0 performs better on AMD. However, both differences are rather small, so that we decided to use the now standard 1.0.0 version for our experiments.

Having a closer look at the figure, one can see that the Xeon platform performs better than the AMD platform. This is very surprising as the AMD system's hardware performance is otherwise much faster than the aged Xeon platform. Things get even more weird if we include libpcap-0.9.8 without MMAP into our comparison (not shown in our plot): As the copy operation is way more expensive than the MMAP, one would expect MMAP to perform better than the copy operation. This assumption is true on the Xeon platform. On the AMD platform however, we can observe that libpcap-0.9.8 *without* MMAP performs better than libpcap-0.9.8-mmap or libpcap-1.0.0. This points to some unknown performance problems which prevents the AMD system from showing its superior hardware performance.

The next comparison is between standard Linux capturing with PF_PACKET and capturing on Linux using the PF_RING extension from Deri [8]. We therefore use Deri's patches to libpcap-1.0.0 which enables libpcap to read packets from PF_RING. Two important PF_RING parameters can be configured. The first one is the size of the ring buffer, which can be configured in number of packets that can be stored in the ring. Our experiments with different ring sizes reveal that in our setup, the size of the memory mapped area is not of much influence. We conducted similar experiments with the sizes of Berkeley Packet Filter on FreeBSD and other buffer sizes on Linux. All these experiments showed that increasing the buffer size beyond a certain limit does not boost capturing performance measurably. Instead, we found evidence that too large buffers have a negative impact on capturing performance. These findings are contrary to the findings from Schneider et al. [4], who found large buffers to increase the capturing performance.

The biggest factor regarding the performance is our capturing application—since our hardware, at least the AMD platform, is able to transfer all packets from the network card into memory. If the software is able to consume and process incoming packets faster than wire-speed, the in-kernel buffers will never fill up and there will be no packet loss. However, if the application is not able to process the packets as fast as they come in, increasing the buffer will not help much—rather, it will only reduce the packet loss by the number of elements that can be stored in the buffer, until the buffer is filled.

Schneider et al. sent only 1,000,000 packets per measurement run, whereas we produce packets with 1 GE speed (i.e., more than 100 Megabytes per second), which usually amounts to much more than 1,000,000 packets *per second*, over a time interval of 100 seconds. Increasing kernel buffer sizes to 20 megabytes, as recommended by Schneider et al., allows them to buffer a great share of their total number of packets, but does not help much on the long term. If we increase the buffers to the recommended size, we cannot see any significant improvements in our experiments.

Buffer size can be crucial, though: This is the case when the monitoring is not able to process packets at wire-speed, e.g., it can consume up to N packets per second (pps), and bursty Internet traffic is captured. If this traffic transports less or equal than N pps on average but has bursts with a higher pps rate, then having a sufficient dimensioned buffer to store the burst packets is obviously very important.

The second important parameter is a configuration option called `transparent_mode`. It configures how PF_RING handles packets:

- Transparent mode 0: Captured packets are inserted into the ring via the standard Linux socket API.
- Transparent mode 1: The network card driver inserts the packets directly into the ring (which requires an adopted network driver). Packets are also inserted into the standard Linux network stack.
- Transparent mode 2: Same as mode 1, but received packets are not copied into the standard Linux network stack (for capturing with PF_RING only).

It is obvious that *transparent_mode* 2 performs best as it is optimised for capturing. We conducted some comparisons using different packet sizes and packets rates and indeed found PF_RING to perform best in this mode.

We expected PF_RING to outperform the standard Linux capturing due to the evaluations performed in [8] and [6] when using small packet sizes. This performance benefit should be seen with small packets (64 bytes) and a high number of packets per second, and disappear with bigger packet sizes. According to Deri's evaluation, capturing small packet

[2]An exception is the current stable version of Debian 5.0 "Lenny", which still contains libpcap version 0.9.8-5 at the time this paper was written.

