Profiling and Optimization of Software-Based Network-Analysis Applications

Loris Degioanni, Mario Baldi, Fulvio Risso, and Gianluca Varenni
Dipartimento di Automatica e Informatica
Politecnico di Torino
Corso Duca degli Abruzzi, 24 – 10129 Torino, Italy
{loris.degioanni, mario.baldi, fulvio.risso, gianluca.varenni}@polito.it

Abstract

A large set of tools for network monitoring and accounting, security, traffic analysis and prediction — more broadly, for network operation and management — require direct and efficient real-time access to data traveling on the network. Software tools are often preferred because of their low cost and high versatility. However, these tools are often considered to suffer from performance problems on high-speed networks. This paper demonstrates that, despite the common belief, the performance limits for software real-time network analysis tools are still far from being reached and it can even improved with limited hardware support. This work analyzes the performance of a widely used library for network analysis, WinPcap, highlights its bottlenecks, and proposes some solutions that almost double the overall speed, thus enabling the deployment of software-based tools on high speed networks.

1. Introduction

The capabilities of modern networks are growing constantly, along with their bandwidth and their complexity. Operations like monitoring, troubleshooting, and securing a network are becoming more complex, and require both high levels of competence and specialized tools. Some of these tools (like network analyzers, firewalls and monitoring-capable devices) are based on proprietary hardware. However, hardware solutions are usually expensive, difficult to deploy (e.g., hardware cannot be duplicated and easily moved) and they have a low degree of flexibility compared to software solutions.

So far, a fairly large number of software-based solutions (often implemented as extensions to a standard operating system) for providing software applications with real-time access to raw network data have been proposed. These solutions are usually implemented as libraries, like the well-known libpcap [1] and WinPcap [3], which are available on a large number of operating systems (OSes). These libraries export a set of primitives that allow applications to interact with the network without the intermediation of any other layer. Software components are easy to deploy and very flexible: a single packet capture component can provide low-level access to a wide range of applications (e.g. firewalls, NAT, sniffers, network monitors, etc.). Moreover they are inexpensive and can be updated easily, which is the reason for many professionals preferring software tools for monitoring and analyzing networks. However, performance is the Achilles' heel of software-based tools, which makes hardware solutions a must when dealing with high-speed networks. Although current CPUs are very powerful, there is still no way to perform software real-time traffic analysis on a links operating at multi-gigabit speeds.

Despite a certain degree of high-level research by several teams all around the world (notably [10], [13]), improving the overall performance of a network analysis tool is still an open issue. The biggest problem of current approaches is that they focus on some specific components of traffic analysis (for example packet filtering) and propose solutions for improving the performance of these functions. The outcome of this work shows that this approach is not effective given that users are interested in the performance of the whole traffic analysis system rather than a single component. Specifically, this work identifies the components involved in network analysis and measures their relative weight. A set of optimizations is then implemented in an experimental version of the WinPcap library and tested with the aim of quantifying the improvement. As shown by the obtained figures, optimizing a component that accounts for a small

percentage of the overall system performance is not particularly beneficial from the end-user point of view.

This paper is organized as follows. Section 2 provides an overview of related research activities. Section 3 describes the architecture of WinPcap as an example of a typical component that extends the OS to permit raw-access to the data traveling on the network. Section 4 presents the results of a detailed performance evaluation by characterizing each component involved in software analysis, including capture-specific components, OS and application-level aspects. Section 5 shows a set of optimizations, quantify their relative importance, and measures how they affect the overall capture process. Finally, conclusive remarks are made in Section 6.

2. Related Work

The CMU/Stanford packet filter [14] (or CSPF) is the first publicly available system for packet filtering and user-level access to the data-link layer and ancestor of most current solutions. It introduced, among other things, the concept of a *filtering virtual machine*, which is basically a virtual CPU (with registers, etc.) with a compact and efficient instruction set targeted to packet filtering. A filter is compiled to a small program executable on the virtual machine.

One of the most important advances in the field is due to McCanne and Van Jacobson who published the Berkeley Packet Filter (BPF) [2] in 1993. This improves CSPF by limiting the number of copies packets undergo and by defining a new, more efficient, register-based virtual processor with a small but complete instruction set (i.e. basically load, store, compare and jump instructions). BSD-derived OSes still provide BPF as a default capture facility; other systems have compatible implementations as well. The BPF virtual processor is the preferred base for the `libpcap` library.

The Mach Packet Filter (MPF) [9], PathFinder [12], DPF [11], BPF+ [10] are examples of works focusing on improving traffic analysis performance by focusing on the filtering process. Packet classification [15], another traffic analysis component that is conceptually very similar to packet filtering, got a lot of attention also because of its role in the packet forwarding process within routers.

On the other side, only a few works focus on other aspects of packet capture, like buffering and copying. The NFR team proposed an enhanced version of BPF with a bigger buffer and examined the possibility to use a shared buffer to prevent packets from being copied twice [13]. WinPcap [3], an open-source Windows library, improves `libpcap` by implementing a more efficient buffering system with respect to the memory occupancy. However, no previous work focused on the whole analysis process, which is the objective of this paper. The obtained results show how existing optimizations account only for a small percentage of the overall cost, in terms of execution time, of the capture process.

