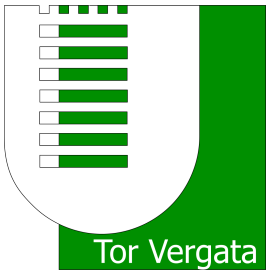


UNIVERSITY OF ROME TOR VERGATA

Faculty of Engineering

COMPUTER AND INFORMATION ENGINEERING

A.Y. 2017/2018

Master's Thesis

HUGE PAGES IMPACT ANALYSIS ON HIGH-SPEED NETWORK PACKET CAPTURE IN THE LINUX KERNEL

ADVISOR

Prof. Marco Cesati

CANDIDATE

Laura Trivelloni
0245894

CO-ADVISORS

Eng. Emiliano Betti
Eng. Pierpaolo Santucci

A papà che, ne sono certa, non avrebbe avuto dubbi.

Contents

Acknowledgments

I would like to express my sincere gratitude to my advisor, Marco Cesati, and to my co-advisors, Emiliano Betti and Pierpaolo Santucci, to have shared with me their extraordinary enthusiasm, their patience and especially their extensive knowledge and professionalism.

I am grateful to my mother and my family to have accompanied me on the road to this until the end, also when it was not so easy.

My father deserves a special mention. To me it is not hard to imagine how he would have been proud of me. I dedicate to him my thesis.

I would like to thank anybody who I shared with breaks, projects, anxieties and laughs during this years: Manu, Ovi, Marius and the others have made the University such a good place to me.

I wish to thank Luca to have given me back the chance to do this, too.

I would like to thank my wonderful love, Daniele: if he is with me, he makes every my personal fulfilment even more beautiful than it could ever be.

Abstract

In a world immersed in the network connections existing among different devices, there are systems where analyzing the data flows exchanged becomes a sensitive matter. The Network Traffic Analysis is an activity for monitoring what is happening into the network. It is aimed at recognition of the traffic characteristics and at other security goals as information gathering for fighting criminal or terroristic activities.

This thesis spark from this application scenario and it is focused on optimizing the performance of programs that monitor the packets arriving at a high speed network interface. A critical issue of any packet monitoring application consists of the relationship between the OS's buffer size where the packets are temporarily stored, and the time it costs to analyze all of them. The target we want to achieve with this experimental work is to change the management of the kernel side allocation of the socket buffer for reception of a high rate of incoming packets. We were inspired from existing frameworks for high-performance packet capture, finding in the huge page table mechanism a possible improvement for the operations on the memory resources. The Linux kernel supports huge pages as optimization of the memory access, reducing the overhead caused by the TLB miss events. We studied two solutions to extends the features of the `AF_PACKET` domain module of the Linux open-source operating system, working on the kernel version 4.20.

We analysed the packet socket interface implementation and the socket buffer

structure and management during the capture process. The study shows the feasibility of a modification of the buffer allocation that originally provides three types of memory requests, tempted in order of preference: direct reclaim; only virtually contiguous amount of memory; forced reclaim memory. We inserted into the module five new allocation policies: two for requesting a memory buffer backed with huge pages, using both the Linux kernel Transparent Huge Pages and the pre-allocated ones from the HugeTLB filesystem; one corresponding to each attempt of memory request in the original version. We exposed the choice of the allocation option to the user space. This is only for test purposes, making the comparison among different solutions easier.

Changing this subsystem impacts on the most common applications used for traffic analysis, as `Wireshark`, `Snort` or `tcpdump`. Indeed, it is important to work for integration with those softwares to avoid the isolation of the solution. The effect led to the mentioned softwares is not direct. These softwares uses the `pcap` library that implements higher level APIs for the packet sockets. Hence, this library is the means by which it is possible to complete the integration. But since adapting a stable, widespread code, requires cascaded updates, this has a not negligible cost. Therefore, we move towards an alternative way, resorting to the so-called `LD_PRELOAD` trick. This trick exploits the dynamic linking. It allows to execute a *preloaded* implementation of a no-static function before the original one. This enables to intercept the calls to the symbols functions of the `pcap` library that needs changes, running an ad-hoc routine. We used an additional trick to differentiate the behavior of the preloaded functions. The trick regards the setting of an environment variable to keep globally the option to use for the block allocation.

In order to prepare a realistic environment to evaluate the solutions, we considered a synthetic network traffic using the Linux module `pktgen` properly configured. We

produced a `pcap` trace file with `tcpdump` to repeat the test with the same kind of incoming packets. Furthermore, in the user application that implements the capture process, we introduced a delay factor every a defined number of packets. The delay allows to simulate the processing time spent on the packets of interest and to study the behavior of the buffer when it accumulates packets.

We run experiments aimed at comparing the performance of all the analysed allocation policies. In addition to the comparison between the original block allocation routine and the new one, we included also every single type of memory request originally included in the previous version. In order to measure the performance of each solution, we discuss about which metrics to take into account. We used the library of the `perf` tool that is based on the values of the hardware-based performance counters present in the Performance Measurement Unit (PMU). We select the values of the number of TLB misses (both for load and store operation), the number of page-faults, the time elapsed. From these measurements we obtained the TLB miss rate computed as ratio between the number of TLB misses and the number of cache accesses.

We also studied the trend of the packet loss measured increasing the processing time per-packet. The packet loss is due to the overloading of the reception queue. It is an indication of the speed reached in the activity of remove packets from the buffer.

Our performance tests concerned packet capture at 10 GBit/s. The results shows a partial success. As expected, using buffers backed with huge pages translates into a decrease of the TLB miss rates compared to the original implementation. However, the improvement does not impact so much to affect the percentage of packet loss during the capture process. The frameworks for high-speed traffic created an own domain. Otherwise, in this case the attempt to change an existing domain does not came across such an high-performance guarantee, suffering a pre-designed subsystem.

Introduction

Since some decades, our world is immersed in the network connections existing among different kinds of devices: from the most powerful mainframes to the personal computers.

When we talk about any Internet connection we have to consider that there are data exchanged among the endpoints and that in certain cases some information contained in that data can be extremely sensitive.

Therefore, in some systems it is important to monitor what is happening into the network at packet level, going to analyze all the data flows involved. This activity is the **Network Traffic Analysis** and it is aimed at recognition of the traffic characteristics and at other security goals as information gathering for fighting criminal or terroristic activities. The most common tools for traffic analysis are Wireshark, Snort, tcpdump.

The Network Traffic Analysis provides statistics to identify type, size, origin and destination and the content of packets for data security purposes. But this does not concern only the network security, the opportunity to understand and evaluate the network utilisation and the download/upload speed is useful for monitoring purposes too.

The existing Network Interface Controllers are going to increase the level of the supported data rates up to 160 Gigabit/s. In the most common configuration the

10 Gigabit Ethernet interfaces are available on motherboards of high performance computers.

In parallel with the increment of the data rate of the incoming network packets a new need is born: the capability to face with a faster amount of data to be handled at arrival time.

This thesis spark from this application scenario and it is focused on optimizing the performance of programs that monitor the packets arriving at a high speed network interface.

A critical issue of any packet monitoring application consists of the relationship between the OS's buffer size where the packets are temporarily stored, and the time it costs to analyze all of them. In particular, we analyse the free and open-source **Linux** OS.

State-of-the-art Linux packet capture provides external frameworks to reach high-performance traffic reception. Some examples of these frameworks are: PFQ, Intel DPDK, PF_RING ZC.

If in a smaller interval of time more packets are captured by the NICs, the reception buffer needs more space or the reception handler has to be as quick as possible to avoid packet loss.

Anyway, finding a solution for the issue concerning the buffer size it is not resolvent at all. During the process of traffic analysis, the capture of the wanted type of packet occurs more or less frequently. This event requires all the processing time serving for every detection operation to be performed on the packet. It means that there is no buffer size that possibly can cope with a too-long packet-time.

Increasing the **buffer size** can lead to a performance improvement especially if it is coupled with a beneficial modality of resource allocation. Following this attempt

to improve the packet capture changing the size of the buffer, we have to deal with a larger memory area. The larger is the memory area, the larger is the set of memory addresses to translate. And also the time spent translating addresses affects the overall application performance.

Following the mechanism of the Linux memory management, we can assume that for allocating a larger memory area the use of larger pages of memory in the kernel can likely bring to a performance improvement. In Linux there are configurable larger pages of memory and they are called **huge pages**.

This is a mechanism used also in the frameworks mentioned before. In particular, **PFQ** reaches the line-rate on 10 Gbit links both in reception and transmission processes. The project was source of inspiration for this thesis to try to incorporate in the kernel its features changing the network domain it relies on: `AF_PACKET`.

Improvements are desirable both relating to the quality of the packet reception and to the rate of the reception respectively in terms of percentage of packet loss and number of bytes per second. Packet loss has to be minimized, because among those lost packets there could be some relevant data that can no more be taken into account by the applications.

In order to analyse and to study a solution concerning the problem, we worked on the **Linux** kernel version 4.20. Linux is an open source operating system and this allows to view and edit the source code of the software freely and publicly.

The target we want to achieve with this experimental work is to change the management of the kernel side allocation of the socket buffer for reception of a high rate of incoming packets.

The thesis is structured as follows:

- **Chapter 1** describes the memory management system of the Linux operating

system in order to give an overview of the used mechanisms and the available facilities

- **Chapter 2** describes the Linux networking interface from the point of view of the low-level one, and the usage and implementation in user and kernel space
- **Chapter 3** discusses the analysis of the problem: we describe the design and implementation details of the current algorithm, and we comment on the feasibility and the design of our proposed improvement
- **Chapter 4** discusses the proposed solutions from the design to the adaptation with the existing solutions
- **Chapter 5** describes the implementation details of the proposed solutions and the relating integration with the existing ones
- **Chapter 6** analyses the results of the performance evaluation of the implemented features and commented the comparison with the original ones
- **Chapter 7** addresses the conclusions of this thesis and possible future works

Chapter 1

Linux memory management

Linux memory management subsystem is responsible to handle memory resources allocation in the system. Its main functionalities are physical memory allocation and deallocation, implementation of virtual memory and demand paging, memory allocation both for kernel side structures and user space programs, mapping of files into processes address space, together with many other minor functions.

Linux is available for a wide range of architectures, so the memory description needs to be **architecture-independent**.

Many multi-processor machines are NUMA (*Non-Uniform Memory Access*) systems. In such systems the memory is arranged into banks that incur different access costs depending on the distance from the processor. Each bank is referred as node and for each node Linux represents it by a `pglist_data` structure, even if the architecture is UMA, containing information about per-node set of zones, lists of free and used pages and other statistics counters.

1.1 Physical memory

1.1.1 Zones

Every physical page frame in the system has an associated `page` structure which is used to keep track of its status. When allocating a page, Linux uses a **node-local allocation policy** to allocate memory from the node closest to the running CPU.

Often hardware constraints limit the usage of different physical memory ranges. Linux groups memory pages into **zones** according to their possible usage, based on the size of physical memory relating to the maximal addressable memory, the possibility to perform DMA (*Direct Memory Access*), and other constraints. These zones are architecture specific and are described by a `zone_struct` structure, and each one is of type `ZONE_DMA`, `ZONE_NORMAL` or `ZONE_HIGHMEM`. `ZONE_DMA` is memory in the lower physical memory ranges which certain legacy ISA devices require. Memory within `ZONE_NORMAL` is directly mapped by the kernel into the upper region of the linear address space. `ZONE_HIGHMEM` is the remaining available memory in the system that is not mapped into the kernel address space.

The x86_64 architectures need two `ZONE_DMA` regions for both supporting devices that are only able to do DMA to the lower 16M and 32 bit devices that can only do DMA areas below 4G.

The kernel maintains a list of free pages for each zone. When a request for physical memory arrives, the kernel satisfies the request using the appropriate zone.

1.1.2 Page allocators

Each zone has its own page allocator, which is responsible for allocating and freeing all physical pages for the zone and is capable of allocating ranges of physically contiguous pages on request.

The allocator uses a **buddy system** to keep track of available physical pages. In this scheme, adjacent units of allocatable memory are paired together. Each allocatable memory region has an adjacent partner. Whenever two allocated partner regions are freed up, they are combined to form a larger region. Conversely, if a small memory request cannot be satisfied by allocation of an existing small free region, then a larger free region will be subdivided into two partners to satisfy the request. Under Linux, the smallest size allocatable under this mechanism is a single physical page. Buddy system is indeed aimed at reducing external fragmentation, at the expense of internal fragmentation.

Kernel offers several specialized memory-management subsystems to use the underlying basic page allocator. The most important are the virtual memory system, the slab allocator and the page cache.

The **slab allocator** is used for allocating memory for kernel data structures and is made up of one or more physically contiguous pages. A cache consists of one or more slabs. There is a single cache for each unique kernel data structure. The number of kernel objects in the cache depends on the size of the associated slab. The slab allocator is aimed at reducing internal fragmentation, because several fixed-sized objects can be packed on several pages, and objects can cross the page boundaries.

The **page cache** memory management subsystem is used for caching files' content. Whenever a file is read, the data is put into the page cache to avoid expensive disk access on the subsequent reads. Similarly for write operation, the written data is placed in the page cache and eventually gets into the backing storage device. The written pages are marked as dirty and when Linux decides to reuse them for other purposes, it makes sure to synchronize the file contents on the device with the updated data.

1.2 Virtual memory

1.2.1 Regions

The addresses seen by user programs do not directly correspond to the physical addresses used by the hardware. Virtual memory introduces a layer of indirection that allows programs to allocate far more memory than is physically available, improving memory utilisation. Virtual memory also allows each programs to have its own unique view of the memory. It creates pages of virtual memory on demand and manages loading those pages from disk and swapping them back out to disk as required.

Under Linux, the virtual memory manager maintains two separate views of a process's address space: as a set of separate regions and as a set of pages. The first view of an address space is the **logical view**, describing instructions that the virtual memory system has received concerning the layout of the address space. In this view, the address space consists of a set of nonoverlapping regions, each region representing a continuous, page-aligned subset of the address space.

Each **region** is described internally by a single `vm_area_struct` structure that defines the properties of the region and information about data sharing protection and control, including the process's read, write, and execute permissions or files associated with the region. The regions for each address space are linked into a balanced binary tree to allow fast lookup of the region corresponding to any virtual address.

1.2.2 Page tables

The kernel also maintains a **physical view** of each address space. This view is stored in the hardware page tables for the process. The **page table** is a mechanism for translating virtual address into the corresponding physical address. Each entry identifies the exact current location of each page of virtual memory, whether it is on

disk or in physical memory. It is implemented as a multilevel structure depending on the architecture.

Each process maintains a pointer to its own **Page Global Directory** in `mm_struct->pgd` which is a physical page frame copied into the `CR3` register at context switch time. This frame contains an array of type `pgd_t` which is an architecture specific type defined in *asm/page.h*. Depending on the configured value of `PGTABLE_LEVELS`, the page table walks are longer or less long.

For example, for a 4-level page table each active entry in the PGD table points to a page frame containing an array of **Page Upper Directory** (PUD) entries of type `pud_t` which in turn points to page frames containing **Page Middle Directory** (PMD) entries of type `pmd_t` which in turn points to page frames containing **Page Table Entries** (PTE) of type `pte_t`, which finally points to page frames containing the actual user data. In Figure 1.1 is displayed an overview of the page tables walk for x86 architecture.

1.2.3 Virtual memory area

The physical view is managed by a set of routines, which are invoked from the kernel's software interrupt handlers whenever a process tries to access a page that is not currently present or accessible in the page tables.

Each `vm_area_struct` in the address space description contains a field pointing to a table of functions that implement the key page management functionality for any given virtual memory region. All requests to read or write an unavailable page are eventually dispatched to the appropriate handler in the function table for the `vm_area_struct`, so that the central memory management routines do not have to know the details of managing each possible type of memory region.