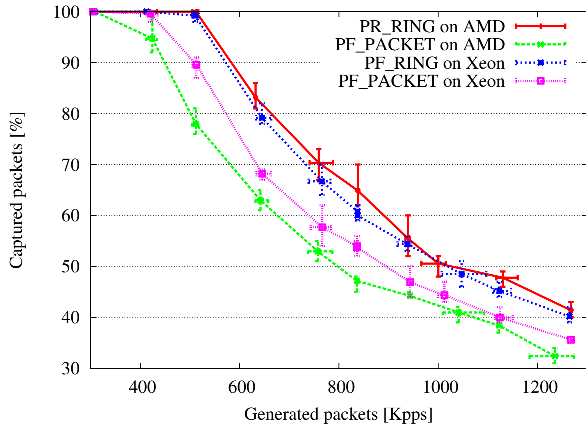

Figure 7: PF_PACKET vs. PF_RING

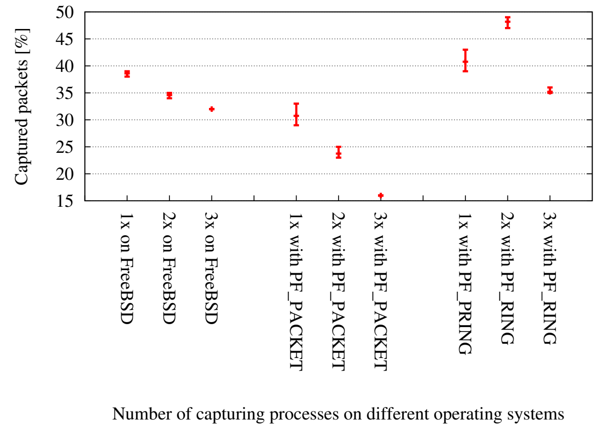

Figure 9: Capturing with multiple processes

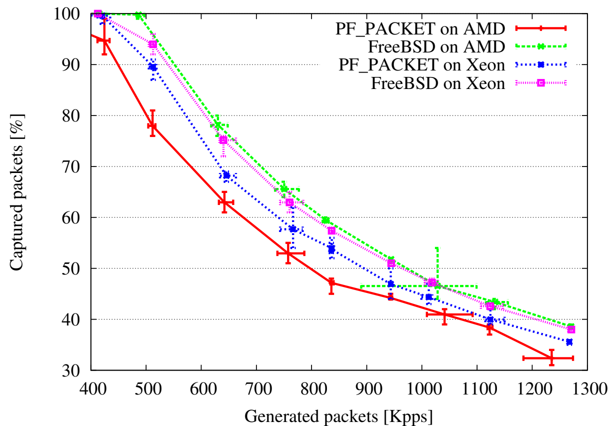

Figure 8: Linux vs. FreeBSD

streams using PF_PACKET in Linux 2.6.1 with libpcap-mmap only captured 14.9% of all packets while PF_RING within the same kernel version was able to capture 75.7% of all packets. Our comparison confirms that PF_RING indeed performs better than PF_PACKET, as can be seen in Figure 7. We can also see that PF_RING is better than PF_PACKET on both systems. Furthermore, PF_RING on the faster AMD hardware performs better than PF_RING on Xeon. However, the performance difference is small if one considers the significant differences in hardware, which again points to some performance problems which we pointed out before. One more observation concerns the difference between PF_PACKET and PF_RING within the same platform. Although there is some difference, it is not as pronounced as in previous comparisons. We explain this by the improvements that have been made in the capturing code of PF_PACKET and within libpcap since Deri's and Cascallana's evaluations in 2004 and 2006.

We now compare the measurement results of FreeBSD against the Linux measurements. FreeBSD has two capturing mechanisms: Berkeley Packet Filter (BPF) and Zero Copy Berkeley Packet Filter (ZCBPF) as described in Section 2.1. At first, we compared both against each other, but we could not find any significant differences in any of our tests. Schneider et al. found FreeBSD to perform amazingly good even though FreeBSD employed this packet copy operation. This might indicate that the copy operation is indeed not a significant factor that influences the performance of the FreeBSD capturing system. We are uncertain about the true reason for Zero Copy BPF not performing better than BPF; therefore, we do not include ZCBPF into our further comparisons.

Our comparison with the standard BPF is shown in Figure 8 and presents the differences between PF_PACKET and FreeBSD. We can see some differences between capturing with PF_PACKET on Linux and capturing with BPF on FreeBSD. FreeBSD performs slightly better on both platforms, which confirms the findings of Schneider et al. [2, 4].

Differences increase if more than one capturing process in running on the systems, as shown in Figure 9. This figure shows the capturing performance of FreeBSD, Linux with PF_PACKET and Linux with PF_RING capturing a packet stream of 1270 kpps on the AMD system with one, two and three capturing processes running simultaneously.