3. Packet Capture Architectures

This Section defines the model used in the context of this work by identifying the components of a typical architecture for packet capture and traffic analysis. Particularly, we focus on the path followed by a packet that is received by the Network Interface Card (NIC), transferred to the workstation main memory, and to the final application through the intermediation of a device driver and the OS. Even though the NPF (NetGroup Packet Filter) [3] architecture, which is derived from BPF and embedded in the WinPcap library, is often specifically referred to, the basic principles are common to many other solutions. The steps, and involved components, required by WinPcap to process an incoming packet and deliver it to the application are shown in Figure 1 and Figure 2.

3.1. Network Card and NIC Device Driver

Modern NICs have a truly limited amount of on-board memory, usually a few Kbytes. This memory is required to enable the receiving and sending packets at the full link speed, independently of the host workstation capabilities. Moreover, NICs perform some preliminary checks, such as CRC errors, short Ethernet frames, while packets are stored in the on-board memory so that invalid frames can be discarded immediately.

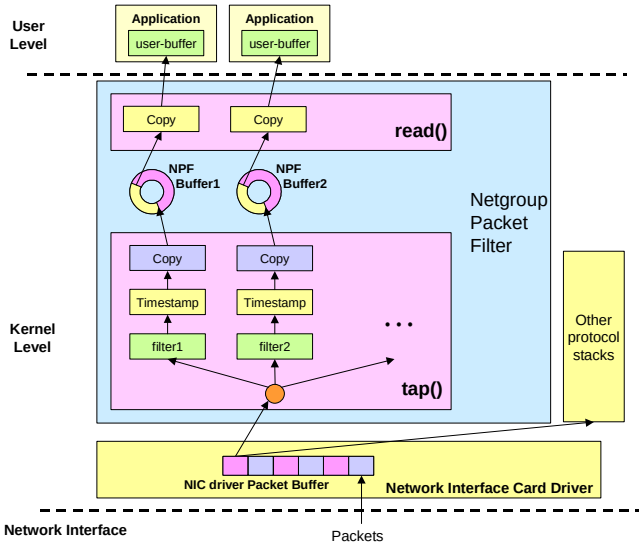

Figure 1. NPF Structure.

After a valid packet has been received by the NIC, this generates a request toward the bus controller for a bus-mastering data transfer. At this point, the NIC takes control of the bus, transfers the packet to the NIC buffer in the

workstation's main memory (see Figure 2), releases the bus, and generates a hardware interrupt toward the Advanced Programmable Interrupt Controller (APIC) chip. This chip wakes up the OS interrupt handling routine, which triggers the *Interrupt Service Routine* (ISR) of the NIC device driver.

The ISR of a well-written device driver has little to do. Basically it checks if the interrupt relates to itself (a single interrupt can be shared among several devices in x86 machines) and acknowledges it. Then, the ISR schedules a lower-priority function (called *Deferred Procedure Call*, or DPC) that will later process the hardware request and notify the upper-layer drivers (i.e., protocol layer drivers, packet capture drivers) that a packet has been received. The CPU will process the DPC routine when no interrupt requests are pending. Interrupts coming from the NIC are disabled when a NIC device driver is performing its work, because the processing of a packet has to be completed before the next one is served. Moreover, since interrupt generation is a very costly operation, modern NICs allow more than one packet to be transferred in the context of a single interrupt, so that an upper-layer driver is able process several packets each time it is activated.

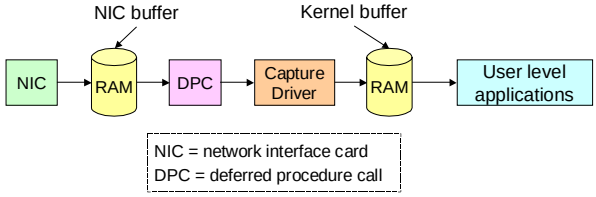

Figure 2. Path from NIC to applications.

3.2. Packet Capture Driver

Packet capture components are usually transparent to other software modules like protocol stacks, thus not influencing the standard system's behavior. They just insert a hook in the system so that they can be notified — usually through a callback function called `tap()` — as soon as a new packet arrives from the network. Packet capture components are usually implemented as network protocols drivers in Win32.

The first action performed by the `tap()` is *filtering*, i.e. packets are analysed to detect whether they are interesting for the user. Being derived from the BPF, the filtering engine of NPF is a virtual processor with a simple set of instructions that is able to perform some basic processing on a generic buffer of bytes — the packet dump. WinPcap (and `libpcap`) provides a user level API that transforms a high level expression (e.g. "*pick up all UDP packets*") into a set of pseudo instructions (e.g. "*if the* `ethertype` *field of the Ethernet header is IP and the* `protocol type` *field of the IP header is equal to 17, then return* `true`") and sends them to the filtering machine, activating it. The presented architecture applies the filter to the packet while it is still in the NIC driver's buffer, thus avoiding further copies of non-conformant packets, although they already consumed bus resources because the have been transferred into the system memory.