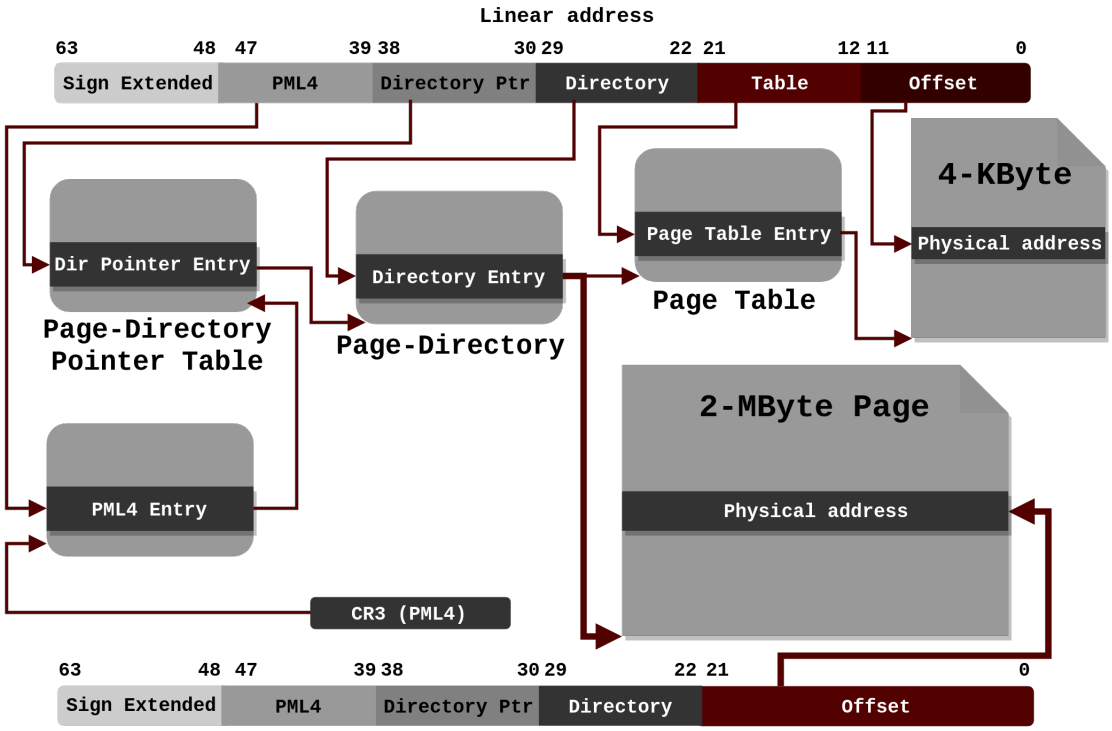


Figure 1.1: Four-levels page tables in x86_64 for 4K and 2M page size.

Linux implements several types of virtual memory regions. One property that characterizes virtual memory is the **backing store** for the region, which describes if the pages for the region come from a file or from no store.

A region backed by nothing represents demand-zero memory: when a process tries to read a page in such a region, it is simply given back a page of memory filled with zeros.

A region backed by a file acts as a viewport onto a section of that file: when a process tries to access a page within that region, the page table is filled with the address of a page within the kernel's page cache corresponding to the appropriate offset in the file.

The mapping of a region into the process's address space can be either private or shared. If a process writes to a **privately mapped** region, then the pager detects

that a copy-on-write is necessary to keep the changes local to the process. In contrast, writes to a **shared region** result in updating of the object mapped into that region, so that the change will be visible immediately to any other process that is mapping that object.

Linux reserves for its own internal use a constant, architecture dependent region of the virtual address space of every process, marking as protected the page table entries that map to the relating kernel pages. This kernel virtual memory area contains two regions. The first is a static area that contains page table references to every available physical page of memory in the system, so that a simple translation from physical to virtual addresses occurs when kernel code is run.

The remainder of the kernel's reserved section of address space is not reserved for any specific purpose, but are used by kernel facilities to allocate virtual memory, as `vmalloc()` and `vremap()` functions. The `vmalloc()` function allocates an arbitrary number of physical pages of memory that may not be physically contiguous into a single region of virtually contiguous kernel memory. The `vremap()` function maps a sequence of virtual addresses to point to an area of memory used by a device driver for memory-mapped I/O.

1.2.4 Translation Lookaside Buffer

When the processor needs to resolve a virtual address into a physical address, it must traverse the full page directory searching for the PTE of interest; this operation would normally occur for each memory reference.

To avoid this considerable overhead, architectures provide a **Translation Lookaside Buffer (TLB)**, which is a small associative memory that caches virtual to physical page table resolutions. The advantage that this component introduces is

based on the **principle of locality**: the large numbers of memory references tend to be for a small number of pages.

The amount of memory that can be translated by this cache is referred to as the **TLB reach** and depends on the size of the page and the number of TLB entries. Inevitably, a percentage of a program's execution time is spent accessing the TLB and servicing the TLB misses.

The amount of time spent translating addresses depends on the workload as the access pattern determines if the TLB registers are sufficient to store all translations needed by the application.

On a miss, the exact cost depends on whether the information necessary to translate the address is in the CPU cache or not. The **TLB miss time** is formulated as:

$$TLB_{MissTime} = \frac{(Cycles_{tlbmiss\_cache} + Cycles_{tlbmiss\_full})}{ClockRate}$$

where $Cycles_{tlbmiss\_cache}$ is the product between the miss rate and the time spent to resolve it and $Cycles_{tlbmiss\_full}$ is the product between the probability to find the cache full and the time spent to resolve the translation walking the page table levels.

If the TLB miss time is a large percentage of overall program execution, then a valid optimization goal might be to reduce the miss rate and achieve better performance. One means of achieving this is to translate addresses in larger units than the base page size, like **huge pages**, because larger pages will cover wider range with the same number of TLB entries.

1.3 Huge pages

1.3.1 Virtual address decoding

The Figure 1.1 describes the path walked during the translation operation from the virtual address to the physical address. The virtual address decoding in the figure concerns both the 4K-pages and 2M-pages in the Intel x86_64 architecture.

The page walk to obtain a physical address in the case of a page size of 4K is:

1. the entry in the first table (PML4) is reached summing the offset in the bit range 39-47 of the linear address to the base address of the table in the CR3 register.

2. the entry in the second level page table (PUD) is reached summing the offset in the bit range 30-38 of the linear address to the base address in the previous PML4 entry.

3. the entry in the third level page table (PMD) is reached summing the offset in the bit range 22-29 of the linear address to the base address in the previous PUD entry.

4. the entry in the last level page table (Page Table) is reached summing the offset in the bit range 12-21 of the linear address to the base address in the previous PMD entry.

5. in the last step, the physical address is reached summing the offeset in the bit range 0-11 to the base address of the 4K page in the previous Page Table entry.

On the other hand, the page walk to obtain a physical address using the page of size of 2M is:

1. the entry in the first table (PML4) is reached summing the offset in the bit range 39-47 of the linear address to the base address of the table in the CR3 register.

2. the entry in the second level page table (PUD) is reached summing the offset in the bit range 30-38 of the linear address to the base address in the previous PML4 entry.

3. the entry in the third level page table (PMD) is reached summing the offset in the bit range 22-29 of the linear address to the base address in the previous PUD entry.

4. in the last step, the physical address is reached summing the offeset in the bit range 0-21 to the base address of the 2M page in the previous PMD entry.

1.3.2 Benefits and limitations

The **huge pages support** in the Linux kernel is built on the top of multiple page size support provided by most modern architectures. For example, x86 CPUs allow support for 4K, 2M or 4M page sizes and other sizes can be supported in newer models.

As described in 1.2.4, a TLB is a cache of virtual-to-physical translations. Typically this is a very scarce resource on processor. Operating systems try to make best use of limited number of TLB resources. This optimization is more critical now as bigger and bigger physical memories (several GBs) are more readily available.

Users can use the huge page support in Linux kernel by either using the `mmap` system call or standard System V shared memory system calls (`shmget`, `shmat`).

The performance gain might be brought by the lower number of translations which

in turn results in a lower number of CPU cycles. If page size is larger then the base one, each cache entry includes a larger address space, so the amount of memory translated for each TLB entry is larger too.

Another way to increase the TLB reachable address space could be to add the number of entries, but there are some hardware issues as the energy consumption, the power dissipation and the chip dimension.

A topic relating to the huge pages subsystem is the **fragmentation avoidance mechanism**, because it decrease the probability of failure performing high-order kernel allocations. Like in the case of huge pages, they can not be allocated in absence of a region of contiguous memory. So a greater benefit using huge pages can be obtained when the memory status is less fragmented as possibile.

On the other hand, huge pages are not suitable for a sparse workload pattern, because it involves a small number of references per-huge-page and makes vain the load of larger pages in the TLB cache.

Other issues concerning the use of the huge pages affect not only possible performance problems due to the type of application workload, but even the allocation policies that can be adopted on NUMA systems. As pages with base size, huge pages follow a node-local policy. Because of the likeliness that a node covers data of different threads, there are more cross-node accesses. For this subsystem, a further complexity arises from retrofitting onto the existing code.

Finally, huge pages have to be supported by the architecture in use, so there are possible architectural limitations.

1.3.3 Interfaces

If the kernel is built with `CONFIG_HUGETLBFS` option, the following **interfaces** for the huge pages support are enabled:

- memory mapping: specifying option `MAP_HUGETLB` in the `mmap` system call
- shared memory: enabling `SHM_HUGETLB` flag using `shmget()` function
- hugetlb filesystem: RAM based filesystem for shared or private settings accessing with `mmap()` or `read()`, providing a bare interface to the underlying hardware huge page capabilities that requires application awareness or library support (`libhugetlbfs`)

There are other current types of memory that can be backed with huge pages. For instance, the heap can use the `__morecore` hook to increase heap region backed with huge pages (if possible) and text and data sections, to be relinked to align them. Conversely, the stack memory can not yet support to be backed with huge pages.

1.3.4 HugeTLB filesystem

To use hugetlb support, the Linux kernel needs to be built with both the `CONFIG_HUGETLBFS` and `CONFIG_HUGETLB_PAGE` configuration options. As result, */proc/filesystems* should also show a filesystem of type `hugetlbfs` configured in the kernel.

At mount time, some configuration parameters can be specified, if different from the default ones: `pagesize`, `size` of the maximum amount of memory, `nr_nodes` to limit the number of files and accordingly the number of possible mappings.

The */proc/meminfo* file provides information about the persistent hugetlb pages in the kernel's huge page pool. It displays:

- `HugePages_Total`: size of the pool of huge pages
- `HugePages_Free`: number of huge pages in the pool that are not yet allocated
- `HugePage_Rsvd`: number of huge pages for which a commitment to allocate from the pool has been made, but no allocation has yet been made
  - guarantee that an application will be able to allocate a huge page from the pool of huge pages at fault time
- `HugePages_Surp`: number of huge pages in the pool above the value in */sys/kernel/mm/hugepages/hugepages-2048kB/nr_hugepages*
  - when supported multiple huge page sizes, `nr_hugepages` indicates the number of pre-allocated huge pages of the default size
  - a user with root privileges can dynamically allocate more or free some persistent huge pages by increasing or decreasing the value of `nr_hugepages`
  - the hugetlb subsystem is allowed to try to obtain that number of `surplus` huge pages when the persistent huge page pool is exhausted
  - as these surplus huge pages become unused, they are freed back to the kernel's normal page pool
  - the maximum number of surplus huge pages is controlled by the value in */sys/kernel/mm/hugepages/hugepages-2048kB/nr_overcommit_hugepages*
- `Hugepagesize`: default huge page size in Kb
- `Hugetlb`: total amount of memoty (in Kb) consumed by huge pages of all sizes

The huge page size is needed for generating the proper alignment and size of the arguments to system calls that map huge page regions.

Pages that are used as huge pages are **reserved** inside the kernel and cannot be used for other purposes. Huge pages cannot be swapped out under memory pressure.

Once a number of huge pages have been **pre-allocated** to the kernel huge page pool, a user with appropriate privilege can use either the mmap system call or shared memory system calls to use the huge pages.

The success or failure of huge page allocation depends on the amount of physically contiguous memory that is present in system at the time of the allocation attempt. If the kernel is unable to allocate huge pages from some nodes in a NUMA system, it will attempt to make up the difference by allocating extra pages on other nodes with sufficient available contiguous memory, if any. To check the per node distribution of huge pages in a NUMA system, see at huge pages information in */sys/devices/system/node/node*/meminfo*.

With support for multiple huge page pools at run-time available, much of the huge page userspace interface in `sysfs` is duplicated in */proc/sys/vm*. The `proc` interfaces have been retained for backwards compatibility.

The root huge page control directory in `sysfs` is */sys/kernel/mm/hugepages*, where for each huge page size supported by the running kernel, there is a subdirectory (`hugepages-${size}kB`) to control per-node attributes.

1.3.5 Transparent huge pages

Another implementation for huge pages usage in Linux is the transparent huge pages subsystem. THP provides larger pages to applications with no user-space awareness As discussed in 1.3.2, some workloads performe worse using larger pages, so it was added a per-process option (flag) in the per-process `task_struct` structure. The global configuration about transparent huge pages is available changing the value of

/sys/kernel/mm/transparent_hugepage/enabled.

Transparent huge pages provides ***graceful fallback***: memory management components which do not have transparent hugepage knowledge fall back to breaking huge pmd mapping into table of ptes and, if necessary, split a transparent hugepage. Therefore, on memory pressure these components can continue working on the regular pages or regular pte mappings. If a hugepage allocation fails because of memory fragmentation, regular pages should be gracefully allocated instead and mixed in the same vma without any failure or significant delay and without userland noticing.

Not all THP's can be created at page fault time. `Khugepaged` is a kernel thread that slowly scans all processes' page tables and replaces eligible groups of base pages with THP's. Some `Khugepaged` tuning options is available at */sys/kernel/mm/transparent_hugepage/khugepaged* concern the sleeping period between page table scans (`scan_sleep_millisecs`) and how many page table entries to scan before sleep (`pages_to_scan`).

It is also possible to limit defrag efforts disabling `defrag` option to avoid to perform compaction to allocate huge pages and simply fallback to regular pages unless huge pages are immediately available. THP does not require memory reservation and in turn it uses huge pages whenever possible.

The */proc/meminfo* file provides information about the number of bytes of anonymous huge pages currently in use (`AnonHugePages`). To identify what applications are using anonymous transparent huge pages, it is necessary to read */proc/PID/smaps* and count the `AnonHugePages` fields for each mapping. The number of file transparent huge pages mapped to userspace is available by reading `ShmemPmdMapped` and `ShmemHugePages` fields in */proc/meminfo*. To identify what applications are mapping file transparent huge pages, it is necessary to read */proc/PID/smaps* and count the

`FileHugeMapped` fields for each mapping.

The THP usage statistics are displayed in */proc/vmstat*. Some of the counters of the occurred events are:

- `thp_fault_alloc`: incremented every time a huge page is successfully allocated to handle a page fault
- `thp_collapse_alloc`: incremented by `khugepaged` when it has found a range of pages to collapse into one huge page
- `thp_fault_fallback`: incremented if a page fault fails to allocate a huge page and instead falls back to using base pages
- `thp_collapse_alloc_failed`: incremented if `khugepaged` found a range of pages that should be collapsed into one huge page but failed the allocation
- `thp_file_alloc`: incremented every time a file huge page is successfully allocated
- `thp_file_mapped`: incremented every time a file huge page is mapped into user address space
- `thp_split_page`: incremented every time a huge page is split into base pages, implying splitting all PMD the page mapped with
- `thp_split_page_failed`: incremented if kernel fails to split a huge page
- `thp_deferred_split_page`: incremented when a huge page is put onto split queue because it is going to be split under memory pressure
- `thp_split_pmd`: incremented every time a PMD is splitted into table of PTE, (splitting only page table entry, not huge page itself)

Allocating huge pages may be expensive, because when memory is heavily loaded the system uses **memory compaction** to copy data around memory to free a huge page for use. There are some other counters in */proc/vmstat* to help monitor this overhead:

- `compact_stall`: incremented every time a process stalls to run memory compaction so that a huge page is free for use
- `compact_success`: incremented if the system compacted memory and freed a huge page for use
- `compact_fail`: incremented if the system tries to compact memory but failing
- `compact_pages_moved`: incremented each time a page is moved implying that the system is copying a lot of data to satisfy the huge page allocation
- `compact_pagemigrate_failed`: incremented when the underlying mechanism for moving a page failed
- `compact_blocks_moved`: incremented each time memory compaction examines a huge page aligned range of pages

Chapter 2

Linux networking

2.1 The packet socket interface

2.1.1 The raw socket mode

A **raw socket** is an internet socket that allows direct sending and receiving of Internet Protocol packets without any protocol-specific transport layer formatting, hence capturing link-level information. It is created specifying the option `SOCK_RAW` during the socket inizialization.

A raw socket is used to send and receive **low level** messages (i.e. ICMP and IGMP messages) or new user-space protocol implementation. But this is not the only application of this socket.

On of the main usages of this type of socket is introduced into sniffing activities, where packet capture (decribed below) and promiscuous mode (receiving all packet incoming on a shared medium) are crucial. Indeed, the raw sockets are used by network monitoring tools as *Wireshark*, *tcpdump*, *Suricata*.

2.1.2 Packet capture

The **packet capture** is an OS utility that allows the user to intercept any packet that is seen by the network device in the raw form.

When a network card picks up a packet from the network, the OS has to hand off it to determine what type of packet it is, striping off the packet's header looking to the next layers. Then, the packet payload is handed over to the application that the packet is sent for.

Therefore, the packet capture is a low level facility to grab the incoming packet in its entirety, regardless of which application is being sent to.