Capturing performance on both systems decreases due to the higher CPU load due to the multiple capturing processes. This is an obvious effect and has also been observed in [4]. Linux suffers more from the additional capturing processes compared to FreeBSD, which has also been observed in related work. Amazingly, Linux with PF_RING does not suffer much from these performance problems and is better than FreeBSD and Linux with PF_PACKET. A strange effect can be seen if two capturing processes are run with PF_PACKET: Although system load increases due to the second capturing process, the overall capturing performances increases. This effect is only visible on the AMD system and not on the Xeon platform and also points to the same strange effect we have seen before and which we will explain in Section 4.3.

If we compare our analysis results of the different capturing solutions on FreeBSD and Linux with the results of earlier evaluations, we can see that our results confirm several prior findings: We can reproduce the results of Deri [8] and Cascallana [6] who found PF_RING to perform better than PF_PACKET on Linux. However, we see that the difference between both capturing solutions is not as strong

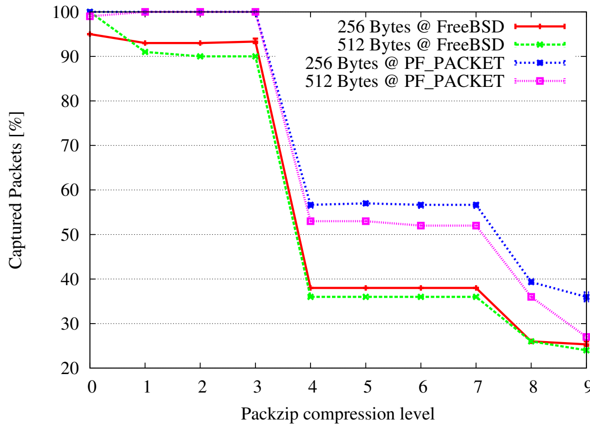

Figure 10: Packzip on 256 and 512 byte packets

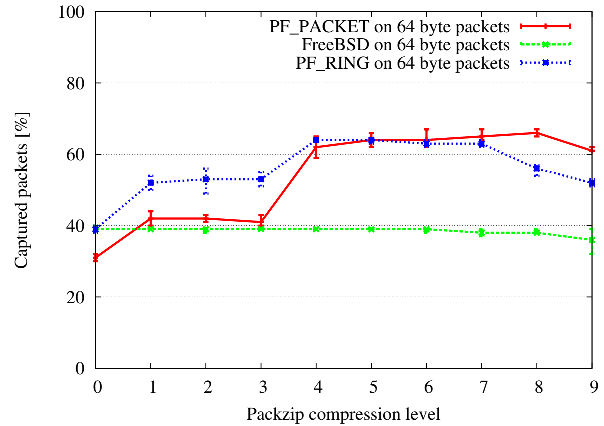

Figure 11: Packzip on 64 byte packets

as it used to be in 2004 and 2006 due to general improvements in the standard Linux software stack. Furthermore, we can confirm the findings of Schneider et al. [4] who found that FreeBSD outperforms Linux (with PF_PACKET), especially when more than one capturing process is involved. In contrast, our own analyses reveal that PF_RING on Linux outperforms both PF_PACKET and FreeBSD. PF_RING is even better than FreeBSD when more than one capturing process is deployed, which was the strength of FreeBSD in Schneiders' analysis [4]. We are now comparing the capturing solutions with increased application load and check whether our findings are still valid in such setups.

4.3 Comparing packet capture setups under high application load

So far, we analysed the capturing performance with very low load on the application by writing captured packets to `/dev/null`. We now increase the application load by performing more computational work for each packet by using the tool *packzip*, which compresses the captured packets using `libz` [19]. The application load can be configured by changing the compression level libz uses. Higher compression levels result in higher application load; however, the load does not increase linearly. This can be seen at the packet drop rates in Figure 10.

The figure presents the results of the capturing process on various systems when capturing a stream consisting of 512 and 256 bytes sized packets at maximum wire-speed at the AMD platform. It can clearly be seen that a higher application load leads to higher packet loss. This happens on both operating system with every capturing solution presented in Section 2.1, and is expected due to our findings and the findings in related work. Packet loss kicks in as soon as the available CPU processing power is not sufficient to process all packets. We note that FreeBSD performs worse on our AMD platform compared to Linux with PF_PACKET with higher application load. This observation can be made throughout all our measurements in this section.