Packets accepted by the filter are associated with physical layer information, such as length and reception timestamp, that might be useful for applications accessing and processing them. Packets are then copied into a buffer, usually known as *kernel buffer*, that stores packets awaiting to be transferred to user-level (see Figure 2). The size and the architecture of this buffer are important parameters for the performance of the capture process. For instance a large and well-engineered buffering system is able to compensate for the slowness of user-level applications during bursts and to reduce the number of system calls required to transfer data from the capture driver (i.e. kernel buffer) to the application.

User-level applications retrieve packets from the kernel buffer by means of a `read`-like system-call. When NPF is deployed, this call triggers the invocation of the hook function `read()` (see Figure 1), which checks the status of the NPF kernel buffer: if the buffer is not empty, its content is transferred to a user-allocated memory, indicated as *user-buffer* in Figure 1. The application is awoken as soon as the data has been copied to user-level so that it can begin processing the packets.

4. Performance evaluation

This section presents the results of a detailed measurement campaign on a network analysis system. The objective is to determine the efficiency of the capture process as a whole and the exact amount of resources required by each of the components described in Section 3.

In order to be as general as possible, processing costs are expressed in CPU clock cycles. In fact, this measurement unit can be used to compare the performance of significantly different systems because it does not depend on absolute time and CPU speed.

4.1. Testbed

Figure 3 shows the testbed used for the profiling: two PCs directly connected through a Fast Ethernet link. One PC acts as traffic generator, while the other is used for the actual tests and has been installed with a modified version of WinPcap that includes profiling extensions for measurement purposes. Particularly, profiling extensions make use of performance monitoring counters available in the Pentium family of microprocessors [5] [6]. Every processor of this family has a certain number of internal counters (whose type and number varies according to the processor model) that can be programmed to keep track of

events such as the number of instructions decoded, the number of interrupts received, the number of cache loads, and more. For example, the `CPU_CLK_UNHALTED` counter stores the number of effective clock cycles spent by the CPU in a given time interval, discriminating between the ones consumed at user-level and the ones consumed at kernel-level. A program can retrieve these counters by means of the `rdpmc` instruction.

Another profiling extension makes use of a custom Dynamic Link Library (DLL) that can be used by a kernel driver to measure the CPU clocks required by a specific portion of code. This library uses the `rdtsc` (ReaD Time-Stamp Counter) x86 instruction to determine the exact amount of clock ticks consumed by the CPU during the execution of the given code.

Finally, profiling extensions use sampling techniques available though the Intel Vtune Performance Analyzer [16]: the CPU is frozen at precise intervals and its state is inspected to determine which driver/function is being executed. This sampling process, continued over a significant amount of time, gives a statistical insight on which software modules are involved into packet capture and their relative weight.

The two PCs are equipped with network adapters from different vendors. A 3Com 3C996 Gigabit Ethernet network card, operated at 100 Mb/s to avoid saturation of the host computer hardware, was used most of the times because of its excellent performance. Some detailed analysis was performed also on an Intel 85527 Fast Ethernet adapter because of the availability of the source code of its driver, which was provided by Microsoft in the Driver Development Kit [7]. This is one of the few cases in which the source code of a NIC driver for Windows is available and allowed a more precise study of performance and bottlenecks.

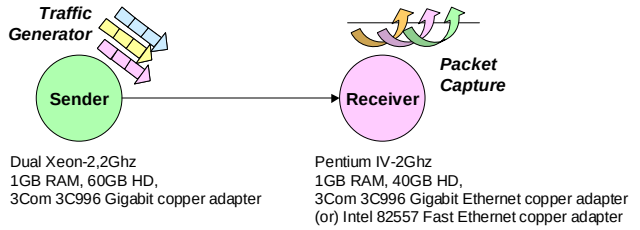

Figure 3. Testbed.

A traffic generator tool able to generate bursts of packets with precise frame rate is installed on one of the PCs. The generated traffic is directed to a non-existent host on the network so that the protocol stacks of the two PCs are not affected by the traffic. Both PCs were running Microsoft Windows XP Professional. Tests were carried on at different packet rates, although most of them refer to the maximum number of frames per seconds allowed on a Fast Ethernet link (148809 frames/sec, with 64 bytes frame size), which is the worst operating conditions for a packet monitoring and analysis tool in terms of CPU processing.

Test traffic pattern is fairly simple (constant frame rate) because our objective is to test the software under the maximum load for long period of time. Thus, more realistic traffic patterns (e.g. variable size busts, Poisson arrival rates, etc) are out of scope since they represent a better operating condition compared to our choice.

4.2. External processing cost

Processing a packet involves several components, like the NIC driver and the OS, which are not strictly part of the capture architecture. The cost associated with the intervention of such components, in terms of the time they require to process a packet, is called *external processing cost* and is shown to be of primary importance.