The `pcap` library provides an higher level, implementation-independent access to the underlying packet capture facility. It covers only a small set of `AF_PACKET` features, but it makes programs portable because it is a standard among the different operating systems. More about this library will be discussed in 4.2.

2.2 The memory map approach

We refer to the **packet socket interface** as a low level access to the network interface, used for capturing or transmitting network traffic as well for implementing any other service that needs raw access to a network interface.

In order to improve efficiency of packet raw reception and transmission in the Linux kernel, the `PACKET_MMAP` facility is made available with socket interface on kernel from 2.6 version. This patch makes transmission process a zero-copy mechanism and increases RX bandwidth of raw socket, avoiding to incur in the copying of data that dominates the performance-sensitive networking applications.

If `PACKET_MMAP` is not enabled, the capture process is very inefficient. It uses very

limited buffers to collect incoming packets and it requires one system call to capture each packet (via `recvmsg`). It requires two system calls to get packet's timestamp (via `ioctl`).

In the other hand, `PACKET_MMAP` is very **efficient** if it is enabled. It provides a size configurable circular buffer mapped in user space that can be used to either send or receive packets. This way reading packets just needs to wait for them, most of the time there is no need to issue a single system call. Concerning transmission, multiple packets can be sent through one system call to get the highest bandwidth. Using a shared buffer between the kernel and the user has also the benefit of minimizing packet copies.

But `PACKET_MMAP` is not the only way to improve performance of the capture or transmission process. Concerning high-speed capture processes, it should be checked if the device driver of the network interface card supports an interrupt load mitigation as the **NAPI** (*New API*) extension. It has to be checked if it is enabled, too. But today NAPI is always enabled, and it is simply not conceivable to drive a high data rate card without NAPI enabled.

2.3 User level code

2.3.1 Packet socket settings

Some packet socket options that may be configured by calling `setsockopt` with level `SOL_PACKET` are:

- `PACKET_RX_RING`: to create a memory-mapped ring buffer for asynchronous packet reception; the packet socket reserves a contiguous region of application address space, lays it out into an array of packet slots and copies packets into subsequent slots. Details about the ring buffer allocation are described in 2.4.3

- `PACKET_TX_RING`: similarly to `PACKET_RX_RING`, to create a memory-mapped ring buffer, but for packet transmission
- `PACKET_TIMESTAMP`: to generate a timestamp when the packet is copied into the ring and the packet receive ring always stores it in the metadata header
- `PACKET_STATISTICS`: to count dropped and total packets
- `PACKET_VERSION`: to choose the variant based on which the ring has to be created; the three variants are described in 2.3.4

The `AF_PACKET` domain allows to use one socket for capture and transmission. The reception (RX) and transmission (TX) buffers is allocated with one call to `setsockopt`.

2.3.2 Circular buffer

The **circular buffer** (ring) for each socket is an area of unswappable memory defined as the structure `tpacket_req3` shown in Figure 2.1:

```
struct tpacket_req3 {
        unsigned int    tp_block_size;        /* minimal size of contiguous block */
        unsigned int    tp_block_nr;          /* total number of blocks */
        unsigned int    tp_frame_size;        /* size of frame */
        unsigned int    tp_frame_nr;          /* total number of frames */
        unsigned int    tp_retire_blk_tov;    /* timeout in msecs */
        unsigned int    tp_sizeof_priv;       /* offset to private data area */
        unsigned int    tp_feature_req_word; /* additional functionalities */
};
```

Figure 2.1: The `tpacket_req3` struct definition in *include/uapi/linux/if_packet.h*

User must provide these parameters passing them to the `setsockopt()` call via a pointer to the struct when configuring the `PACKET_RX_RING` option.

A ring buffer is a memory region used to store datagram data. Each datagram is stored in a separate **frame**. Frames are grouped in **blocks**. A ring buffer is a set of

`tp_block_nr` blocks. A frame can not be spawned across more blocks, so the frame size must fit in a block.

The value of `tp_frame_nr` is a redundant value, because it can be obtained by:

$$tp\_frame\_nr = frames\_per\_block \times tp\_block\_nr \tag{2.3.1}$$

where the number of frames per blocks is given by 2.3.2 and it is multiplied by the total number of blocks.

$$frames\_per\_block = \frac{tp\_block\_size}{tp\_frame\_size} \tag{2.3.2}$$

The timeout in `tp_retire_blk_tov` indicates the amount of time after which a block is retired, even if it is not fully filled with data.

`tp_sizeof_priv` is the size of per-block private area. This area can be used by a user to store arbitrary information associated with each block.

Finally, `tp_feature_req_word` includes a set of flags to enable some additional functionality.

Each block has an associated **header**, which is stored at the very beginning of the memory area allocated for the block. The block header struct is called `tpacket_block_desc` and has a `block_status` field, which indicates whether the block is currently being used by the kernel or available to the user. The structure definition is shown in Figure 2.2 and the relating scheme in Figure 2.8.

The usual workflow is that the kernel stores packets into a block until it is full and then sets `block_status` to `TP_STATUS_USER`. The user then reads required data from the block and releases it back to the kernel by setting `block_status` to `TP_STATUS_KERNEL`.

Each frame includes not only the link level frame, but has also an associated header defined in the the `tpacket3_hdr` struct. This header holds the datagram's

```
union tpacket_bd_header_u {
        struct tpacket_hdr_v1 bh1;
};

struct tpacket_block_desc {
        __u32 version;
        __u32 offset_to_priv;
        union tpacket_bd_header_u hdr;
};

struct tpacket_hdr_v1 {
        __u32                block_status;
        __u32                num_pkts;
        __u32                offset_to_first_pkt;
        __u32                blk_len;
        __aligned_u64        seq_num;
        struct tpacket_bd_ts ts_first_pkt, ts_last_pkt;
};
```

Figure 2.2: Structs definitions in *include/uapi/linux/if_packet.h*

```
struct tpacket3_hdr {
        __u32            tp_next_offset;
        __u32            tp_sec;
        ...
        __u32            tp_len;
        __u32            tp_status;
        __u16            tp_mac;
        __u16            tp_net;
        union {
                struct tpacket_hdr_variant1 hv1;
        };
        __u8             tp_padding[8];
};
```

Figure 2.3: `TPACKET_V3` header struct definition *include/uapi/linux/if_packet.h*

meta information like timestamp. The `tp_next_offset` field points to the next frame within the same block. The structure is shown in Figure 2.3.

The status field `tp_status` is zero when the frame is ready to be used by the kernel (`TP_STATUS_KERNEL` for reception or `TP_STATUS_AVAILABLE` for transmission) and it corresponds to the initialization value. If it is not zero, there is a frame that the user can read or send based on the specified status flags. Once the packet is read or transferred by the user, the field must be set to zero to allow the kernel to use again that frame.

When a block is fully filled with data, that is when a new packet does not fit into the remaining space), it is closed and released to user space.

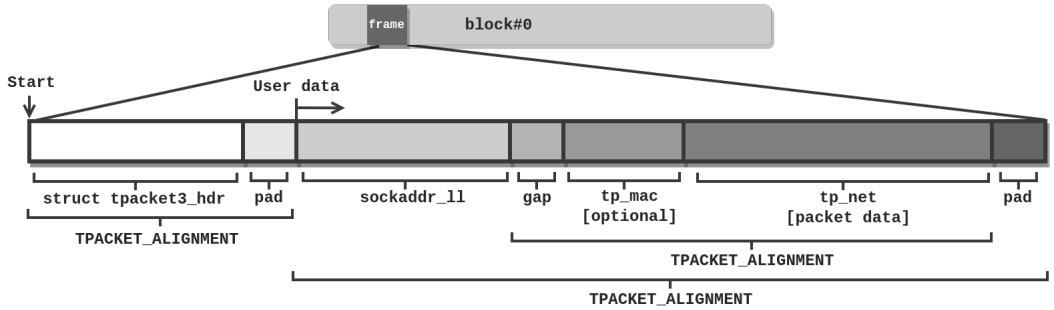


Figure 2.4: The TPACKET_V3 frame structure.

A block is retired by the kernel also when the timeout expires, as controlled by the `tp_retire_blk_tov` parameter. The timer is due to the user need to read packets as soon as possible.

As shown in Figure 2.4, the frame structure includes both in capture and transmission process:

- Start of the frame must be aligned to `TPACKET_ALIGNMENT` (=16)
- struct `tpacket_hdr`: frame status
- pad to `TPACKET_ALIGNMENT=16`
- struct `sockaddr_ll`
- Gap, chosen so that packet data (Start+tp_net) aligns to `TPACKET_ALIGNMENT`=16
- Start+tp_mac: optional MAC header
- Start+tp_net: Packet data, aligned to `TPACKET_ALIGNMENT`=16
- Pad to align to `TPACKET_ALIGNMENT`=16

The user should put data at

$$frame\_base + TPACKET3\_HDRLEN - sizeof(struct\ sockaddr\_ll)$$

So, using whatever socket mode (`SOCK_RAW`, `SOCK_DGRAM`) the beginning of the user data is at

$$frame\_base + TPACKET\_ALIGN(sizeof(struct\ tpacket\_hdr))$$

The `sockaddr_ll` structure contains the link-level informations about the device-independent physical-layer address. If it is created a `SOCK_RAW` socket, at receiving time the address is parsed and is passed in the structure. At transmission time, the user supplied buffer should contain the physical header.

2.3.3 The capture and transmission processes

The use of `PACKET_MMAP` to improve the **capture process**, involves the following system calls:

1. setup:
   - `socket()`: creation of the capture socket
   - `setsockopt()`: circular buffer allocation with option `PACKET_RX_RING`
   - `mmap()`: mapping of the allocated buffer to the user process
2. capture:
   - `poll()`: active waiting for incoming packets
3. shutdown:
   - `close()`: socket destruction and resources deallocation

It is possible to use one socket for capture and transmission, mapping both RX and TX buffer rings with a single `mmap()` call. Even if the buffer consists of several

physically discontiguous blocks of memory, they are **contiguous** to the user space, so just one `mmap` call is needed.

The **transmission process** is similar to the capture one:

1. setup:
    - `socket()`: creation of the transmission socket
    - `setsockopt()`: circular buffer allocation with option `PACKET_TX_RING`
    - `bind()`: specifying an address in the `sll_ifindex` parameter in `struct sockaddr_ll` to bind the socket to a network interface to know the header size of frames used in the circular buffer
    - `mmap()`: mapping of the allocated buffer to the user process
2. transmission:
    - `poll()`: (*optional*) active waiting for free packets
    - `send()`: send all *ready* packets in the ring following a blocking or a non-blocking policy
3. shutdown:
    - `close()`: socket destruction and resources deallocation

2.3.4 The `TPACKET` evolution

The `tpacket` **variant** setting (`PACKET_VERSION`) in the `setsockopt` call can be `TPACKET_V1` (default), `TPACKET_V2` or `TPACKET_V3`.

The first version is the default one if not otherwise specified. `RX_RING` and `TX_RING` are available.

The changes from the `TPACKET_V1` to the `TPACKET_V2` are the following ones:

```
struct tpacket2_hdr {
        ...
        __u16           tp_vlan_tci;
        __u16           tp_vlan_tpid;
}
```

Figure 2.5: Added fields for `TPACKET_V2` header definition in *include/uapi/linux/if_packet.h*

- made 64 bit clean due to unsigned long usage in `TPACKET_V1` structures, thus this also works on 64 bit kernel with 32 bit userspace and the like
- Timestamp resolution in nanoseconds instead of microseconds
- `RX_RING` and `TX_RING` available
- VLAN metadata information available for packets: added two fields in the `tpacket2_hdr` structure indicating VLAN TPID or TCI values settings

The new fields in the second variant of the structure are shown in Figure 2.5.

The further changes that has been introduced with `TPACKET_V3` are:

- blocks can be configured with static or non-static frame-size
- update memory mapping to enable variable size
- read/poll at block level (instead of at packet level)
- added poll timeout to avoid indefinite user-space wait on idle links
- RX Hash data available in user space

For now only `RX_RING` is available, the transmission buffer is not yet implemented.

These changes brought benefits in the CPU-usage reduction, increased packet capture rate and packet density, and port aggregation analysis. Moreover, the non-static frame size configuration allows to capture the entire packet payload.

```
struct packet_sock {
     struct sock                sk;
     ...
     union tpacket_stats_u      stats;
     struct packet_ring_buffer  rx_ring;
     struct packet_ring_buffer  tx_ring;
     ...
     struct mutex               pg_vec_lock;
     int                        ifindex;  /* bound device */
     enum tpacket_versions      tp_version;
     int                        (*xmit)(struct sk_buff *skb);
};
```

Figure 2.6: Packet socket struct definition in *net/packet/internal.h*

The switching among versions from 1 to 3 is possibile using, respectively, the `tpacket_hdr` or the `tpacket2_hdr` or the `tpacket3_hdr` struct and setting the proper protocol version number.

2.4 Kernel level code

2.4.1 `AF_PACKET` sockets' implementation

Whenever a packet socket is created, an associated `packet_sock` structure is allocated in the kernel. It is defined in *net/packet/internal.h* as shown in Figure 2.6.

The `tp_version` field in this struct holds the ring buffer version, which in our case is set to `TPACKET_V3` by a `PACKET_VERSION setsockopt` call.

The `pg_vec_lock` mutex protects concurrent accesses to the socket structure to update the ring buffer structure values.

The `stats` field maintains counters of the number of packets received (`tp_packets`) and of the dropped packets (`tp_drops`). For the `tpacket_version` 3 is available even the counter of the times the queue has been freezed (`tp_freeze_q_cnt`).

The `rx_ring` and `tx_ring` fields describe the receive and transmit ring buffers in case they are created via `PACKET_RX_RING` and `PACKET_TX_RING` setsockopt calls. These two fields have type `packet_ring_buffer` which is defined as shown in Figure 2.7.

```
struct packet_ring_buffer {
        struct pgv                *pgv;
        ...
        struct tpacket_kbdq_core  prb_bdqc;
};

struct pgv {
        char *buffer;
};
```

Figure 2.7: Ring buffer and block pointers structures definitions in *net/packet/internal.h*

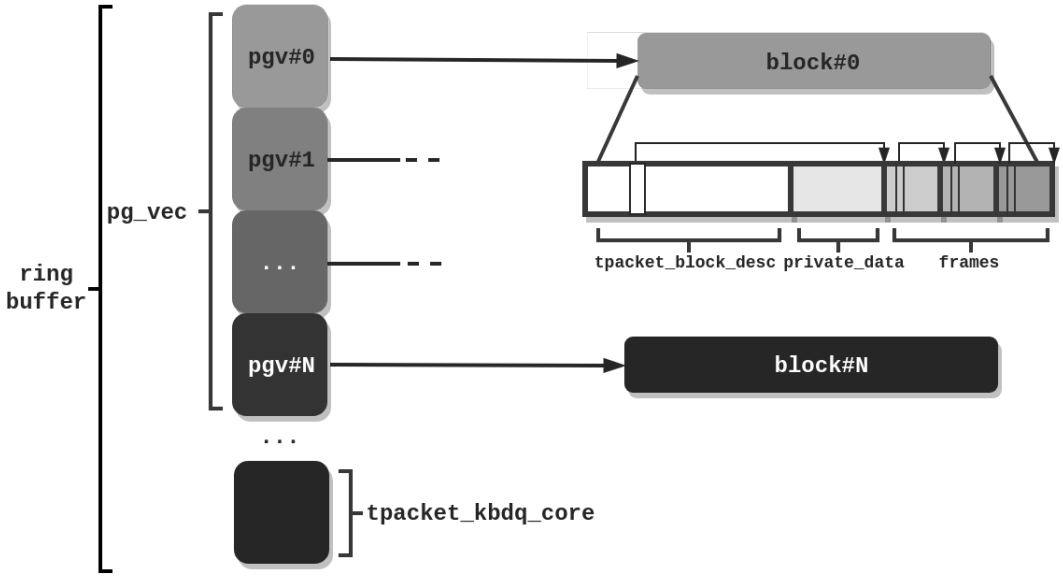


Figure 2.8: The ring buffer and the blocks structures.

The `pg_vec` is a pointer to an array of `struct pgv`, each of which holds a reference to a block.

In Figure 2.8 is shown the relationship between the ring buffer and the blocks it points to and the single block structure. Details about blocks structure and allocation is described in 2.4.3.

The `prb_bdqc` field is of type `tpacket_kbdq_core` that is the kernel block descriptor queue and its fields describe the current state of the ring buffer (Figure 2.9).