However, if we look at a packet stream that consists of 64 byte packets, we can see an amazing effect, shown in Figure 11. At first, we see that the overall capturing performance is worse when no application load (compression level 0) compared to the capturing results with 256 and 512 byte packets. This was expected because capturing a large number of (small) packets is more difficult than small number of (big) packets. However, if application load increases, the capturing performance increases as well. This effect is quite paradox and points to the same strange effect as seen in the experiments before. It can be observed with PF_PACKET and with PF_RING but is limited to Linux. FreeBSD is not affected by this weird effect; instead, capturing performance is nearly constant with increasing application load but almost always below Linux' performance.

In order to better understand what is causing this bewildering phenomenon, we have to take a closer look at the capturing process in Linux. An important task within the capturing chain is the passing of a captured packet from the kernel to the application. This is done via a shared memory area between the kernel and application within libpcap. The shared memory area is associated with a socket, in order to allow signaling between the kernel and the application, if this is desired. We will call this memory area SHM in the remainder of this section.

Packets are sent from kernel to user space using the following algorithm:

- The user application is ready to fetch a new packet.
- It checks SHM for new packets. If a new packet is found, it is processed.
- If no new packet is found, the system call `poll()` is issued in order to wait for new packets.
- The kernel receives a packet and copies it into SHM.
- The kernel "informs" the socket about the available packet; subsequently, `poll()` returns, and the user application will process the packet.

This algorithm is problematic because it involves many systems calls if the application consumes packets very quickly, which has already been found to be problematic in Deri's prior work [8]. A system call is a quite expensive operation as it results in a context switch, cache invalidation, new scheduling of the process, etc. When we increase the compression level, the time spent consuming the packets in-

creases as well, thus less system calls are performed[3]. Obviously, reducing the number of system calls is beneficial to the performance of the packet capture system.

There are several ways to achieve such a reduction. The most obvious solution to this problem is to skip the call to `poll()` and to perform an active wait instead. However, this solution can pose problems to the system: If capturing and analysis are scheduled to run on the same processor (which is not recommended, as we pointed before), polling in a user space process eats valuable CPU time, which the capturing thread within the kernel would need. If capturing and analysis run on different cores, there still are penalties: The kernel and the user space application are trying to access the same memory page, which the kernel wants to write and the user space application wants to read. Hence, both cores or CPUs need to synchronise their memory access, continuously leading to cache invalidations and therefore to bad performance. We patched libpcap to perform the active wait and found some, but only little, improvement due to the reasons discussed above.

Hence, we searched for ways to reduce the number of calls to `poll()` that do not increase the load on the CPU. The first one was already proposed by Deri [8]. The basic idea is to perform a sleep operation for several nano seconds if the SHM is found to be empty. Although the sleep operations still implies a system call, it is way better than multiple calls to `poll()`, as the kernel capturing gets some time to copy several packets into the SHM. Deri proposed an adaptive sleep interval that is changed according to the incoming packet rate. We implemented this modification into the libpcap which used PF_PACKET. We found that it is not an easy task to choose a proper sleep interval, because sleeping too long will result in filled buffers, whereas a sleep interval that is too short will result in too many system calls and therefore does not solve the problem. Unfortunately, the optimal sleep interval depends on the hardware, the performed analysis and, even worse, on the observed traffic. Hence, a good value has to be found through the end-user by experiments, which requires quite some effort. Deri's user space code that uses PF_RING does not implement his proposed adaptive sleep, probably due to the same reasons. Instead, poll() avoidance is achieved by calls to sched_yield(), which interrupts the application and allows the scheduler to schedule another process. A call to pool() is only performed if several sched_yield() was called for several times and still no packets arrived. Figure 11 shows that the algorithm used by PF_RING yield better performance compared to the simple call to poll() with PF_PACKET. However, we can also see negative effects of calling sched_yield() often.

We therefore propose a new third solution that works without the necessity to estimate a timeout and works better than calling sched_yield(): We propose to change the signalling of new incoming packets within `poll()`. As of now, PF_PACKET signals the arrival of every packet into the user space. We recommend to only signal packet arrival if one of the two following conditions are true:

- N packets are ready for consuming.
- A timeout of m microseconds has elapsed.