4.2.1. Operating System

The OS is the first software component involved when the network card receives a packet. The cost of OS processing varies with the packet rate, but it is mostly proportional to the number of interrupts, which is the mechanism used by the NIC to inform the system that a packet has been received and it is waiting for processing. Particularly, every interrupt requires approximately 2700 clock cycles on our test machine. In the tests, the 3Com NIC generated 2999 interrupt/sec at 148K fps, which correspond to an average of 54 clock cycles per frame. This cost varies with different adapters and frame rates, and is due to the OS kernel for performing operations like interrupt handling. For instance, with the adapters under test, three OS functions account for a remarkable amount of clock cycles: `HalBeginSystemInterrupt()` (that raises the current interrupt level and masks the interrupt controller), `KeDispatchInterrupt()` (that executes the DPC routine of the NIC driver) and `KeInitializeInterrupt()`, that is undocumented (but it will probably decrease the current interrupt level and un-mask the interrupt controller).

4.2.2. NIC and device driver

Although the network card performs its job without requiring any effort from the central CPU, its behavior can influence some other components, notably the OS and the device driver processing. For instance, the number of interrupts to be served (which influence the OS cost) and the number of I/O operations to access registers on the NIC (which influence the device driver cost) have a significant impact on performance. Referring to the latter cost, the ISR function (which is usually called once for each interrupt and it is the first function of the device driver) is

very simple but quite costly (about 850 clock cycles[1]) because it performs a couple of I/O operations on the NIC to signal that the driver is currently handling some packet.

Performance can be improved by being able to retrieve several packets from the NIC buffer in response to a single interrupt if the load exceeds a certain value. This decreases the OS cost (smaller number of interrupt) and the driver cost (smaller number of I/O operations on the card). The obtained results show that the number of packets per interrupt transferred grows about linearly with the network load. For instance, the 3Com NIC reaches an average value of 49.61 frames served per interrupt (corresponding to 2999 interrupt/sec) when receiving 148K frames per second. This means that the relative overhead of low-level components (interrupt handling, NIC driver) on packet processing is higher for low packet rates and becomes progressively less significant for growing packet rates.

According to our tests, the cost per packet of the NIC driver (at the maximum rate of 148809 packets per second) when used in conjunction with WinPcap, with no other protocols active on the machine, is 2260 clock cycles with the Intel 85527 adapter and 1497 clock cycles with the 3Com 3C996 adapter.

Finally, there is an additional cost that cannot be quantified. Packets are transferred from the NIC card to the main memory through a bus-mastering transfer, which does not consume CPU clocks. However, the bus (which is a resource anyway) is busy during the transfer. For high loads this can be a non-negligible cost even if this process does not consume (apparently) any CPU clocks, because the bus is unavailable and it can delay CPU requests.

4.3. Capture driver

This Section analyses the cost of all components of the capture driver, i.e., the cost of the packet path from the NIC device driver (and OS) to the user-level application.

4.3.1. Filtering process

The filtering process deserves particular attention, because it is the only component that handles all incoming packets. Obviously, its cost depends not only on the efficiency of the filtering engine, but also on the complexity of the filter, i.e., on the number of checks that are to be done on each packet. Figure 4 shows the cost of three filters (in order of increasing complexity): a simple one that accepts only IP packets (it requires the execution of 3 pseudo-instructions of the NPF virtual processor), one that checks the packet's TCP port against 5 different values (21 pseudo-instructions), and a more complex one that checks the packet against 10 IP addresses and 10 TCP ports (50 pseudo-instructions).

[1] Since this cost depends solely on the NIC and the system bus architecture, it has no relationship with clock cycles. However, we use this measurement unit for coherence with the rest of the paper.

Packets are generated so that all the filtering code must be executed before the filter returns. As expected, the number of clock cycles grows linearly with the number of instructions, as shown in Figure 4. Typical filters require a few hundreds clock cycles.

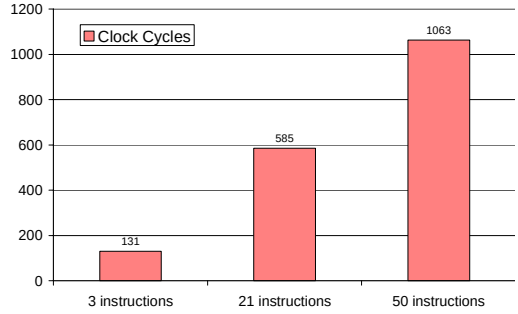

Figure 4. Filters with different complexity.

4.3.2. Memory copies

As explained in Section 3, every packet is copied twice in the main memory before reaching the user (Figure 1): the first copy transfers the packet from the NIC buffer to the kernel buffer (Figure 2), the second one transfers it in the user-application buffer. Figure 5 shows the cost of the two copies, in CPU cycles per byte, on the machine under test.

According to the NDIS specification, the first copy is performed by the `NdisTransferData()` function. The cost of this function is particularly high for two reasons.