The `blk_sizeof_priv` fields contains the size of the per-block private area. The `nxt_offset` field points inside the currently active block and shows where the next

```
struct tpacket_kbdq_core {
        ...
        unsigned short     blk_sizeof_priv;
        ...
        char               *nxt_offset;
        ...
        struct timer_list  retire_blk_timer;
};
```

Figure 2.9: Block status struct definition in *net/packet/internal.h*

```
struct timer_list {
        ...
        struct hlist_node entry;
        unsigned long     expires;
        void              (*function)(unsigned long);
        unsigned long     data;
        ...
};
```

Figure 2.10: Timer struct definition in *include/linux/timer.h*

packet should be saved. The `retire_blk_timer` field has type `timer_list` (Figure 2.10) and describes the timer which retires current block on timeout.

2.4.2 Load balancing

By the kernel side, **load balancing** is achieved using multiple algorithms:

- **Round-Robin**
- **Per-flow**: packets of a given flow sent to the same socket
- **Per-CPU**: packets treated by the same CPU sent to the same socket

Furthermore, by the network driver side, the Network Interface Controller (NIC) achieves load balancing through:

- **Receive-Side Scaling (RSS)**, using a multi-queue receive to distribute network receive processing across several queue programmed by user or per-flow load balanced
- **hash load balancing** and set the CPU affinity to keep the cache line

2.4.3 Ring buffer implementation

The main code about `AF_PACKET` network domain is included in *net/packet/af_packet.c*.

The kernel uses the `packet_setsockopt()` function to handle setting socket options for packet sockets. It implements the protocol operation corresponding to the `setsockopt` call.

Via the `PACKET_RX_RING` (or the `PACKET_TX_RING`) socket option, it is created a receive (or a transmit) **ring buffer**. When packets are coming through a network interface, kernel puts them in the ring buffer. The ring buffer is a dynamically allocated structure implemented as an array of pointers to each block. The size of the array limits the number of blocks that can be allocated:

$$block\_num\_limit = \frac{size\_max}{pointer\_size}$$

If user specifies one option between `PACKET_RX_RING` and `PACKET_TX_RING`, it is called `packet_set_ring()` for the buffer allocation.

First, `packet_set_ring` performs sanity checks on the ring buffer parameters provided by the user, among which there are the following ones:

- `tp_block_size` must be more than zero
- `tp_block_size` must be aligned to the page size
  - it should be chosen to be a power of two to avoid waste of memory
- `tp_frame_size` must be more or equal than the minimum size
  - minimum size is the sum of the `tpacket` header lenght and the reserved bytes specified in the `tp_hdrlen` and `tp_reserve` fields in the `packet_socket` struct

```
static struct pgv *alloc_pg_vec(struct tpacket_req *req, int order) {
        ...

        pg_vec = kcalloc(block_nr, sizeof(struct pgv), GFP_KERNEL);
        ...

        for (i = 0; i < block_nr; i++) {
                pg_vec[i].buffer = alloc_one_pg_vec_page(order);
                ..
        }
        ...
}
```

Figure 2.11: Ring buffer allocation in *net/packet/af_packet.c*

- `tp_frame_size` must be a multiple of `TPACKET_ALIGNMENT`
- in the case of `TPACKET_V3` (our case), the sum of the size of the private data of the request and the `tp_frame_size` must be less than `tp_block_size`
- frames per block computed as in 2.3.1 must be more than zero
- `tp_block_size` must be less than the maximum size
  - maximum block size is given by the ratio between the integer maximum value and the number of blocks (`tp_block_nr`)
- `tp_frame_nr` must be equal to the number of frames per blocks multiplied by the `tp_block_nr`

Then, it allocates the ring buffer calling `alloc_pg_vec()`, defined as shown in Figure 2.11.

The order of the allocation request is obtained from the size of memory to be allocated using `get_order()`, defined in *include/asm-generic/getorder.h*. The order returned is used to find the smallest allocation granule required to hold a block of memory of the specified size.

First, the function allocates dynamically the ring buffer (`pg_vec`) using `kcalloc`. The slab allocator is the responsible for this allocation. It maintains a pool of pre-

```
static char *alloc_one_pg_vec_page(unsigned long order)
{
        char *buffer;
        gfp_t gfp_flags = GFP_KERNEL | __GFP_COMP |
                          __GFP_ZERO | __GFP_NOWARN | __GFP_NORETRY;

        buffer = (char *) __get_free_pages(gfp_flags, order);
        if (buffer)
                return buffer;

        ...
        /* __get_free_pages failed, fall back to vmalloc */
        buffer = vzalloc(array_size((1 << order), PAGE_SIZE));
        if (buffer)
                return buffer;

        /* vmalloc failed, lets dig into swap here */
        gfp_flags &= ~__GFP_NORETRY;
        buffer = (char *) __get_free_pages(gfp_flags, order);
        if (buffer)
                return buffer;

        /* complete and utter failure */
        return NULL;
}
```

Figure 2.12: Block allocation in *net/packet/af_packet.c*

determined sizes of physically contiguous memory, imposing the maximum memory that `kcalloc` can allocate. This limit can be checked in the *size-bytes* entry of `/proc/slabinfo`. For example, in a x86 architecture the limit is 4194304 bytes corresponding to 1024 pages. In 64 bit architecture, pointers are 4 bytes long, so the total number of pointers to blocks is

$$\frac{4194304}{4} = 1048576 \text{ blocks.}$$

Then, for each one of the blocks (`tp_block_nr`), it is allocated the corresponding memory area calling the function `alloc_one_pg_vec()` (Figure 2.12). Hence the blocks are allocated separately not as one contiguous memory region.

Finally, `packet_set_ring()` calls `init_prb_bdqc()`, which performs some additional steps to set up a `TPACKET_V3` receive ring buffer specifically (block transmit is not supported yet), as shown in Figure 2.13.

The `init_prb_bdqc()` function copies provided ring buffer parameters to the

```
case TPACKET_V3:
        if (!tx_ring) {
                init_prb_bdqc(po, rb, pg_vec, req_u);
        }
        ...
```

Figure 2.13: From `packet_set_ring` in *net/packet/af_packet.c*

`prb_bdqc` field of the ring buffer struct (discussed in 2.4.1), calculates some other parameters based on them, sets up the block retire timer, and calls `prb_open_block()` to initialize the first block.

One of the things that the `prb_open_block()` function does is it sets the `nxt_offset` field of the `tpacket_kbdq_core` struct to point right after the per-block private area.

Chapter 3

Problem analysis

3.1 Feasibility study

3.1.1 Limits and goals

The work related to this master thesis was motivated by a practical goal: to analyze real network traffic in order to collect all the relevant data that are extracted from the incoming packets.

The existing Network Interface Controllers are going to increase the level of the supported data rates up to 160 Gigabit/s. In the most common configuration the **10 Gigabit Ethernet** are available on motherboards of high performance computers.

In parallel with the increment of the data rate of the incoming network packets a new need is born: the capability to face with a larger amount of data to manage at arrival time.

Packet loss is undesirable, or at least to minimize, because among those lost packets could be included some relevant data that can no more be taken into account for the analysis. In some applications packet loss has to be avoided at all costs, while in other applications packet loss, up to some point, can be acceptable.

In this work the assumption is that the user-level monitoring application accepts a moderate packet loss. In order to study the trend of the packet loss percentage of the application, tests will be performed varying the load on the buffer. The overloading of the buffer is obtained with a simulated processing time on a packet.

Obviously, a countermeasure in order to decrease the packet loss is to set a **larger buffer size** especially to cope with the peaks of traffic. But when large memory areas are allocated, there is the operating system memory management to consider for performance advantages and disadvantages evaluation referring to this use.

Althought enlarging the size of packet buffers might help in reducing packet loss, it could also lead to performance impairment. In fact, larger memory areas brings to a larger set of memory addresses to translate. And also the time spent translating addresses affects the overall application performance.

As described in section 1.2.4, the amount of time spent to translate addresses can be reduced minimizing the TLB miss rate. And the TLB miss rate, in turn, can be reduced maximizing the TLB reachable address range using **huge pages** to cover wider range of addresses. The number of entries is kept unchanged, but there is the *illusion* to have a larger TLB. Without changing the table size inserting new entries, every entry now refers to a range that includes more addresses, allowing to be equivalent to a parallel system with more entries resolving the same number of addresses. Using huge page table makes more likely to count a TLB hit.

We chose to use the Linux operating system for the study of a solution that the kernel can incorporate. In this way the Linux kernel could offer an optimization of the high-performance packet capture application.

Linux is commonly used in this scenario, because it allows to patch its code in order to approach the implementation of a software that is useful for the wanted

application.

There are many frameworks that extend and modify the Linux kernel for high-speed packet capture. Some examples are: **PFQ**, **Intel DPDK**, **PF_RING ZC**.

In particular, the **PFQ (Packet Fair Queueing)** was a source of inspiration for this thesis. It is a project started by an italian researcher, Nicola Bonelli. PFQ is a high-performance framework for packet capture that have the capacity to reach the link speed also when the rate is of the order of 10 Gb/s.

It is a rich module that implements many features. Some of them were incorporated in the Linux kernel. An implementation choice in the PFQ code it is interesting for our work: the buffer for packet reception/transmission shared between user and kernel is a memory area backed with huge pages. The structure of the buffer is simple. There are two buffers of contiguous memory that receive alternately the incoming packets.

Currently, in the Linux kernel the `AF_PACKET` domain used for high-speed packet capture are not required huge pages for the buffer allocation.

The question is to evaluate if it is possible to use huge page also in this domain and what is the impact on the performance.

The target we want to achieve with this experimental work is to change the management of the kernel side allocation of the ring buffer for reception of a high rate of incoming packets.

3.1.2 Packet reception

Whenever a new packet is received, the kernel is supposed to save it into the ring buffer. The reception is managed with the function `tpacket_rcv` that, after the `sk->rx_queue.lock` is taken, calls `packet_current_rx_frame`. For `TPACKET_V3`, the

key function here is `__packet_lookup_frame_in_block()`. For previous `tpacket` versions, the corresponding function is `packet_lookup_frame()`.

The `__packet_lookup_frame_in_block()` function checks whether the currently active block has enough space for the incoming packet. If space is available, it saves the packet to the current block and returns. In the other hand, it closes the current block with `__prb_retire_current_block()` call and then dispatches the next block with `__prb_dispatch_next_block()` call.

If the dispatch is successful, it saves the packet there with `prb_fill_curr_block` call. Otherwise, the next block is currently blocked by the user and the queue of the block is frozen with `prb_freeze_queue()` call. The incoming packet will get dropped. Then, if link goes idle, the timer refreshed with the last `open_block()` call will expire and the first block can be re-opened in the near future. The timer logic is to refresh the timer only when a block it is opened. By doing this it is avoided to waste cycles refreshing the timer on packet-by-packet basis.

Instead, if the link is busy, keeping on receiving packets, `__packet_lookup_frame_in_block()` will check if the block-0 is free and can be re-used.

The packet reception code is shown in Figure 3.1.

3.1.3 Block allocation

Each block is a memory region allocated through the `alloc_one_pg_vec()` function described in 2.4.3 that makes a call to `__get_free_pages()` function, defined in *mm/page_alloc.c* as:

unsigned long __get_free_pages(gfp_t gfp_mask, unsigned int order);

where the `gfp_mask` argument specifies the combination of the required types of allo-

```
static void *__packet_lookup_frame_in_block(
                                struct packet_sock *po,
                                struct sk_buff *skb,
                                int status,
                                unsigned int len)
{
        ...
        curr = pkc->nxt_offset;
        pkc->skb = skb;
        end = (char *)pbd + pkc->kblk_size;
        if (curr+TOTAL_PKT_LEN_INCL_ALIGN(len) < end) {
                prb_fill_curr_block(curr, pkc, pbd, len);
                return (void *)curr;
        }
        prb_retire_current_block(pkc, po, 0);
        curr = (char *)prb_dispatch_next_block(pkc, po);
        if (curr) {
                pbd = GET_CURR_PBLOCK_DESC_FROM_CORE(pkc);
                prb_fill_curr_block(curr, pkc, pbd, len);
                return (void *)curr;
        }
        return NULL;
}
```

Figure 3.1: Packet reception in *net/packet/af_packet.c*.

cation. All flags are described in *include/linux/gfp.h*. The `order` of a page allocation is the base 2 logarithm.

The allocated size is computed as:

$$pagesize * 2^{order}$$

where the order argument has to be less than the `MAX_ORDER` threshold, defined in *include/linux/mmzone.h*. In a v4.20 kernel `MAX_ORDER` is 11. The `PAGE_SIZE` used in the function is defined in *include/asm-generic/page.h*, depending on the architecture specific value of `PAGE_SHIFT`.

For example, in a x86 architecture, the `PAGE_SHIFT` is defined in *arch/x86/include/asm/page_types.h* and has the value of 12. Then, the generic `PAGE_SIZE` defined as

$$(1UL << PAGE\_SHIFT)$$

is equal to

$$1 \times 2^{PAGE\_SHIFT}$$

byes. So the page size is 4 KB.

3.2 Memory allocation using huge pages

3.2.1 Pre-allocated Huge pages reservation

Huge pages as described in 1.3.4 are typically preallocated for application use. These huge pages are instantiated in a task's address space at page fault time if the VMA indicates huge pages are to be used. If no huge page exists at page fault time, the task is sent a SIGBUS and often dies. After huge page support was added, it was determined that it would be better to detect a shortage of huge pages at `mmap()` time: if there were not enough huge pages to cover the mapping, the `mmap()` would fail. First. this was done with a simple check in the code at `mmap()` time to determine if there were enough free huge pages to cover the mapping. Then, the idea was to *reserve* huge pages at `mmap()` time to ensure that huge pages would be available for page faults in that mapping. The description below attempts to describe how huge page reserve processing is done in the v4.20 kernel.

In the Linux kernel source code (in file *mm/hugetlb.c*) the function `hugetlb_init` allocates multiple physically contiguous pages of normal page size to form clusters of pages which can be used for large page sizes.

The number of pages which are allocated like this depends on the value of `max_huge_pages` variable. This number can be passed on as a kernel command line option by using the `hugepages` parameter. The large page size allocated depends on the macro `HUGETLB_PAGE_ORDER` which in turn depends on `HPAGE_SHIFT` macro.

For example, in a x86 architecture, the `HPAGE_SHIFT` macro is defined in *arch/x86/include/asm/page_types.h* and has the value of the marco `PMD_SHIFT`, that determines the size of the area a middle-level page table can map and has the value

of 21 (as defined in *arch/x86/include/asm/pgtable_64_types.h*). Then, the default `HPAGE_SIZE` defined as

$$(1UL << HPAGE\_SHIFT)$$

is equal to

$$2^{HPAGE\_SHIFT}$$

bytes. Threfore the huge page size is 2 MB.

The pages allocated as mentioned previously are enqueued into `hugepage_freelists` for the respective node, where the page is allocated from, by the function `enqueue_huge_page`. Each memory node (in case of NUMA) will have one `hugepage_freelists`.

A HugeTLB memory area is created via the `shmget` or `mmap` system calls specifying the relating large page flags. Then the file operations corresponding to this file structure will be assigned to `hugetlbfs_file_operations`. The large pages gets reserved by the function `hugetlb_reserve_pages` which will increment the reserve pages count (`resv_huge_pages`) which shows up as `HugePages_Rsvd` in the */proc/meminfo*. When one of the system calls is made, sanity checks are done by using `hugetlb_get_unmapped_area` function.

When a page fault occurs, the virtual memory area which corresponds to the address is found. The VMA which corresponds to a hugetlb shared memory location will have `vma->vm_flags` set as `VM_HUGETLB`, and is detected by calling `is_vm_hugetlb_page`. When a hugetlb VMA is found the `hugetlb_fault` function is called. This procedure sets up large page flag in the page directory entry then allocates a huge page based on a copy-on-write logic from the global pool of large pages initialized previously. The large page size itself is set in the hardware by setting the `_PAGE_PSE` flag in the pgd.

3.2.2 Transparent Huge Page allocation

If a kernel module needs to allocate big chuncks of memory is usually preferred to request whole pages [3]. To allocate (and respectively free) pages the following functions are available:

- `get_zeroed_page()`: returns a new page filled with zeros
- `__get_free_page()`: returns a new page
  - `free_page()`: de-allocates a page
- `__get_free_pages()`: returns a pointer to the first byte of several physically contiguous pages
  - `free_pages()`: de-allocatesa sequence of pages

Each function requires to specify the allocation flags among those defined in *include/linux/gfp.h* to characterize the type of allocation. The following are some of the available flags:

- `GFP_KERNEL`: typical internal kernel allocation request for `ZONE_NORMAL` or a lower zone for direct access
- `__GFP_COMP`: address compound page metadata
- `__GFP_ZERO`: return a zeroed page on success
- `__GFP_NOWARN`: to suppress allocation failure reports
- `__GFP_NORETRY`: attempt of direct memory request
  - to handle the failure that is quite likely to happen under heavy memory pressure

These ones just described are the flags that are currently set during the single block allocation (Figure 2.12).