Using these conditions, the number of system calls to `poll()` is reduced dramatically. The timeout is only necessary if less than N packets arrive within m, e.g. if the incoming packet rate is low. In contrast to the sleep timeout discussed before, choosing m properly is not necessary to achieve good capturing performance.

[3]We confirmed this by measuring the number of calls to `poll()`.

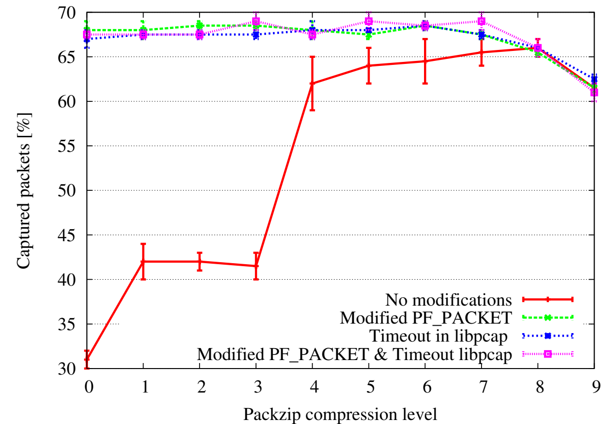

Figure 12: Capturing 64 byte packets with and without modifications

We implemented all the solutions into PF_PACKET and libpcap and compared their implications on the performance. For the sleep solutions, we determined a good timeout value by manual experiments. The results are summarised in Figure 12. As we can see, the reduced number of system calls yields a large performance boost. We can see that both timeout and our proposed modification to PF_PACKET yield about the same performance boost. Combining both is possible, but does not result in any significant further improvements. We did not have the time to include and evaluate the same measurements with PF_RING, but we are confident that PF_RING will also benefit from our proposed modification.

4.4 Driver Improvements

In the experiments presented up to now, we were not able to process small packets at wire-speed, even with our previous improvements. We now move further down the capturing software stack and test driver improvements.

Deri proposed to use modified drivers in order to improve the capturing performance [5]. His driver modifications focus on changing two things previously described:

- Create a dedicated thread for the packet consumption in the driver (respectively for every RX queue of the card).
- Reuse DMA memory pages instead of allocating new pages for the card.

His driver modifications hence help to use the power of multi-core CPUs by spawning a kernel thread that handles the card. This implies that no other thread is scheduled to perform the driver tasks and ensures that there is always a free thread to handle the packets, which is good for the overall performance. Additionally, if interrupts are bound to a given core, the driver thread will run on the same core. It is unclear to us which of the modifications has

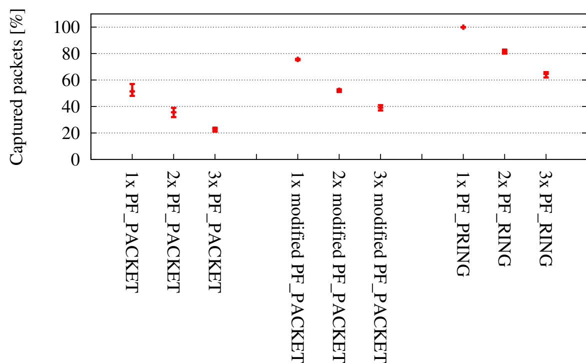

Figure 13: TNAPI with different capturing solutions on Linux

a bigger influence on the performance. If TNAPI is used with PF_PACKET instead of PF_RING, an additional copy operation is performed to copy the packet from the DMA memory area into a new memory area. The DMA area is reused afterwards.

We tested these improvements with different solutions on Linux but could not consider FreeBSD as there where no modified drivers at the time this paper was written. Our comparison is plotted in Figure 13. In contrast to our previous experiment, we deployed one more traffic generator, which allowed us to generate traffic at real wire-speed (i.e., 1.488 million packets). The plot shows results on the AMD system and compares PF_PACKET against our modified PF_PACKET and PF_RING. As can be seen, the TNAPI-aware drivers result in improvements compared to normal drivers. PF_PACKET capturing also benefits from TNAPI, but the improvements are not very significant.

Using TNAPI with our modified PF_PACKET results in good capturing results which are better than the results with standard drivers and also better than the results with standard FreeBSD. The best performance, however, can be found when TNAPI is combined with PF_RING, resulting in a capturing process that is able to capture 64 byte packets at wire-speed.