1. Some additional overhead is required before the copying process. According to the DDK documentation [7] a driver must use this function since the whole packet could not be available when the NIC driver leaves the control to the packet driver. Thus, this function first checks whether the whole packet has been transferred in memory by the NIC; if not, it waits until the transfer is complete.
2. The function operates on data that is not in the CPU cache. The packet to be copied has just been transferred from the NIC on-board memory to the main memory by means of a bus-mastering transfer (see Section 3.1).

Previous points explain the cost of the first copy as shown in Figure 5: some constant processing is due independently to the amount of data that has to be transferred (this explain higher costs per bytes in case of small packets). For larger packets, these costs are spread over a larger mount of data, accounting for a smaller value for each byte transferred.

The second copy uses a standard C library function (such as *memcpy()*). Its results are comparable with the ones obtained when copying mostly non-cached user level memory buffers, since most packets in the kernel buffer are not in the CPU cache. The cost per byte slightly increases because the larger the packet is, the higher the

probability not to have it in cache (at least partially). The cost per byte increases also according to the kernel buffer size. For instance, if the amount of data in the kernel buffer is small, there is a higher probability that most of it is still in the CPU cache since its transfer from the NIC buffer during the first copy.

Summarizing, the cost of the first copy varies between 540 and 10500 clock cycles per packet, while the one of the second copy varies between 259 and 8550 clock cycles per packet. Actually, considering a 20 bytes header (containing a timestamp, the packet length and other information) that is added to each packet before storing it in the kernel buffer, the total cost of the second copy varies between 364 and 8664 clock cycles per packet.

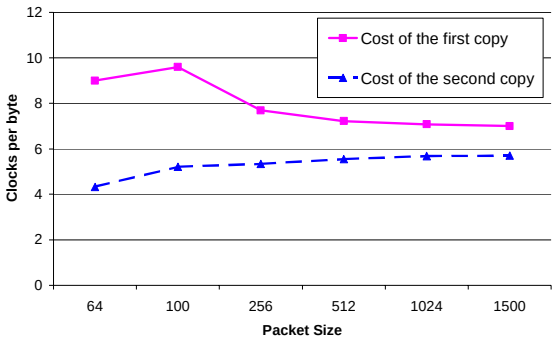

Figure 5. Cost per byte of memory copies with different packet sizes.

4.3.3. Interaction with the application

All the interactions between the application and the packet driver are done via system calls. Windows provides the `ReadFile()`, `WriteFile()` and `DeviceIoControl()` system calls for I/O purposes. All these calls involve two context switches[2]: the first one transfers the execution from user level (the application) to kernel level (the driver), the second one returns the control back to user level.

The context switch is well known as being a complex (it usually involves the generation of an interrupt and the initialization of some OS data structures) and therefore costly process. On the machine deployed for our measurements, a *read()*-like system call requires 33500 clock cycles. Such a high cost makes copying a single packet per system call very ineffective; therefore the capture driver transfers blocks of packets each time an application invokes a system call. The number of packets transferred within a system call is determined by the occupation of the kernel buffer and grows proportionally with the CPU load. This, in turn, depends on the complexity of the user-level application (if an application requires a long time to process packets, it retrieves data from the kernel buffer at long time intervals and the kernel buffer is not adequately drained) and on the cost of the capture driver processing (this code runs at higher priority, therefore the kernel buffer is constantly filled up).

Since the frequency of the read operations, and hence the number of packets retrieved per call, are highly variable, a general characterization of the operation cost in clock ticks per packet is not possible. On an overloaded machine (i.e. 100% CPU usage) receiving minimum-size frames, the capture driver transfers 256 Kbytes per system call[3], corresponding to 3200 packets (including the 20 bytes header added by the driver). In this situation the average cost of a context switch per packet is approximately 10 clock cycles, which is negligible given the cost introduced by other components.

4.3.4. Other processing components

Although the general feeling is that a capture driver spends most of its execution time in filtering and copying packets (and this explains why almost all the performance improvement work in literature focuses on one of them), our profiling revealed that other factors significantly affect the cost of capturing a packet. Among them, timestamp gathering is the most remarkable.

The NPF driver obtains the timestamp for a packet through the `KeQueryPerformanceCounter()` Win32 function, which is the only kernel function that provides a time reference with microsecond precision. The cost of this function is very high because it has to interact with the system timer chip: approximately 1800 clock cycles[4] on the machine used in this work. Paradoxically, this function requires multiple microseconds to return a result with an accuracy of a microsecond. However, this bias is almost constant, therefore this timestamp can be considered a valid measure of their arrival time.

Additional costs include the interaction with NDIS and with the kernel (most of these interactions make use of callback functions, which are costly mechanisms), the management (mapping and unmapping) of memory buffers in use in the kernel, and the creation of the header that NPF adds to every packet. In summary, the costs associated to the execution of a packet driver, excluding filtering and copying amount to about 830 clock cycles.

4.4. Total processing cost

Figure 6 shows a summary of the results presented in current Section by plotting the relative cost of each operation relative to the processing of 64 byte packets at

[2] The term *context switch* is used improperly here, since a transfer from user level to kernel level is actually a privilege level switch and does not necessarily implies a switch of the execution context.