Other flags that are which we are particularly interested in are those ones concerning the transparent huge pages allocation:

- `GFP_TRANSHUGE`: direct reclaim used by `khugepaged` thread
- `GFP_TRANSHUGE_LIGHT`: no attempt at all to reclaim or compact memory

Chapter 4

The designed solution

4.1 Description

4.1.1 New kernel configuration entry

To confine the change made in the kernel code to implement the new features, we introduced a new kernel configuration option named `CONFIG_PACKET_HUGE`.

Adding a new option is equivalent to create a new **Kconfig** option. This operation requires updates of one of the Kconfig files within the kernel source code.

We insert our new configuration entry in *net/packet/Kconfig* keeping bearing with other configurations of the `AF_PACKET` domain.

Following the kernel documentation ([20]), we define the attributes of the new entry:

- dependencies on other entries
  - obvious dependency on `PACKET` to use the Packet Protocol
  - dependency on `HUGETLBFS` to allow the support for the hugetlb file system to be compatible with the hugetlb block allocation option

  - dependency on `TRANSPARENT_HUGEPAGE` to allow the support for transparent huge pages to be compatible with the THP block allocation option

- type definition: set to `tristate` to allow to build or not to build this feature or to build it as a kernel module

- default value: set to `n` as recommended for new options to avoid to weight down the build

- help text

In the Figure 4.1 is shown the description of the entry.

```
config PACKET_HUGE
        tristate "Packet socket buffers backed with THP and hugetlbfs"
        depends on PACKET && TRANSPARENT_HUGEPAGE && HUGETLBFS
        default n
        ---help---
                Support for AF_PACKET socket buffers backed with huge pages,
                both using the HugeTLB filesystem and the Transparent Huge Pages.
                If unsure, say N.
```

Figure 4.1: New kernel configuration option in *net/packet/Kconfig*.

4.1.2 Interface of the new features

In order to enable the use of the new policies for the block allocation, we decided to introduce the opportunity of selecting a certain option by user space.

After creating the packet socket, the user chooses which settings has to be applied to the socket just created. This is done making calls to `setsockopt` to set, for example, the variant of `tpacket` or the transmission and reception mode.

Concerning the setting of the capture process, the allocation of a ring buffer for the socket is performed as seen in Figure 2.12.

The current implementation provides only one option, the `PACKET_RX_RING` one,

that includes more allocation policies. These policies are tempted in order of decreasing efficiency.

The first allocation attempt is the preferred one: it tries to allocate contiguous memory directly by means of `__get_free_pages` (3.2.2) without resorting to costly actions.

On memory pressure this attempt can easily fail. The fallback allocation attempt is made calling `vmalloc` function. On success we obtained an amount of memory only virtually contiguous.

Finally, the failure of even the second attempt means that the memory of the system is heavily loaded and the allocation operation falls back to allocate contiguous memory with `__get_free_pages` repeating the allocation requests indefinitely, even recurring to the swap area (when it is available). This is the most indesiderable case, but it is still preferred rather than failing the block allocation at all.

On the other hand, in our implementation we introduced different options to let the user force a specific allocation policy relating to a socket buffer. This is done for the only purposes of testing each policy and comparing their performance.

Instead, at production time it can be more reasonable to follow the original model of an allocation mode based on the first mechanism in order of preference than does not fail.

In a new file *net/packet/if_huge_packet.h*, the following new options are defined:

- `PACKET_HUGETLB_RING`: block allocation backed with the **HugeTLB** file system
- `PACKET_TRANSHUGE_RING`: block allocation backed with the **Transparent Huge Pages**
- `PACKET_GFPNORETRY_RING`: block allocation with direct memory reclaim for

physically contiguous area

- `PACKET_VZALLOC_RING`: block allocation of only **virtually contiguous** memory area
- `PACKET_GFPRETRY_RING`: block allocation of **physically contiguous** memory, also resorting to the swap area (if available)

4.2 Integration with the pcap library

4.2.1 Reasons

The **pcap** library [2] is a publicly available packet capture library. This library provides a high level interface to packet capture services with an implementation-independent access to the facility. It covers only a small set of `AF_PACKET` features, but it makes programs portable because it is a standard among the different operating systems.

Furthermore, in terms of popularity the `pcap` library is used in the most common applications for traffic analysis like **Wireshark**, **tcpdump** (Figure 4.2). Therefore, we faced with the integration of the library with the kernel patch to be sure to keep compatibility with the applications mentioned above.

4.2.2 Lack of compatibility

The patch described in this chapter allows to choose among different option enabling one of the allocation policies for the reception buffer setup.

The points into the source code of the `pcap` library that are involved in order to support the new allocation options regard:

- the modality of getting the **page size** value

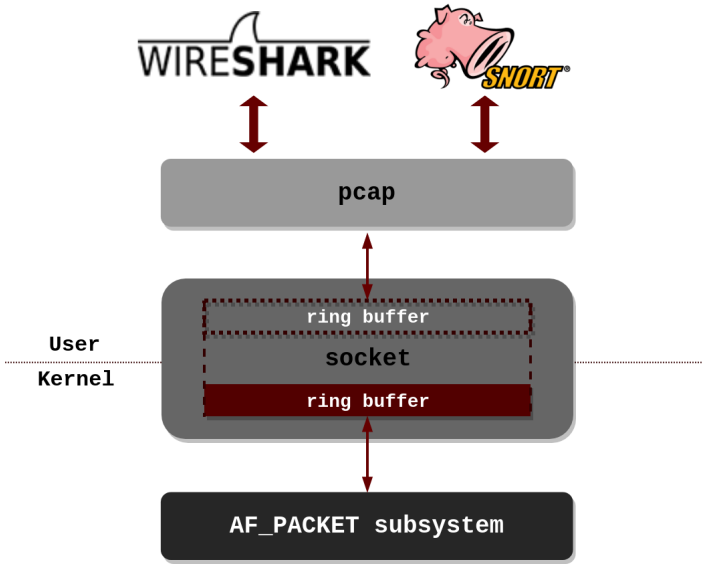


Figure 4.2: Layers between the applications and the kernel module.

- the level of abstraction of the **ring buffer allocation**: locating where `setsockopt()` is called with the original option `PACKET_RX_RING`
- the modality of configuration of the **ring buffer properties**: the size and number of the blocks, the size of the frames

As described in 6.3.4, the library function API for setup the raw socket is `pcap_create()` which on success return the `pcap` socket descriptor. This function checks the validity of the device and setting the different configuration options: enabling or disabling the promiscuous mode; setting the socket timeout value; setting the size of the snapshot length (the number of bytes taken from each incoming packet, regardless of the actual packet length); setting the socket buffer size.

There are dedicated APIs for the last mentioned settings that are, respectively: `pcap_set_promisc`, `pcap_set_timeout()`, `pcap_set_snaplen()`, `pcap_set_buffer_size()`.

After the socket configuration phase, the activation of the socket descriptor with

`pcap_activate()` completes the `pcap` socket descriptor setup.

During the activation operation, it is created the socket and set the relating options. The implementation for creation and setting options is specific per system architecture. Furthermore, `pcap_activate()` it is the lowest level of abstraction reachable using the library.

Going to analyze the internal implementation, in our case we consider the `pcap_activate_linux()` one to search for the hidden configuration of the buffer and the following relating allocation.

When the function `activate_new` is called there is an attempt to allocate an `AF_PACKET` socket and set the chosen timeout and buffer size. Both timeout and buffer size can be configured before activation: the `pcap` socket descriptor is used for the API `pcap_set_timeout()` and `pcap_set_buffer_size()`.

Then, the core of our interest resides in the `activate_mmap` function that tries to activate the memory-mapped access creating the ring buffer calling to `create_ring()` function. Here the buffer is setup filling the `tpacket_req3` (we assume to use the `TPACKET_V3` version).

The following one is the hidden configuration and creation of the ring:

- `tp_frame_size`: set arbitrarily to a constant `MAXIMUM_SNAPLEN`
  - defined as 262144 bytes to be big enough to capture the largest packets (Linux loopback packets, some USB packets)
- `tp_frame_nr`: deduced from the ratio between the buffer size and the frame size
  - by default the buffer size option is set to 2M, but it is configurable by using `pcap_set_buffer_size()` API

  - the buffer size is rounded up to avoid to allocate a buffer smaller of the required or a zero size if the buffer is less than a single frame

- `tp_block_size`: computed as the minumum size to handle a frame
  - starting from the page size obtained with `getpagesize()` and multiplying by two until the frame size is covered

- frames per block: computed as the integer ratio between the block size and the frame size

- `tp_block_size`: computed as the product between the total number of frames in the buffer and the number of frames per block

Moreover, in the case of the HugeTLB option, the socket setup flow is radically different from the standard one. Indeed, the `pcap` library follows the steps described in 2.3.3, but the `PACKET_HUGETLB_RING` option requires an alternative one (5.2.1).

Accordingly to the analysis, it seems that the points that clashes with all the new options are the block size setting and the creation of the ring at the moment of the `setsockopt` call. The latter is included because obviously it sets by default the option to `PACKET_RX_RING` that was the only option available for reception buffer allocation.

Furthermore, considering the option that uses the HugeTLB filesystem, the workflow to be run for setup the socket changes, too. The `mmap()` function has to be adapted. In this case, it becomes necessary, because the mapping of the allocated memory is done by the user before invoking the `setsockopt()` to setup the socket buffer (as described in detail in 5.4). Accordingly, even the unmapping function has to be adapted, too.

Therefore, the operation of integration has to take into account to alter the `getpagesize()` and the `setsockopt()` functions.

First, `getpagesize()` is the function from which depends the block size configuration. Originally it returns always the base page size given by `PAGE_SIZE`. This behavior is in contrast with the `PACKET_HUGETLB_RING` and `PACKET_TRANSHUGE_RING` options that conversely refers to the huge pages.

In particular, every attemp to allocate a block size smaller than one huge page would be more costly than the original implementation: it would be allocated a whole huge page uselessly, because the most part of the space would be unused.

Finally, the changes regarding the `setsockopt()` function extend the range of the available options, allowing to select the specified one also when it is different from the default one.

4.2.3 `LD_PRELOAD` trick

The first solution to the integration of the `pcap` library with the kernel patch is to patch the library itself. But in order to keep a high **backward compatibility**, we choose to avoid the direct patch, because going to change the source code of the library would bring towards cascaded software updates.

This is not a rare situation, because it is more convenient to indirectly modify the library when needed than alter a whole library. Specially if the changes that it is necessary to put on are minimal in front of the reminder of the library's content.

There is a simple but powerful trick that goes in this direction: the so-called **LD_PRELOAD trick**. It exploits the functionality of the dynamic linker on Unix systems to link during load or run-time, allowing to bind symbols provided by a certain shared library *before* other libraries. A symbol is any function, structure or

variable declaration that a program can reference in code.

Therefore, the solution to avoid to alter the `pcap` library is to implement our own alternative implementations of those functions that are crucial for integration with the new kernel feature keeping the same symbols of the original ones.

At the start of the capture application the chosen option of allocation is exported as an **environment variable** to be recoverable in our functions in the preloaded library, using the `setenv` and the `getenv` functions from `stdlib.h`.

The use of an environment variable is a way to globally know the chosen option. Therefore, when running the code inside the preloaded function, it is possibile to get the variable value and to differentiate the behavior based on it: the routine can bring back to the original function or can execute the instructions concerning the new options.

Since the invocation of a certain routine can happen many times even when it is not the case to be preloaded, it is necessary to recognize if that is the case of interest too. And if it is not, to call the original function recovering the respective address.

In order to use the original function definition, `dlsym()` [9] returns the address of the specified symbol. It is implemented in `dlfcn.h` (`dl` library). The function finds the next occurrence in the search order after the current one specifying as handle `RTLD_NEXT`. In our case, calling it from the preloaded routine means to obtain the address of the original one.

Chapter 5

The implementation details

5.1 The AF_PACKET module

5.1.1 The user interface

Approaching to the implementation details of the proposed solution, we show in this section the changes made to the kernel code using the typical syntax of the code versioning software. Hence, to highlight the difference between the original version and the new one the lines that was added to the previous version are preceded by the symbol "+".

The interface for the use of the new features is the first change required. As described in 4.1.2, the new setting options expand the set of which were already available.

The kernel module that implements the AF_PACKET domain's facilities is implemented in *net/packet/af_packet.c*.

The generic socket operations are mapped with the packet protocol ones in the struct packet_ops. This structure keeps the pointers to the specific protocol functions corresponding to the generic ones. Among these functions, there are the following

ones:

- `packet_setsockopt`: to apply the socket configuration with the settings described in 2.3.1
- `packet_getsockopt`: to get the value of the specified option
- `packet_mmap`: to create a new mapping in the virtual address space of the user process for the socket file descriptor
- `packet_release`: to close the socket and release the relating allocating resources

The options introduced can be chosen by the user following the same mode as before: making a call to `setsockopt` to configure the socket as desired.

Then, as shown in the Figure 5.1, six new cases corresponding to the six new options are inserted in the `packet_setsockopt` function.

The handler of the switch case which those options refers to is the same for each option of them concerning the allocation of the ring buffer.

Instead, the sixth option is the one introduced only for setup the memory buffer backed with the pre-allocated huge pages. As described in 5.2, this option has to be handled before the initialization of the reception ring that would fail otherwise.

The differenziated behavior is left to `packet_set_ring`. Because of this the function definition is changed in order to keep the reference to the specific option to handle.

5.1.2 The socket ring setup

As shown in Figure 5.2, the function declaration changes from the original one, but only semantically. The last argument is still of type *integer*, but now it takes the value

```
+#ifdef CONFIG_PACKET_HUGE
+       case PACKET_HUGETLB_ENABLE:
+       {
+               unsigned long val;
+
+               if (optlen != sizeof(val))
+                       return -EINVAL;
+
+               if (copy_from_user(&val, optval, optlen))
+                       return -EFAULT;
+
+               if (po->huge_mm.shmaddr)
+                       return -EEXIST;
+
+               po->huge_mm.shmaddr = val;
+
+               return 0;
+       }
+       case PACKET_HUGETLB_RING:
+       case PACKET_TRANSHUGE_RING:
+       case PACKET_GFPNORETRY_RING:
+       case PACKET_VZALLOC_RING:
+       case PACKET_GFPRETRY_RING:
+#endif
        case PACKET_RX_RING:
        case PACKET_TX_RING:
        {
                ...
+#ifdef CONFIG_PACKET_HUGE
+               ...
+               if (optname == PACKET_HUGETLB_RING
+                                       && !po->huge_mm.shmaddr)
+                               ret = -EINVAL;
+               else
+                               ret = packet_set_ring(sk, &req_u,
+                                       0, optname);
+#else
                ret = packet_set_ring(sk, &req_u, 0,
                                      optname == PACKET_TX_RING);
+#endif
        }
        release_sock(sk);
        return ret;
 }
```

Figure 5.1: New cases handled in `packet_setsockopt()`.

of the option number rather than a boolean value to indicate if it is a transmission buffer or not.

In the part of the function where the sanity checks are made, a new one is added to assure that the block size specified in the request is aligned to the huge page size (`HPAGE_PMD_SIZE`) in case of THP block allocation or to the huge page size configured in the `hugetlbfs` filesystem in case of HugeTLB block allocation.

This further **sanity check** is introduced to avoid the allocation of a size of memory smaller than the desired one. Indeed, when the actual allocation occurs, the size to be allocated is forcely rounded down to the nearest page size multiple to become aligned.

Then, there is a modification in the call to the function for allocating the ring buffer and each block being part of it. As for the `packet_set_ring` function, the declaration of the `alloc_pg_vec` is changed in order to keep the value of the type of allocation to be performed.

The change in the declaration of the function `packet_set_ring` brings to the adjustment of the usages of the function in other points of the code.

5.1.3 The socket release

The `packet_release` function makes a call to the `packet_set_ring` setting the argument *closing* in order to release the resources of the socket to be closed.

Hence, as shown in Figure 5.3, when there is a transmission buffer to release the option `PACKET_TX_RING` is specified.