5. RECOMMENDATIONS

This section summarises our findings and gives recommendations to users as well as system developers. Our recommendations for developers of monitoring applications, drivers and operating systems are listed in subsection 5.1. Users of capturing solutions can find advices on how to configure their systems in subsection 5.2.

5.1 For Developers

During our evaluation and comparison, we found some bottlenecks within the software which can be avoided with careful programming. Developers of monitoring applications, operating systems or drivers should consider these hints in order to increase the performance their systems provide.

Our first advice is targeted at developers of network card drivers. We were able to determine that having a separate kernel thread that is only responsible for the network card can really help to improve performance. This is especially useful if more than multi-core or multi-CPU systems are available. With current hardware platforms tending to be multi-core systems and the ongoing trend of increasing the number of cores in a system, this feature can be expected to become even more important in the future. In addition, reusing memory pages as DMA areas and working with statically allocated memory blocks should be preferred over using dynamically allocated pages. However, it is unclear to us which of this two recommendations results in the greatest performance boosts.

Signalling between different subsystems, especially if they are driven by another thread, should always be done for accumulated packets. This assumption is valid for all subsystems, ranging from the driver, over the general operating system stacks, up to the user space applications. We therefore recommend the integration of our modifications to PF_PACKET into Linux.

Other areas besides packet capturing may also benefit from our modification: A generic system call that allows to wait for one or more sockets until N elements can be read or a timeout is seen, whichever happens first, would be a generalised interface for our modifications to PF_PACKET. Such system calls could be of use in other applications that need to process data on-line as fast as the data arrives.

5.2 For Users

Configuring a capture system for optimal performance is still challenging task. It is important to choose proper hardware that is capable of capturing the amount of traffic in high-speed networks.

The software that drives the hardware is very important for capture as well, since it highly influences the performance of the capturing process. Our findings conclude that *all* parts of the software stack have great influence on the performance—and that, unfortunately, a user has to check *all* of them in order to debug performance problems.

Performance pitfalls can start at the network card drivers, if interrupt handling does not work as expected, which we found to be true with one of the drivers we used. Checking for an unexpectedly high number of interrupts should one of the first performance debugging steps. Here, enabling polling on the driver could help to solve the issue. However, we found that the POLLING option, a static compile time options for many drivers, did not improve the performance in our tests.

We also recommend to use PF_RING with TNAPI, as its performance is superior to the standard capturing stack of FreeBSD or Linux. If using TNAPI or PF_RING is not an option, e.g., because there is no TNAPI-aware driver for the desired network interface card, we recommend to use Linux with our modifications to PF_PACKET.

Regardless of the capturing solution used, pinning all the involved threads and processes to different cores is highly recommended in most of the application cases. Using the default scheduler is only recommended when low packet rates are to be captured with low load at the application layer. Kernel and driver threads should also be pinned to a given core if possible. This can be achieved by explicitly setting the interrupt affinity of a network cards' interrupt.

Finally, if it is not an option to apply one of our proposed techniques for reducing the number of system calls (cf. Sec-

tions 4.3 and 5.1), the user should check if performance improves if he puts a higher load on the application capturing process. As we have seen in 4.3, a higher application can reduce the number of calls to poll() and therefore improve the performance. This holds especially for applications that do not involve much CPU workload, e.g., writing a libpcap stream to disk.

6. CONCLUSION

This paper summarised and compared different capturing solutions on Linux and FreeBSD, including some improvements proposed by researchers. We compared the standard capturing solutions of Linux and FreeBSD to each other, leading to a reappraisal of past work from Schneider et al [4] with newer versions of the operating systems and capturing software. The evaluation revealed that FreeBSD still outperforms standard Linux PF_PACKET under low application load. FreeBSD is especially good when multiple capturing processes are run.

Our comparison between standard Linux capturing with PF_PACKET and PF_RING confirmed that performance with PF_RING is still better when small packets are to be captured. However, differences are not as big as they where in 2004 or 2006, when the last evaluations were published. Further analyses showed that PF_RING performs better than PF_PACKET if multiple capturing processes are run on the system, and that performance with PF_RING is even better than FreeBSD. During our work, we found a performance bottleneck within the standard Linux capturing facility PF_PACKET and proposed a fix for this problem. Our fix greatly improves the performance of PF_PACKET with small packets. Using our improvements, PF_PACKET performs nearly as good as PF_RING.