[3] This value is an upper bound chosen by the current WinPcap.

[4] The cost of this function has been measured on several single-processor machines with equivalent results.

148 Kfps. Results have been obtained with the 3Com 3C996 Gigabit adapter and the 21 pseudo-instruction filter; the total cost to process a packet is 5680 clock cycles.

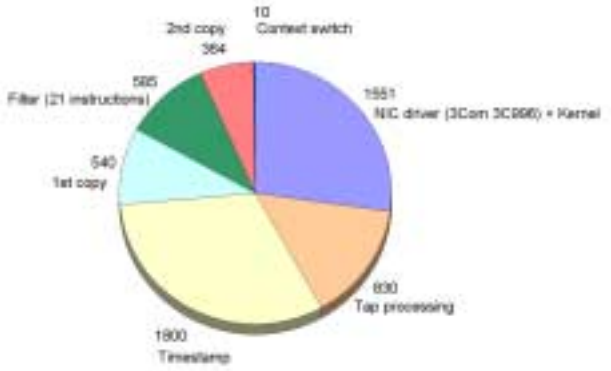

Figure 6. Details of CPU clock cycles.

As it is evident from Figure 6, the costs associated to the timestamp gathering and to the NIC driver are predominant in case of short packets. Since both costs depend mostly on the hardware, software optimizations are useless in their respect. Some minor optimizations are possible into NIC drivers, but they are usually made unfeasible by vendors because they do not publicly release the source code of their Win32 drivers. In any case, NIC driver optimizations are far less useful than a more intelligent chipset on the NIC card.

Most notably, Figure 6 shows how most optimizations present in the literature, which focus on copying and filtering, aim at reducing a cost that accounts only for 15% of the total processing time, which is a very limited value indeed.

Considering mostly short packets for analyzing performance is not a limitation of the current work. For instance, a large set of network analysis tools (notably, sniffers and network monitors) requires only the initial part of the packet, e.g. the first 98 bytes, so that the capture driver will discard all the remaining of the packet. This confirms our assumption that the profiling has to be done considering particularly short packets.

4.5. Extending the validity of the results

Although presented results refer to a specific tool (WinPcap) on a specific platform (Win32), their validity is more general. The costs related to WinPcap (namely the tap processing, first and second copies, filtering) are quite similar to the same costs on other architectures (for instance, NPF is quite similar to BPF). A similar rational is behind costs related to the operating system: NIC driver, timestamp gathering, and context switch. NIC driver costs may be reduced by a network card design that pushes in hardware some of the operation normally done in software, but this could be rather expensive. Hardware-based timestamp gathering in one of the most viable optimizations: the widely used DAG cards from Endace [17] provide such an example. For instance, Intel-based hardware does not have any simple way to get sub-microseconds timestamps because of the lack of specialized chips in the x86 reference design and more precise timestamps must be gathered by interpolation (e.g. by means of the CPU hardware counters). In addiction, microsecond precision involves reading data from the 8253/8254 chips (or equivalent), whose access is rather slow because they require IN/OUT operation through the system bus.

For the last point, the context-switching impact is negligible and it does not change considerably among different operating systems (because, for instance, is one of the most carefully optimized parameter in modern operating systems).

5. Optimization

This Section presents and evaluates optimizations that have been implemented in the NPF with the purpose of limiting bottlenecks highlighted in the previous Section.

5.1. Filtering process

The filtering system used by WinPcap, the BSD Packet Filer (BPF), was proposed in 1993 in [2]. Several other filtering systems exist in literature [9] [10][11][12], but their speedup with respect to the BPF is negligible in the most common operating conditions.

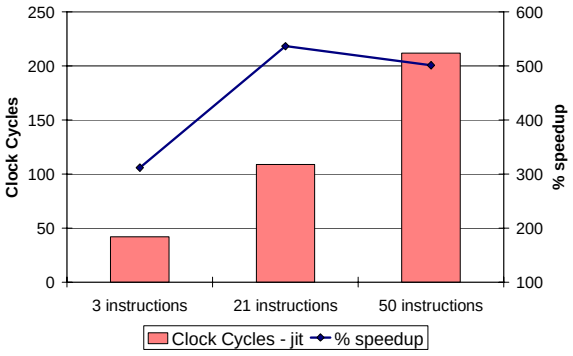

Figure 7. Cost and speedup of three filters with different complexity using JIT compilation.

Among the solutions to optimize BPF, dynamic code generation (i.e. the translation of packet filters into CPU-native executable code) guarantees impressive performance improvements according to [11] and [10]. Therefore, a Just In Time (JIT) engine that translates BPF filters into 80x86 binary code, was implemented and incorporated in the NPF. As shown in Figure 7, the speedup brought by this optimization varies from 3.1 to 5. This corresponds to an 8% improvement of the total capturing cost (with a 21 pseudo-instruction filter).