On the other hand, when there is a reception buffer to release, it is no necessary to change this case, too. In `packet_set_ring` the evaluations of the option argument is interesting only when it indicates a transmission ring; otherwise, the behavior is equivalent for any type of reception ring.

```
+#ifdef CONFIG_PACKET_HUGE
+static int packet_set_ring(struct sock *sk, union tpacket_req_u *req_u,
+                int closing, int optname)
+#else
 static int packet_set_ring(struct sock *sk, union tpacket_req_u *req_u,
                 int closing, int tx_ring)
+#endif
 {
          ... /* initialization */
+#ifdef CONFIG_PACKET_HUGE
+        int tx_ring = (optname == PACKET_TX_RING);
+        int huge_ring = (optname == PACKET_HUGETLB_RING);
+        int thuge_ring = (optname == PACKET_TRANSHUGE_RING);
+        unsigned int hpagesize = huge_page_size(&default_hstate);
+        po->huge_mm.thp_block = thuge_ring;
+#endif
         rb = tx_ring ? &po->tx_ring : &po->rx_ring;
         rb_queue = tx_ring ? &sk->sk_write_queue : &sk->sk_receive_queue;
         ... /* sanity checks */
         if (unlikely(!PAGE_ALIGNED(req->tp_block_size)))
                 goto out;
+#ifdef CONFIG_PACKET_HUGE
+        if (unlikely(!IS_ALIGNED(req->tp_block_size, HPAGE_PMD_SIZE)
+                        && thuge_ring))
+                goto out;
+        if (unlikely(!IS_ALIGNED(req->tp_block_size, hpagesize)
+                        && huge_ring))
+                goto out;
+#endif
         ...
         order = get_order(req->tp_block_size);
+#ifdef CONFIG_PACKET_HUGE
+        pg_vec = alloc_pg_vec(req, &po->huge_mm, optname);
+#else
         pg_vec = alloc_pg_vec(req, order);
+#endif
         ...
```

Figure 5.2: The changes in the `packet_set_ring` function.

```
 static int packet_release(struct socket *sock)
        ...
        lock_sock(sk);
        if (po->rx_ring.pg_vec) {
                memset(&req_u, 0, sizeof(req_u));
+#ifdef CONFIG_PACKET_HUGE
+               if (po->huge_mm.shmaddr)
+                       option = PACKET_HUGETLB_RING;
+               else if (po->huge_mm.thp_block)
+                       option = PACKET_TRANSHUGE_RING;
+               else
+                       option = PACKET_RX_RING;
+               packet_set_ring(sk, &req_u, 1, option);
+#else
                packet_set_ring(sk, &req_u, 1, 0);
+#endif
        }

        if (po->tx_ring.pg_vec) {
                memset(&req_u, 0, sizeof(req_u));
-               packet_set_ring(sk, &req_u, 1, 1);
+#ifdef CONFIG_PACKET_HUGE
+               packet_set_ring(sk, &req_u, 1, PACKET_TX_RING);
+#else
+               packet_set_ring(sk, &req_u, 1, 1);
+#endif
        }
        release_sock(sk);
        ...
}
```

Figure 5.3: The changes in the `packet_release` function to adapt to the new definition of `packet_set_ring`.

After the packet release, the ring buffer memory has to be freed. This happens in the function `free_pg_vec` (Figure 5.10) that was properly adapted to the new allocation modes.

The only case to be adapted is the one concerning the use of the HugeTLB filesystem: it is added a specific routine (`free_one_pg_huge`) that is described in 5.2.3.

About what concerns the other options, the buffer is freed using `vfree` when virtually allocated and using `free_pages` otherwise.

```
+#ifdef CONFIG_PACKET_HUGE
+static void free_pg_vec(struct pgv *pg_vec, unsigned int order,
+                        unsigned int len, struct huge_mm *hmm)
+#else
 static void free_pg_vec(struct pgv *pg_vec, unsigned int order,
                         unsigned int len)
+#endif
 {
        int i;
        for (i = 0; i < len; i++) {
                if (likely(pg_vec[i].buffer)) {
+#ifdef CONFIG_PACKET_HUGE
+                       if (hmm->shmaddr)
+                               free_one_pg_huge(i,
+                                       (unsigned long)pg_vec[i].buffer,
+                                       len, hmm);
+                       else if (is_vmalloc_addr(pg_vec[i].buffer))
+                               vfree(pg_vec[i].buffer);
+#else
                        if (is_vmalloc_addr(pg_vec[i].buffer))
                                vfree(pg_vec[i].buffer);
+#endif
                        else
                                free_pages((unsigned long)pg_vec[i].buffer,
                                           order);
                        pg_vec[i].buffer = NULL;
                }
        }
        kfree(pg_vec);
 }
```

Figure 5.4: The changes in the `free_pg_vec` function.

As shown in 5.5, the number of arguments described in the declaration of the function is changed from two to three.

In order to make available the information about the chosen option and the relating data, the function is re-defined with three parameters: the first, as the previous version, includes the content of the `tpacket_req3` request; the second, in the

`PACKET_HUGETLB_RING` case, includes the pointer to the buffer memory descriptor when using the huge pages (`huge_mm` structure described in 5.2.2); the last one includes the option itself.

The changes are useful to differentiate the behavior between the cases relating to the specified option.

```
+#ifdef CONFIG_PACKET_HUGE
+static struct pgv *alloc_pg_vec(struct tpacket_req *req,
+                struct huge_mm *hmm, int option)
+#else
 static struct pgv *alloc_pg_vec(struct tpacket_req *req, int order)
+#endif
 {
        unsigned int block_nr = req->tp_block_nr;
        struct pgv *pg_vec;
        int i;
+#ifdef CONFIG_PACKET_HUGE
+       unsigned long size = req->tp_block_size;
+       int order = get_order(size);
+#endif
        pg_vec = kcalloc(block_nr, sizeof(struct pgv), GFP_KERNEL);
        for (i = 0; i < block_nr; i++) {
+#ifdef CONFIG_PACKET_HUGE
+               if (hmm->shmaddr)
+                       pg_vec[i].buffer = alloc_one_pg_vec_hugepage(
+                                       i, size, block_nr, hmm);
+               else
+                       pg_vec[i].buffer = alloc_one_pg_vec_page(order,
+                                       option);
+#else
                pg_vec[i].buffer = alloc_one_pg_vec_page(order);
+#endif
        ...
```

Figure 5.5: The changes made in the `alloc_pg_vec`.

The changes described above are common to both the solutions. Instead, the functions for buffer allocation and de-allocation require a case-specific implementation.

5.2 Using the HugeTLB filesystem

5.2.1 User side socket allocation

The solution implemented by using the Hugetlb filesystem interface involves the user in a **greater effort** compared to the default steps required for using the packet socket (2.3.3). This can be seen comparing the Figures 5.6 and 5.11.

Indeed, this solution delegates to the user both the allocation and the mapping of the buffer. On the other hand, the original implementation and also the second solution proposed below provide a transparent behavior for these purposes.

The modality used for ring allocation and mapping changes. If users want to use the option `PACKET_HUGETLB_RING`, they have to use the required new option `PACKET_HUGETLB_ENABLE`.

Setting the latter option mentioned involves the configuration of the ring buffer by kernel side.

After the creation of the socket raw, the proper **allocation** of the buffer takes place in user space though the interface to the HugeTLB filesystem. Assuming the user mounted the filesystem before (as described in 6.2.3), the allocation of the buffer is performed through the creation of a new file in the path corresponding to the mountpoint of the HugeTLB filesystem.

Since in this solution also the mapping into memory of the buffer takes place in the user space, the user has to perform this operation mapping the file just created into memory. The size of the memory mapping is equal to the buffer size.

When the capture process ends, user takes charge of delete the mapping into memory and to de-allocate the previously reserved huge pages. The first operation requires the `munmap()` of the area and the latter the removal of the file into the HugeTLB filesystem.

Accordingly, the workflow for using this option leads to execution of the following steps:

1. setup:

    - `socket()`: creation of the capture socket

- `open()`: creation of a new file in the HugeTLB filesystem for the buffer allocation
- `mmap()`: mapping the huge file into memory of size equal to the buffer length
- `setsockopt()`: enabling the initialization by kernel side of the ring buffer by transfer the mapped memory address using the option `PACKET_HUGETLB_ENABLE`
- `setsockopt()`: complete the creation of the reception buffer using the option `PACKET_HUGETLB_RING`

2. capture:
   - `poll()`: active waiting for incoming packets
3. shutdown:
   - `munmap()`: delete the mapping of the buffer before de-allocation
   - `close()`: closing the socket file descriptor
   - `close()`: closing the HugeTLB file descriptor
   - `unlink()`: removing the HugeTLB file

In particular, only the initial phase requires an additional option to transfer to kernel side the reference to the allocated buffer (`PACKET_HUGETLB_ENABLE`). The `mmap` socket operation in the `AF_PACKET` module is not used for the allocation using the `PACKET_HUGETLB_RING` option. Indeed, the mapping of the file created in HugeTLB filesystem during the socket setup happens in the user space. This is the reason why

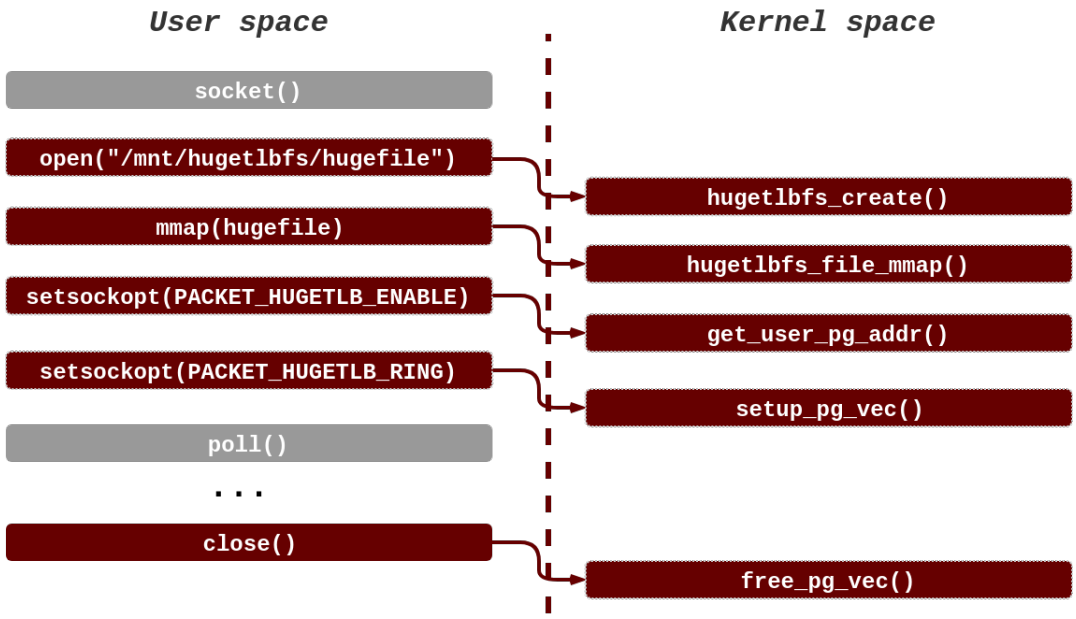


Figure 5.6: The process flow of the application between the user and kernel side.

resorting to a further option to make available to the kernel space the information of the mapping.

The procedure for closing the packet socket includes the release of the buffer by kernel side. The kernel keeps the reference to the memory area allocated at the opening when enabling the use of packet socket with the HugeTLB filesystem.

The schema of the capture flow by both user and kernel space is shown in Figure 5.6.

5.2.2 Kernel side ring initialization

In order to support the new options, the kernel side solution introduces a new structure. The structure is used to keep the required memory references and it is shown in the Figure 5.7. It describes the type of a new field added to the structure `packet_sock` (5.8). i rimanda a In this way, `huge_mm` links the buffer's references to the structure that describes each socket, `packet_sock` (5.8). The new field makes possible to

access them during all the socket's lifecycle.

```
struct huge_mm {
        unsigned long           shmaddr;
        struct page             ***hblocks;
        int                     npages;
        int                     thp_block;
};
```

Figure 5.7: The kernel struct defined in the new file `if_huge_packet.h` as memory descriptor of the huge block.

The `shmaddr` field contains the user memory mapped address trasferred through the `PACKET_HUGETLB_ENABLE` option.

The `hblocks` field is a triple pointer to a `struct page`. It keeps the reference to each page structure that describes each huge page in a block, for each block in the ring buffer. Hence, `hblocks` is an array of size given by the number of the blocks. Every item is another array of size given by the number of the needed huge pages to fill the block size. Every item of this array is a pointer to a page structure that describes a huge page.

The `npages` field counts the number of huge pages per block.

Finally, the `thp_block` field is a boolean variable to know if the chosen option for the ring allocation is the one asking for the use of the transparent huge pages.

```
struct packet_sock {
        ...
        struct net_device __rcu *cached_dev;
        int                     (*xmit)(struct sk_buff *skb);
        struct packet_type      prot_hook ____cacheline_aligned_in_smp;
+#ifdef CONFIG_PACKET_HUGE
+       struct huge_mm          huge_mm;
+#endif
};
```

Figure 5.8: The new field added in the existing structure `packet_sock` defined in *net/packet/internals.h*.

The initialization of the socket raw in the user space requires enabling the ring buffer. It means, as described in 5.2.1, to tranfer the mapped memory address to the

kernel space.

The body of the option `PACKET_HUGETLB_ENABLE` in `packet_setsockopt` (5.1) implements the procedure to set the buffer references in the kernel module. After some checks, it retrieves the user value which is used to fill the `shmaddr` field in the `huge_mm` structure of the packet socket.

Then, now the `AF_PACKET` module knows where was mapped by the user the ring buffer to be used for packet reception.

When it is set the option `PACKET_HUGETLB_RING`, the procedure for setup the ring is different compared to the other options. But the call to the re-defined `packet_set_ring` function (5.1) is still common among all.

Adding a new parameter in the definition allows to keep the chosen option information. Then, it is possible to switch the behavior of the single case.

First, the `packet_set_ring` (5.2) provides an additional sanity check for the allocation using the HugeTLB filesystem: the block size has to be aligned with the huge page size.

Second, the function `alloc_pg_vec` (5.5) handles the allocation of the ring buffer of the packet socket. When the field `shmaddr` is not a null value, it means that we are in the HugeTLB case.

Hence, a function to this (`alloc_one_pg_vec_hugepage`) provides allocation of the block backed with the pre-allocated huge pages.

As shown in the Figure 5.9, the function requires four arguments. The index indicates the reference to the block in the buffer to be allocated. Both the index and the size are required because the block address depends from them. The number of blocks is needed to initialize the array of pointers to each block (`hblocks`). Finally, the pointer to the `huge_mm` structure allows to update the memory descriptor of the

buffer.

The first step performed in the function is the computation of the address where the requested block memory area begins. This is possible because the buffer allocation and mapping is just done by the user. Hence, the current block address is set to *index* times the size of the block after the base buffer address given by `shmaddr`.

Then, the number of required pages (4K) to cover the block size is retrieved by the ratio between the size itself and the `PAGE_SIZE` value.

If the value of index is equal to zero, it means that is the first block to be allocated. Then, the `struct huge_mm` global fields has to be initialized. The number of pages per block (`npages`) is set to the value of the ratio between the block size and value of `PAGE_SIZE`. It is not used the huge page size, because the kernel uses the same structure to represent both regular pages and huge pages. The array of pointers to each block (`hblocks`) is allocated with `vmalloc`.

For each block an array of pointers to each page is initialized (`hpages`). Then, calling `get_user_pages_fast()` allows to pin user pages into memory. In particular, we used it instead of `get_user_pages` considering the application performance of the application. Each user page returned fills the `hpages` values.

Considering the possibility to run in a NUMA system, this function finds the node identified by its id (`nid`) where the first page of `hpages` is allocated.

Huge pages are mapped into the kernel address space, but there is no guarantee of their **sequentiality**. This represents a problem when accessing a buffer backed by more than one huge page. The solution is using `vm_map_remap` to request a linear mapping of those pages. All pages are re-mapped into the node found before.

Finally, after update the block memory reference in the `huge_mm` descriptor, the `alloc_one_pg_vec_hugepage` returns the address of the block.

```
+#ifdef CONFIG_PACKET_HUGE
+static char *alloc_one_pg_vec_hugepage(int index, unsigned long size,
+                int block_nr, struct huge_mm *hmm)
+{
+       char *buffer;
+       void *base_addr;
+       unsigned long user_addr;
+       unsigned long block;
+       struct page **hpages;
+       int npages, nid, pinned;
+
+       user_addr = hmm->shmaddr + index*size;
+
+       npages = size / PAGE_SIZE;
+
+       if (!index)
+               hmm->hblocks = vmalloc(block_nr * sizeof(struct page **));
+
+       if (!hmm->npages)
+               hmm->npages = npages;
+
+       hpages = vmalloc(npages * sizeof(struct page *));
+
+       pinned = get_user_pages_fast(user_addr, npages, 1, hpages);
+
+       nid = page_to_nid(hpages[0]);
+       base_addr = vm_map_ram(hpages, npages, nid, PAGE_KERNEL);

+       hmm->hblocks[index] = hpages;
+
+       block = (unsigned long) base_addr;
+       buffer = (char *) block;
+       if (buffer)
+               return buffer;
+}
+#endif
```

Figure 5.9: Block backed with HugeTLB pages allocation.

5.2.3 Kernel side buffer de-allocation

When the user closes the socket, the module executes the release operation (Figure 5.3). As described in 5.1.3, the function that performs the de-allocation of the buffer is `free_pg_vec`.