Finally, we evaluated Luca Deri's TNAPI driver extension for Linux and found increased performance with all Linux capturing solutions. Best performance can be achieved if TNAPI is combined with PF_RING.

Acknowledgements

We gratefully acknowledge support from the German Research Foundation (DFG) funding the LUPUS project in the scope of which this research work has been conducted. Additional work was funded through ResumeNet, an EU FP7 project within the FIRE initiative.

The authors would like to thank Luca Deri and Gerhard Münz for their valuable support throughout our research.

7. REFERENCES

[1] Endace Measurement Systems, http://www.endace.com/.

[2] F. Schneider and J. Wallerich, "Performance evaluation of packet capturing systems for high-speed networks," in *CoNEXT'05: Proceedings of the 2005 ACM conference on Emerging network experiment and technology*. New York, NY, USA: ACM Press, Oct. 2005, pp. 284–285.

[3] L. Deri, "ncap: Wire-speed packet capture and transmission," in *In Proceedings of the IEEE/IFIP Workshop on End-to-End Monitoring Techniques and Services (E2EMON)*, May 2005.

[4] F. Schneider, J. Wallerich, and A. Feldmann, "Packet capture in 10-gigabit ethernet environments using contemporary commodity hardware," in *In Proceedings of the 8th International Conference on Passive and Active Network Measurement*, Apr. 2007.

[5] L. Deri and F. Fusco, "Exploiting commodity multi-core systems for network traffic analysis," http://luca.ntop.org/MulticorePacketCapture.pdf, July 2009.

[6] G. A. Cascallana and E. M. Lizarrondo, "Collecting packet traces at high speed," in *Proc. of Workshop on Monitoring, Attack Detection and Mitigation (MonAM) 2006*, Tübingen, Germany, Sep. 2006.

[7] J. Mogul and K. K. Ramakrishnan, "Eliminating receive livelock in an interrupt-driven kernel," *ACM Transactions on Computer Systems*, vol. 15, no. 3, pp. 217–252, 1997.

[8] L. Deri, "Improving passive packet capture: Beyond device polling," in *In Proceedings of the Foruth International System Adminstration and Network Engineering Conference (SANE 2004)*, Sep. 2004.

[9] Intel Corporation, "An Introduction to the Intel QuickPath Interconnect," http://www.intel.com/technology/quickpath-/introduction.pdf, 2009.

[10] I. Kim, J. Moon, and H. Y. Yeom, "Timer-based interrupt mitigation for high performance packet processing," in *5th International Conference on High-Performance Computing in the Asia-Pacific Region*, 2001.

[11] Luigi Rizzo, "Device Polling Support for FreeBSD," in *BSDCon Europe Conference 2001*, 2001.

[12] V. Jacobson, C. Leres, and S. McCanne, "libpcap," http://www.tcpdump.org.

[13] Phil Woods, "libpcap MMAP mode on linux," http://public.lanl.gov/cpw/.

[14] R. Olsson, "pktgen the linux packet generator."

[15] tcpdump, http://www.tcpdump.org.

[16] S. Kornexl, V. Paxson, H. Dreger, A. Feldmann, and R. Sommer, "Building a Time Machine for Efficient Recording and Retrieval of High-Volume Network Traffic," in *Proc. of ACM SIGCOMM Internet Measurement Conference (IMC) 2005*, Berkeley, CA, USA, Oct. 2005.

[17] G. Maier, R. Sommer, H. Dreger, A. Feldmann, V. Paxson, and F. Schneider, "Enriching Network Security Analysis with Time Travel," in *Proc. of ACM SIGCOMM 2008 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication*, Seattle, WA, USA, Aug. 2008.

[18] R. T. Lampert, C. Sommer, G. Münz, and F. Dressler, "Vermont - A Versatile Monitoring Toolkit for IPFIX and PSAMP," in *Proceedings of Workshop on Monitoring, Attack Detection and Mitigation (MonAM) 2006*, Tuebingen, Germany, Sep. 2006.

[19] Homepage of the zlib project, http://www.zlib.net/.

[20] C. Satten, "Lossless gigabit remote packet capture with linux," http://staff.washington.edu/corey/gulp/, University of Washington Network Systems, Mar. 2008.