5.2. Memory copies

As stated before, the cost of the first packet copy (from the NIC memory to the kernel buffer) is higher than the second. One reason is the additional processing incurred by the `NdisTransferData()`. However, we noticed that almost all the network controllers (hence the vast majority of network adapters) transfer a *whole* packet in memory before notifying the NIC driver, therefore the NPF driver receives it in a single contiguous buffer. In this case, it is possible to copy it with a standard C library function, with the result shown in Figure 8, otherwise the old method is used. The second copy is still a bit faster because of the higher probability that data is in the CPU cache and because the packets are moved in blocks rather than one at a time. However, the cost of the two copies is comparable and presents similar trends.

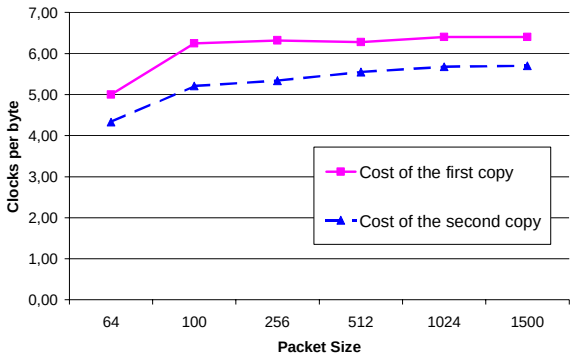

Figure 8. Cost per byte of memory copies with different packet sizes.

Thanks to this optimization the average cost of the first copy decreases from 540 to 300 clock cycles per 64 byte packet on a fully loaded machine, while the cost of the second copy remains unchanged. This corresponds to a 4% improvement of the total capturing cost.

5.3. Timestamp

The usage of `KeQueryPerformanceCounter()` to obtain a timestamp with a microsecond precision can be avoided thanks to the TimeStamp Counter (TSC) included in most 32-bit Intel processors. This high performance counter is incremented at every processor clock cycle, so its precision is equivalent to the CPU frequency. The x86 assembler provides a very fast (one-cycle) instruction to obtain this timestamp, `rdtsc`. The cost of timestamp collection with `rdtsc` is 270 clock cycles, mainly due to two 64-bit divisions that are necessary to convert it into a standard `struct timeval` value. This optimization has a speedup of 6.6 corresponding to a 27 % improvement of the total capturing cost. However, this optimization is by default disabled on the standard distribution of NPF because of the strong dependence on the model (`rdtsc` works only on Intel CPUs or compatible ones, such as AMD ATHLON processors) and speed (some processors adjust their frequency according to external parameters, like the battery level) of the processor. However, this demonstrates that the addition of a simple hardware-based timestamp can improve the packet processing significantly.

5.4. Optimization of the *`tap()`* function

The use of standard C library memory copy routines instead of `NdisTransferData()` also enables a simpler `tap()` function. The process of transferring a packet with `NdisTransferData()` requires the allocation of a structure that will contain the packet during the transfer and the provision of a callback function that will be invoked when the copy is finished. Avoiding these steps when `NdisTransferData()` is not used significantly impacts performance. The simplification of some points of the `tap()` processing reduces its cost from 830 to 560 clock cycles — a 5% improvement of the total capturing cost.

5.5. Total processing cost with the optimizations

Figure 9 illustrates the cost for the kernel-level processing of a 64 byte packet in the same conditions of Figure 6 but with the optimizations presented in this Section. The cost of the optimized processing is 3164 clock cycles, i.e., slightly more than half of the cost without optimizations.

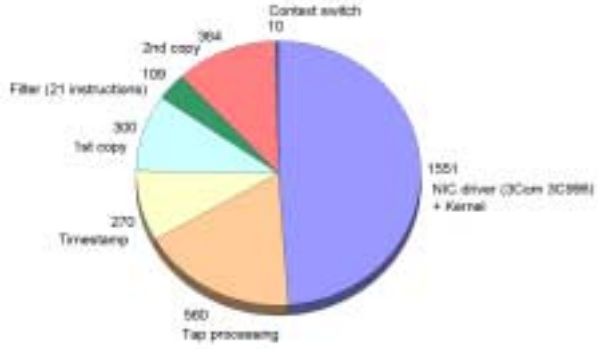

Figure 9. Details of CPU clock cycles.

It must be noted that 49 % of the CPU time is absorbed by the NIC driver and by the kernel interrupt processing. In fact, the cost of these two components is not affected by the optimizations. The speedup of the other components due to the optimizations is approximately 2.6.

5.6. Possible hardware speedups

Figure 9 shows that most of the costs are due to factors that are outside the packet capture components. Endace

[17], a New-Zealand-based company, provides a set of optimized cards for packet capture. Their card does not have sophisticated hardware optimization, but they fix the problems where they are, i.e. they limit the overheads of the operating system. Basically, these cards generate a timestamp (in hardware) for each packet received, and they transfer the packet in the system memory, being clever enough to manage the buffering mostly in hardware. Applications (in user space) can read the data without any other intermediate layers because operating system structures (such as most of the work done by a NIC driver) and the OS-native protocol stack are bypassed completely.