As long as in the buffer backed with the Transparent Huge Pages the routine remains unchanged, in this case it is introduced a new function for this purpose: `free_one_pg_huge` (Figure 5.10).

As for block allocation, the function needs the index of the block to be de-allocated. The index indicates the position in the array of blocks `hblocks`.

First, the read lock related to the mapping of the current thread (`current->mm->mmap_sem`) has to be taken. Thanks to the `huge_mm` descriptor, all the pointers to the pages of the block are available given the index of the blocks. Then, for each page the function performs a check if it is reserved with `PageReserved()`. Normally, the pages mapped from user space should not be marked reserved. If the page is not reserved, the page is marked as `dirty` (`SetPageDirty()`) and released from the page cache (`put_page()`).

Finally, the lock is released and the array of reference to the block pages is freed. To complete the de-allocation of the buffer the structure `huge_mm` is reset when the last block was freed.

5.3 Using THP

5.3.1 User side socket allocation

By the user side, the `PACKET_TRANSHUGE_RING` option does not require a big effort. But the user awareness is required for enabling the use of Transparent Huge Pages

```
+#ifdef CONFIG_PACKET_HUGE
+static void free_one_pg_huge(int index, unsigned int len, struct huge_mm *hmm)
+{
+       int i;
+
+       if (current->mm)
+               up_read(&current->mm->mmap_sem);
+
+       for (i = 0; i< hmm->npages; ++i) {
+               if (!PageReserved(hmm->hblocks[index][i]))
+                       SetPageDirty(hmm->hblocks[index][i]);
+
+               put_page(hmm->hblocks[index][i]);
+       }
+
+       if (current->mm)
+               down_read(&current->mm->mmap_sem);
+
+       vfree(hmm->hblocks[index]);
+
+       if (index == len) {
+               hmm->npages = 0;
+               hmm->shmaddr = 0UL;
+               vfree(hmm->hblocks);
+               hmm->hblocks = NULL;
+       }
+}
+#endif
```

Figure 5.10: Block backed with HugeTLB pages de-allocation.

in the system and for selecting the option when invoking the `setsockopt` call for the ring allocation.

The schema of the capture flow by both user and kernel space is shown in Figure 5.11. Comparing with the Figure 5.6, a heavier effort in the user space can be seen.

5.3.2 Kernel side ring allocation

Therefore, we come to the core of the block allocation that is implemented in `alloc_one_pg_vec` (Figure 5.12). The type of allocations are divided in five possible cases, where the actual one depends on the option passed as argument.

The default case is the original one, where three allocation attempts are executed in order of preference relating to the efficiency: direct physical memory reclaim, only virtually contiguous memory, indefinitely repeated physical memory reclaim.

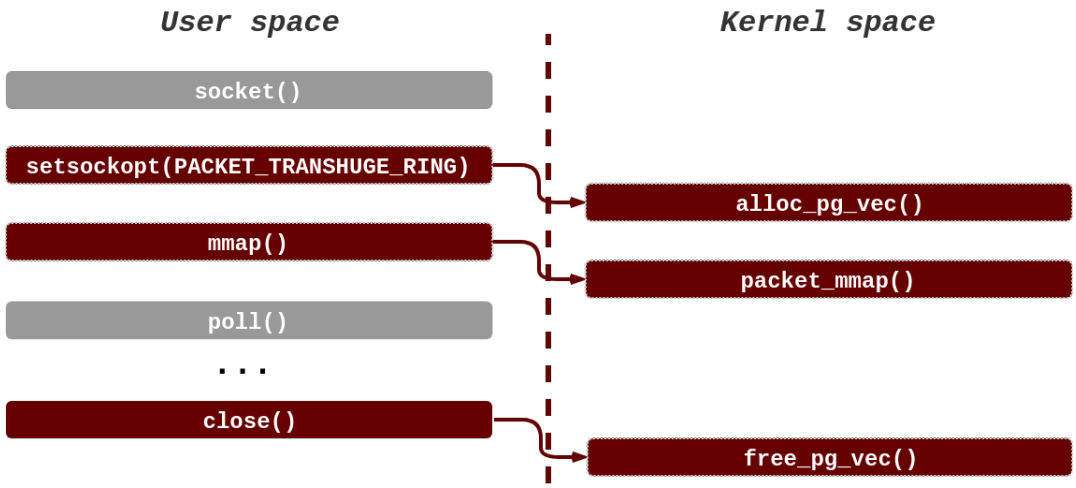


Figure 5.11: The process flow of the application between the user and kernel side.

Therefore, the default case is what concerns the setting of the option `PACKET_RX_RING`. The reminder four cases concern the new options:

- `PACKET_HUGETLB_RING`: memory area backed with huge pages using the HugeTLB filesystem
- `PACKET_TRANSHUGE_RING`: direct memory reclaim of transparent huge pages (the same used by the THP `khugepaged` kernel thread)
- `PACKET_GFPNORETRY_RING`: forced physically contiguous memory allocation even if it can bring esaily to failure in loaded memory conditions
- `PACKET_VZALLOC_RING`: forced the virtually contiguous memory area allocation
- `PACKET_GFPRETRY_RING`: forced the physically contiguous memory allocation retrying the memory reclaim (sometimes resorting to the swap area)

5.3.3 Kernel side ring de-allocation

As described in 5.1.3, the phase of release the memory resources relating to the ring buffer is equal to the original implementation in the case of a buffer backed with

```
+#ifdef CONFIG_PACKET_HUGE
+static char *alloc_one_pg_vec_page(unsigned long order, int optname)
+#else
 static char *alloc_one_pg_vec_page(unsigned long order)
+#endif
 {
        char *buffer;
        gfp_t gfp_flags = GFP_KERNEL | __GFP_COMP |
                          __GFP_ZERO | __GFP_NOWARN | __GFP_NORETRY;
+#ifdef CONFIG_PACKET_HUGE
+       switch (optname) {
+       case PACKET_TRANSHUGE_RING:
+       {
+               gfp_flags |= GFP_TRANSHUGE;
+               buffer = (char *) __get_free_pages(gfp_flags, order);
                ...
+               break;
+       }
+       case PACKET_GFPNORETRY_RING:
+       {
+               buffer = (char *) __get_free_pages(gfp_flags, order);
                ...
+               break;
+       }
+       case PACKET_VZALLOC_RING:
+       {
+               buffer = vzalloc(array_size((1 << order), PAGE_SIZE));
                ...
+               break;
+       }
+       case PACKET_GFPRETRY_RING:
+       {
+               gfp_flags &= ~__GFP_NORETRY;
+               buffer = (char *) __get_free_pages(gfp_flags, order);
                ...
+               break;
+       }
+       default:
+#endif
        buffer = (char *) __get_free_pages(gfp_flags, order);
        ...
        buffer = vzalloc(array_size((1 << order), PAGE_SIZE));
        ...
        gfp_flags &= ~__GFP_NORETRY;
        buffer = (char *) __get_free_pages(gfp_flags, order);
        ...
+#ifdef CONFIG_PACKET_HUGE
+       }
+#endif
        return NULL;
 }
```

Figure 5.12: The changes in the `alloc_one_pg_vec` function.

```
gcc -shared -fPIC -o preload.so preload.c -ldl
```

Figure 5.13: Compiling the shared library `preload.so` using GCC.

transparent huge pages.

Indeed, with the option `PACKET_TRANSHUGE_RING`, the execution falls into the `free_pages` case. That function handles the de-allocation of a memory area allocated with `get_free_pages` routine, even if it is specified the `GFP_TRANHUGE` flag.

5.4 The preloaded `pcap` library

Our implementations of the routines discussed in 4.2.2 are included in the file `preload.c`. The preloaded function are compiled into a shared library as shown in Figure 5.13.

Then, for using the `LD_PRELOAD` trick, the capture application implemented with the `pcap` library (described in 6.3.4) has to be preceded by the setting of the `LD_PRELOAD` environment variable before starting the application. The variable has to be set to the value of the path to the preloaded library `preload.so`.

Concerning the alternative implementation of the `getpagesize()` routine, it is introduced a check on the allocation option retrieved by the value of the environment variable set by the application before activate the `pcap` socket descriptor.

If the option is equal to `PACKET_HUGETLB_RING` then the value of page size returned is not the base one (`PAGE_SIZE`), but the huge one. The function used to retrieve the huge page size accesses to the sysfs to get the size configured.

If the option is equal to `PACKET_TRANSHUGE_RING` then the value returned is the page size specified in the `HPAGE_PMD_SIZE` macro.

Otherwise, if any other option is specified, the original routine is called and which reference is found using `dlsym` mentioned above.

```
int getpagesize(void) {
        char *val = getenv(OPTION);
        if (val) {
                int opt = atoi(val);
                if (is_hugetlb(opt))
                        return get_hugepage_size();
                if (is_thp(opt)
                        return HPAGE_PMD_SIZE;
        }
        int (*original_getpagesize)(void);
        original_getpagesize = dlsym(RTLD_NEXT, "getpagesize");
        return original_getpagesize();
}
```

Figure 5.14: Preloaded version of `getpagesize`.

The code of the preloaded `getpagesize` symbol is shown in Figure 5.14.

Concerning the alternative implementation of the `setsockopt()` routine, it is introduced a check on the allocation option in this preloaded routine too.

If the option is equal to any of the new available options (`PACKET_HUGETLB_RING`, `PACKET_TRANSHUGE_RING`, `PACKET_GFPNORETRY_RING`, `PACKET_VZALLOC_RING`, `PACKET_GFPRETRY_RING`) then it is perfomed a call to the original symbol but with an alteration: the value of the parameter `optval` first was assuming the value in the intercepted call (`PACKET_RX_RING`) and then it is switched to the value specified in the environment variable.

Otherwise, if any other option is specified, it is called the original function replaying the same arguments of the intercepted call.

The code of the preloaded `setsockopt` symbol is shown in the Figure 5.15.

As shown in the figure, there is not only one environment variable value indicating the chosen option. Also an environment variable containing the number of the file descriptor of the raw socket can be set.

In order to assign a value to the socket fd variable, that value has to be intercepted elsewhere. Preloading the `socket` symbol of the function that return the socket file descriptor, it will be possible to access to the value in the next preloaded symbols.

```
int setsockopt(int sockfd, int level, int optname,
               void *optval, unsigned int *optlen) {

        int (*original_setsockopt)(int sockfd, int level, int optname,
                        void *optval, unsigned int *optlen);
        original_setsockopt = dlsym(RTLD_NEXT, "setsockopt");

        void *(*original_mmap)(void *addr, size_t length, int prot,
                               int flags, int fd, off_t offset);
        original_mmap = dlsym(RTLD_NEXT, "mmap");

        char *env_option = getenv(OPTION);
        char *env_fd = getenv(SOCKFD);
        if (env_option && env_fd) {
                int opt = atoi(env_option);
                int efd = atoi(env_fd);
                if (is_newoption(opt) && efd == sockfd
                        && level == SOL_PACKET && optname == PACKET_RX_RING) {
                        if (is_hugetlb(opt)) {
                                int prot = PROT_READ | PROT_WRITE, flag = MAP_SHARED;
                                int size;
                                struct tpacket_req3 *req = (struct tpacket_req3 *)
                                    optval;

                                size = req->tp_block_nr*req->tp_block_size;

                                memset(&hmm, 0, sizeof(hmm));

                                hmm.size = size;

                                hmm.usr_addr = (unsigned long) mmap_hugepages(
                                                size, original_mmap, prot, flag);

                                original_setsockopt(
                                        efd, SOL_PACKET, 24,
                                        &hmm.usr_addr, sizeof(hmm.usr_addr));
                        }
                        return original_setsockopt(sockfd, level,
                                        opt, optval, optlen);
                }
        }
        return original_setsockopt(sockfd, level, optname, optval, optlen);
}
```

Figure 5.15: Preloaded version of `setsockopt`.

```
int socket(int domain, int type, int protocol) {

        int (*original_socket)(int domain, int type, int protocol);
        original_socket = dlsym(RTLD_NEXT, "socket");

        int fd = original_socket(domain, type, protocol);
        char env_fd[5];
        sprintf(env_fd, "%d", fd);
        if (domain == AF_PACKET && type == SOCK_RAW) {
                int status;
                status = setenv(SOCKFD, env_fd, 1);
        }
        return fd;
}
```

Figure 5.16: Preloaded version of `socket`.

```
struct huge_mm {
        unsigned long usr_addr;
        char *filename;
        int hfd;
        size_t size;
};
```

Figure 5.17: The global memory descriptor of the buffer backed with huge pages.

The Figure 5.16 shows the preloaded version of the `socket` function. First, the preloaded routine calls the original `socket` to save the file descriptor. If the options of the socket creation corresponds to the `AF_PACKET` domain and to the `SOCKET_RAW` type, then the environment variable is set.

As seen in the Figure 5.15, the case of the option set to `PACKET_HUGETLB_RING` is more complicated that the other ones. Indeed, the pcap application cannot be aware of the additional user space operations required to setup the buffer using the HugeTLB filesystem. Hence, this steps has to be performed into the preloaded function.

When we are in the case of the `HUGETLB` allocation, a new structure describes the memory references to the buffer backed with huge pages, `huge_mm` (Figure 5.17): the `usr_addr` field to keep the value of the address of the mapping transferred via `setsockopt`; the `filename` field for the name of the file in the HugeTLB filesystem; the `hfd` field for the file descriptor of the file; the `size` field for the length of the buffer memory area.

```
void *mmap(void *addr, size_t length, int prot, int flags, int fd, off_t offset) {

        void *(*original_mmap)(void *addr, size_t length, int prot,
                               int flags, int fd, off_t offset);
        original_mmap = dlsym(RTLD_NEXT, "mmap");

        char *env_option = getenv(OPTION);
        char *env_fd = getenv(SOCKFD);

        if (env_option && env_fd && length == hmm.size) {
                int opt = atoi(env_option);
                int efd = atoi(env_fd);
                if (is_hugetlb(opt) && efd == fd)
                        return (void *) hmm.usr_addr;
        }
        return original_mmap(addr, length, prot, flags, fd, offset);
}
```

Figure 5.18: Preloaded version of `mmap`.

When the pcap library creates a memory mapping for the ring buffer, the new allocation policy using the HugeTLB filesystem requires an alternative way of mapping the area of memory using the filesystem. Then, the `mmap` operation has to be preloaded. The Figure 5.18 shows the preloaded implementation of the `mmap` symbol. To know if the intercepted mmap is the one mapping the ring buffer are checked some parameters: the socket file descriptor saved in the environment variable has to be equal to the one passed as argument; the size of the buffer saved on the `huge_mm` descriptor has tobe equal to the one passed as argument; the option saved in the environment variable has to be the `PACKET_HUGETLB` one. If these checks are verified, the address mapped with `mmap_hugepages` is returned. In all the other cases, the original `mmap` implementation is executed.

The same approach is followed for the `munmap` preloaded version (5.19).

Both for mapping and unmapping operations that are included in the preloaded functions, the process to do it is the same described in 5.2.1.

```
int munmap(void *addr, size_t length) {

        int (*original_munmap)(void *addr, size_t length);
        original_munmap = dlsym(RTLD_NEXT, "munmap");

        char *env_option = getenv(OPTION);
        char *env_fd = getenv(SOCKFD);

        if (env_option && env_fd) {
                int opt = atoi(env_option);
                if (is_hugetlb(opt) && length == hmm.size)
                        return munmap_hugepages(original_munmap, length);
        }
        return original_munmap(addr, length);
}
```

Figure 5.19: Preloaded version of `munmap`.

Chapter 6

Performance evaluation

6.1 Profiling tools

6.1.1 Measurements

On much modern hardware, there is a **Performance Measurement Unit** (PMU) which provides a small number of hardware-based counters. The PMU is programmed to increment a counter when a specific low-level event occurs and interrupt the CPU when a threshold, called the sample period, is reached. In many cases, there will be one low-level event that corresponds to a TLB miss so a reasonable estimate can be made of the number of TLB misses.

Hence, the PMU hardware is defined to count low-level events. On Linux, PMU can be programmed with tools as `oprofile` and `perf`. `Perf` analyzer [23] maps high-level (cache misses) to low-level events, but it is not able to recognize some TLB events, such as the number of cycles spent walking a page table. `Oprofile` analyzer [22] can be used to *estimate* the number of TLB misses using the PMU. To avoid excessive overhead counting all the TLB miss events, PMU records events during a sample-period. When the sample-period is reached, an interrupt is raised

and `oprofile` records the details of that event. An estimate of the real number of TLB misses that occurred is then:

$$\text{EstimatedTLB\_Misses} = \text{TLB}_{\text{misses\_sampled}} * \text{SamplePeriod}$$

Concerning the performance evaluation of a block of code, not only the architectural events are significative, but how to measure time application-side is crucial, too. The `perf` tool allows us to measure also the execution time in seconds, so it is equivalent to the measure obtained using the command line `time` tool.