Although we do not have any experimental data (these cards are available only on Linux, while our measurement infrastructure works only on Win32), we can see that their packet-processing overhead is limited to the filtering, plus some additional overhead that can be seen comparable with the tap processing (buffer management cannot be done totally in hardware and some interaction with the kernel-space is needed). In this case, the total processing cost can be estimated as about 670 clock ticks, which is a speedup of 8.4 compared to the original system, and 4.7 compared to an optimized all-software system.

Such a limited hardware support can guarantee a new life for software-based packet capture and analysis applications.

6. Conclusions

This paper presents the results of the profiling and optimization of the software chain at the basis of network analysis and monitoring tools, e.g. a sniffer. The work identifies the components involved in packet capture and measures their cost in terms of CPU clock cycles required for their execution. A valuable result of this study is the quantitative conclusion that, contrary to common belief, filtering and buffering are not the most critical factors in determining packet capture performance. Optimization of these two components, that received most attention so far, is shown to bring little improvement to the overall packet capture cost, particularly in case of short packets (or when small snapshot length are needed). The profiling done on a real system shows that the most important bottlenecks lie in hidden places, like device driver, interaction between application and OS, interaction between OS and hardware.

Thus, since packet capture encompasses various interacting elements, optimizing it requires keeping in mind the overall process rather than concentrating on a single component. This paper shows that some limited optimizations in the right place are far more noticeable than architectural changes, such as some involving filtering and buffering that were proposed in literature. Furthermore, this work shows that the performance of software-based traffic analysis can be pushed further despite the fact that the employed technologies are considered substantially mature, for example by means of some limited hardware support like timestamp gathering, or bypassing operating system overheads.

All the details and measurements made in the context of this work are available on the WinPcap home page [8], along with documentation, source code and examples. Most of the presented optimizations are currently implemented in version 3.0 of WinPcap.

Bibliography

[1] V. Jacobson, C. Leres and S. McCanne, libpcap, Lawrence Berkeley Laboratory, Berkeley, CA. Initial public release June 1994. Available now at http://www.tcpdump.org/.

[2] S. McCanne and V. Jacobson, The BSD Packet Filter: A New Architecture for User-level Packet Capture. Proceedings of the 1993 *Winter USENIX Technical Conference* (San Diego, CA, Jan. 1993), USENIX.

[3] Fulvio Risso, Loris Degioanni, An Architecture for High Performance Network Analysis. Proceedings of the 6*th* *IEEE Symposium on Computers and Communications* (ISCC 2001), Hammamet, Tunisia, July 2001.

[4] Microsoft Corporation, 3Com Corporation, NDIS, Network Driver Interface Specification, May 1988.

[5] Intel Corporation, IA-32 Intel ® Architecture Software Developer's Manual Volume 2: Instruction Set Reference, Order Number 245471-006.

[6] Intel Corporation, IA-32 Intel ® Architecture Software Developer's Manual Volume 3: System Programming Guide, Order Number 245472-006.

[7] Microsoft Windows Driver Development Kits (DDKs), available at http://www.microsoft.com/ddk/.

[8] The NetGroup at Politecnico di Torino, WinPcap: WinPcap web site, http://winpcap.polito.it/.

[9] M. Yuhara, B. Bershad, C. Maeda, J.E.B. Moss, Efficient Packet Demultiplexing For Multiple Endpoints And Large Messages. In Proceedings of the *1994 Winter USENIX Technical Conference*, pages 153-165, San Francisco, CA, January 1994.

[10] A. Begel, S. McCanne, S.L.Graham, BPF+: Exploiting Global Data-flow Optimization in a Generalized Packet Filter Architecture, Proceedings of *ACM SIGCOMM '99*, pages 123-134, September 1999.

[11] Dawson R. Engler, and M. Frans Kaashoek, DPF: Fast, Flexible Packet Demultiplexing, in Proceedings of *ACM SIGCOMM '96*.

[12] Mary L. Bailey, Burra Gopal, Michael A. Pagels, and Larry L. Peterson. PATHFINDER: A Pattern-Based Packet Classifier. In Proceedings of the *First USENIX Symposium on Operating Systems Design and Implementation*, pages 115–123, Monterey, CA, November 1994.

[13] Marcus J. Ranum, Kent Landfield, Mike Stolarchuk, Mark Sienkiewicz, Andrew Lambeth, and Eric Wall (Network Flight Recorder, Inc.), Implementing a Generalized Tool for Network Monitoring, *LISA 97*, San Diego, CA, October 26-31, 1997.

[14] Jeffrey C. Mogul, Richard F. Rashid, Michael J. Accetta. The Packet Filter: An Efficient Mechanism for User-Level Network Code. In Proc. *11th Symposium on Operating Systems Principles*, pages 39-51. Austin, Texas, November, 1987.

[15] Pankaj Gupta and Nick McKeown, Algorithms for Packet Classification, *IEEE Network* Special Issue, March/April 2001, vol. 15, no. 2, pp 24-32.

[16] Intel Vtune Performance Analyzer, Intel Corporation, 2003. Demo available at http://developer.intel.com/software/products/vtune/vpa/.

[17] Endace Measurement Systems, web site at http://www.endace.com.