Another possibility to measure time is the `time.h` library modifying the code to profile.

In order to test our work, we use `perf` [21]. In particular, among all events measurable using the tool, the events we are interested in are:

- `cache-misses`: kernel PMU event
- `page-faults`: software event
- `cpu-cycles`: hardware event
- `instructions`: hardware event
- `dTLB-loads`, `dTLB-stores`, `dTLB-load-misses`, `dTLB-store-misses`: hardware cache event

Moreover, testing on Intel machines (6.1), to measure the number of counter of CPU clocks we access to the value of **Constant Time Stamp Counter** register.

6.2 Administration

6.2.1 Hardware technical specifications

The technical specifications of the two servers employed for running the tests are shown in Table 6.1.

One of them it is used for sending packets at a rate as highest as possibile compared to the sending tool possibility. The other one is the most performing and it is employed to capture incoming packets to test the patches.

Packet generator machine			
Processor	Logic cores	RAM	Ethernet controller
Intel® Xeon E5-3695 v3 2.3 GHz	56	256 GB	Intel® 82599ES

Packet capture machine			
Processor	Logic cores	RAM	Ethernet controller
Intel® Xeon E5-2699 v4 2.2 GHz	88	512 GB	Intel® X710

Table 6.1: Testing machine specifications

6.2.2 Machine performance settings

The testing machine configuration is oriented to reach the highest performance level.

In the network interface cards (NIC) a technology is available for **offloading** part or all the TCI/IP protocol processing from the host CPU to the NIC itself. This technology is called TCP offloading engine (TOE) and it is used with high-speed network interfaces (gigabit Ethernet) where the network stack processing becomes more relevant.

On the other side, there are some configuration where these features are not beneficial at all and the one we choose is one of these. The reason behind this is the fact that the network adapter is not powerful enough to handle the offload capabil-

ities at high throughput. Furthermore, even if we focus on the reception-side of the network, offloading the TCP segmentation operation from the Linux network stack to the adapter can lead to enhanced performance for interfaces with predominately large outgoing packets.

Disabling these features is obtained using **ethtool** command line tool as shown in Figure 6.1.

```
ethtool -K <interface> lro off gro off tso off tx off rx off gso off sg off ufo off
ethtool -A <interface> rx off tx off autoneg off
```

Figure 6.1: The sequence of commands to disable the offloading and pause features.

Another aspect we need to take into account to configure the performance level of our testing machine is the dynamic CPU **frequency scaling**. It is a technique where a processor is not run at the maximum frequency in order to conserve power. Setting the maximum frequency is possible through `sysfs` executing from command line the command in 6.2 for each CPU X.

```
echo performance > /sys/devices/system/cpu/cpuX/cpufreq/scaling_governor
```

Figure 6.2: Enabling the scaling configuration of the CPU X to `performance`.

6.2.3 Huge pages tuning

Due to the inability to swap huge pages, their allocation can bring to a waste of memory. In order to use them, it is necessary to configure a pool with static or dynamic size.

With the static configuration the number of huge pages are pre-allocated at boot-time and the pool size may be set with the kernel boot parameter `hugepages`. At runtime, the pool size may be reconfigured changing the value of `vm.nr_hugepages` via `sysctl` or the value of the virtual file */sys/kernel/mm/hugepages/nr_hugepages*.

```
mount -t hugetlbfs /mnt/hugetlbfs-64k -o pagesize=2048K
echo never > /sys/kernel/mm/transparent_hugepage/enabled
echo <nr> > /proc/sys/vm/nr_hugepages
echo <nr> > /proc/sys/vm/nr_overcommit_hugepages
```

Figure 6.3: The huge pages pool configuration for HugeTLBfs patch.

Moreover, it is possible to allocate an additional number of huge pages up to `nr_overcommit_hugepages` when there are no sufficient pool pages to satisfying all requests for huge pages.

In order to execute our test, assuming we have compiled the kernel enabling the required kernel option as described in 1.3.4, the hugetlb filesystem must be mounted to be accessed, specifying the huge page size.

Then we have to configure the system. First, the use of transparent huge pages is disabled to allow an independent evaluation of the hugetlb subsystem. Then, in our case huge page size is 2 MB and the huge page pool size is configured to be two to provide enough space for reserving 4 MB of memory for the block, as defined in the test code. Figure 6.3 includes the list of the executed commands.

In our case, the number of hugepages allocated in the pool is determined by the number of pages enough to cover the allocation of all the blocks in the buffer. Hence, the number of the available hugepages has to be greater or equal than the multiplication between the number of blocks and the number of pages needed for each block.

6.2.4 Transparent huge pages tuning

In order to run our test for the allocation using THP, we assume to have compiled the kernel enabling the `CONFIG_TRANSHUGE` option.

The page size used by the THP subsystem can be read thought `sysfs` in the read-only file */sys/kernel/mm/transparent_hugepage/hpage_pmd_size*. In our case, the page

size is 2M for THP too.

To be sure that the kernel thread `khugepaged` will try to allocate transparent huge pages insted base pages, we need to force it enabling *always* the allocation mode (Figure 6.4).

```
echo always > /sys/kernel/mm/transparent_hugepage/enable
```

Figure 6.4: Enabling the use of the Transparent Huge Pages.

6.3 Test architecture

6.3.1 Synthetic network traffic with `pktgen`

A part of the tests were run capturing packets coming from a synthetic network traffic. Linux provides the packet generator **pktgen** [14] as a module or an in-kernel feature based on the `CONFIG_NET_PKTGEN` kernel option.

When `pktgen` is running, the module creates a kernel thread for each CPU X with affinity set to that CPU, called `kpktgend_X`. The `proc` filesystem is the interface for configuring and controlling the packet generation defining the network device to use, which sending rate or packet length and other details.

A synthetic packet generator has some limitations concerning the fact that it provides a **simulated** traffic that is a mostly regular one. So, we configure some characteristics to randomize the generated traffic: the size of each packet can vary in a range of sizes; the UDP, IP and MAC addresses and ports can vary, partially changing the content of each packet.

But the flow generated is not completely representative of a real test case.

The Figure 6.5 shows the script for configuring the generator running as a single thread (`kpktgend_X`). The thread is affine to the CPU X for sending traffic to the device specified in *interface*.

```
echo add_device <interface> > /proc/net/pktgen/kpktgend_X
echo min_pkt_size 700 > /proc/net/pktgen/<interface>
echo max_pkt_size 1500 > /proc/net/pktgen/<interface>
echo count 1000000000 > /proc/net/pktgen/<interface>
echo src_min 222.222.222.1 > /proc/net/pktgen/<interface>
echo src_max 222.222.222.255 > /proc/net/pktgen/<interface>
echo dst_mac 11:11:11:11:11:11 > /proc/net/pktgen/<interface>
echo src_mac 11:11:11:11:11:11 > /proc/net/pktgen/<interface>
echo flag IPSRC_RND > /proc/net/pktgen/<interface>
echo flag IPDST_RND > /proc/net/pktgen/<interface>
echo flag UDPSRC_RND > /proc/net/pktgen/<interface>
echo flag UDPDST_RND > /proc/net/pktgen/<interface>
echo flag MACSRC_RND > /proc/net/pktgen/<interface>
echo flag MACDST_RND > /proc/net/pktgen/<interface>
echo start > /proc/net/pktgen/pgctrl
```

Figure 6.5: Configuring and starting `pktgen` on the specified interface.

The thread will run until it generates a total number of packets given by the count value. The setting for transmission rate is omitted to enable the generator to transmit as fast as possible. The rate will be measured at the receiver side with the `nload` tool.

The **nload** application [24] allows to monitor real-time network traffic and bandwidth usage on the network devices, both incoming and outcoming packets. In particular, we need it in order to measure the rate of (Giga)bytes received per seconds.

Finally, after the configuration the generator can be started with `pgctrl`.

6.3.2 Network traffic with `tcpreplay`

A further step in this phase of testing is evaluating the performance by submitting the system to a flux of packets equal for all the tests run.

Replaying network traffic is done via `tcpreplay` tool: given a `pcap` file containing a trace of packets of a network communication, it can send data on a configured interface, at a certain rate and for a number of times. An example of use is shown in Figure 6.6:

```
tcpreplay -i <interface> -tK trace.pcap
```

Figure 6.6: Replaying network traffic from `pcap` files

where also in this case it is specified the option to enable the top speed (`-tK`) to

transmit as fast as possible.

To generate the trace, the `tcpdump` tool allow to dump the packets received from an interface to a `pcap` file. This is exactly what we need. We have the possibility to submit the same traffic to all the systems to test in order to make robust and comparable our results.

6.3.3 Low-level capture application

As described in 2.3.3, the application for packet capture is implemented following the main steps in the Figure 6.7.

```
fd = socket(AF_PACKET, SOCK_RAW, htons(ETH_P_ALL));
... /* setup */
setsockopt(fd, SOL_PACKET, PACKET_VERSION, &v, sizeof(v));
setsockopt(fd, SOL_PACKET, option, &req, sizeof(req));
...
map = mmap(NULL, req.tp_block_size * ring->req.tp_block_nr,
                         PROT_READ | PROT_WRITE, MAP_SHARED | MAP_LOCKED, fd,
                         0);
...
poll(&pfd, 1, 1000);
... /* release */
munmap(map, req.tp_block_size * req.tp_block_nr);
close(fd);
```

Figure 6.7: The low-level capture process main steps.

On the other hand, when the allocation policy is the one using HugeTLB, the application is implemented following the main steps in the Figure 6.8.

First, the creation of the socket of type `SOCK_RAW` in the `AF_PACKET` domain is done. Then the settings for `TPACKET_V3` and type of ring allocation. The total size of the ring buffer is computed as the product of the number of blocks and the respective size and it is memory mapped to the user process. Until a `SIGINT` signal is caught, the application cycles on the blocking function of polling reading data when available on the socket file descriptor. Finally, when the signal is received, the shutdown of the packet socket deallocates the resources and destroy the socket.

```
fd = socket(AF_PACKET, SOCK_RAW, htons(ETH_P_ALL));
... /* setup */
setsockopt(fd, SOL_PACKET, PACKET_VERSION, &v, sizeof(v));

hugefd = open(filename, O_CREAT | O_RDWR, 0755);

shm_hugepages = mmap(NULL, size, PROT_READ | PROT_WRITE, MAP_SHARED, hugefd, 0);

setsockopt(fd, SOL_PACKET, PACKET_HUGETLB_ENABLE, &addr, sizeof(addr));
setsockopt(fd, SOL_PACKET, PACKET_HUGETLB_RING, &req, sizeof(req));
...
poll(&pfd, 1, 1000);
... /* release */
munmap(shm_hugepages, req.tp_block_size * req.tp_block_nr);
close(fd);
close(hugefd);
unlink(hugefd);
```

Figure 6.8: The low-level capture process main steps for HugeTLB allocation policy.

In order to implement a likely capture process, it is introduced a base overhead at each packet reception, consisting in computing a non-cryptographic **hash** function on the bytes of the incoming data. As it happens in the analysis traffic application, this assures to take into account at least to read the content of the packet.

Furthermore, in terms of traffic analysis it is important to consider a longer processing time when it is captured a packet of interest. We implemented this perspective introducing a parameterized **factor of delay** every a defined number of packets (`FREQ_DELAY`).

The factor represents the number of times a fixed unit of delay is repeated. The unit of delay is a block of code where 1000 assembly multiplications are performed.

Our tests will take into exam the trend that the number of dropped packets follows varying with the factor of delay in order to analyze how the ring buffer behaves increasing the overload on it.

Going into the implementation detail of the capture phase, the first operation is to get the pointer to the header of the first block to access to the the relating status, because the next operation to be performed depends from it:

- if the block is not controlled by user space (`TP_STATUS_USER`) then begin the wait indefinitely (negative value of the timeout) until an event on the socket file descriptor occurs

- otherwise, the kernel has released the block and it is possible to access to the data in the buffer (Figure 6.9) and for each one:

  1. updating counters of the bytes of the packet
  2. delaying the next operations if the packet is that one that occurs when it is defined in `FREQ_DELAY`
  3. computing the hash function on the packet to scan every byte at least once

After reading the block, the status is updated to release the block and set to `TP_STATUS_KERNEL`.

All these steps are repeated continuously on all the blocks in the circular buffer.

```
static void walk_block(struct block_desc *pbd, const int block_num,
                long delay)
{
        int num_pkts = pbd->h1.num_pkts, i;
        unsigned long bytes = 0;
        struct tpacket3_hdr *ppd;

        ppd = (struct tpacket3_hdr *)((uint8_t *) pbd +
                                      pbd->h1.offset_to_first_pkt);
        for (i = 0; i < num_pkts; ++i) {
                bytes += ppd->tp_snaplen;

                if (!((packets_total + i) % FREQ_DELAY) &&
                                packets_total)
                        delayPacket(delay);

                hash(ppd, block_num);

                ppd = (struct tpacket3_hdr *)((uint8_t *) ppd +
                                              ppd->tp_next_offset);
        }

        packets_total += num_pkts;
        bytes_total += bytes;
}
```

Figure 6.9: The low-level capture process in a single block.

Therefore, when the process capture is interrupted by a `SIGINT`, statistics about incoming packets are collected. Using `getsockopt` with the `PACKET_STATISTICS` option, we known the number of packets and bytes received and the number of dropped packets.

6.3.4 Capture application using `libpcap`

As an alternative to the application just described, the same process is implemented using the `pcap` library as interface with the raw socket utilities. The main reason to write an equivalent application resides in the fact that it is very important to test the compatibility and the integration of the kernel patch with the library, as discussed in the section 4.2.

The implementation (Figure 6.10) follows the standard usage of the `pcap` APIs for packet capture, except for the need to set the allocation option specified as input as an environment variable. Thus, it is necessary to use the preloaded functions described in 5.4 for making available the new options.

```
setenv(OPTION, option, 1);

p = pcap_create(dev, errbuf);

status = pcap_set_buffer_size(p, 64*4*1024*1024);
status = pcap_set_promisc(p, 1);
status = pcap_activate(p);

while (!sigint) {
        packet = pcap_next(p, &hdr);
        if (packet)
                receive(packet, &hdr, delay);
}
pcap_stats(p, &stats);
```

Figure 6.10: The pcap capture process.

The polling activity to get the next incoming packet is performed with the `pcap_next()`.

About the processing of each incoming packet, the same operations that are per-

formed in the low-level application are implemented in this one too: the hash function is applied on the whole packet and the delay is repeated for a number of times that is specified as the delay factor value of the initial parameter. This packet operations are performed in the function `receive()` (again Figure 6.10).

Finally, the socket statistics about received packets/bytes and the number of dropped packets are obtained using the API `pcap_stats`.

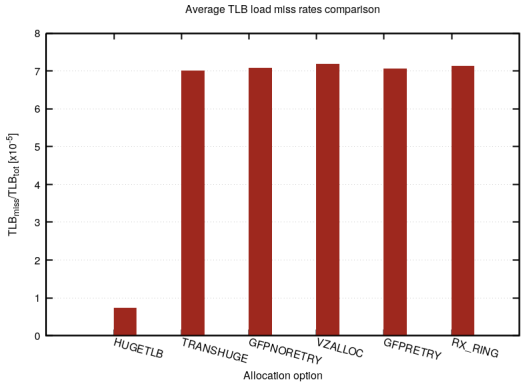
6.4 Results

6.4.1 Test type: randomized synthethic generator

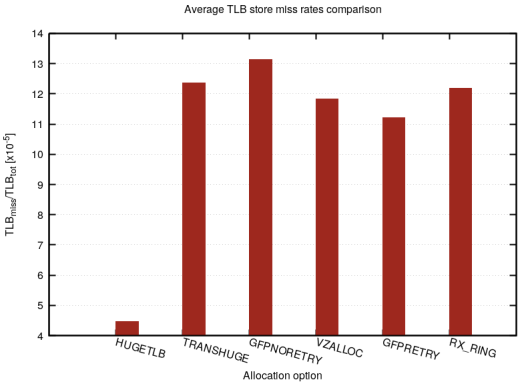
To perform the first type of test, we use a synthetic packet generator (`pktgen`) configured as shown in Figure 6.5. This configuration includes some features of traffic variability to get closer to a real network traffic:

- variable packet size: packet size changes in the range that goes from `min_pkt_size` to `max_pkt_size`
- variable IP address: packet IP address changes in the range that goes from `src_min` and `src_max`
- random addresses values: specified flags for random IP addresses (`IPSRC_RND`, `IPDST_RND`), MAC addresses (`MACSRC_RND`, `MACDST_RND`) and UDP ports (`UDPSRC_RND`, `UDPDST_RND`)

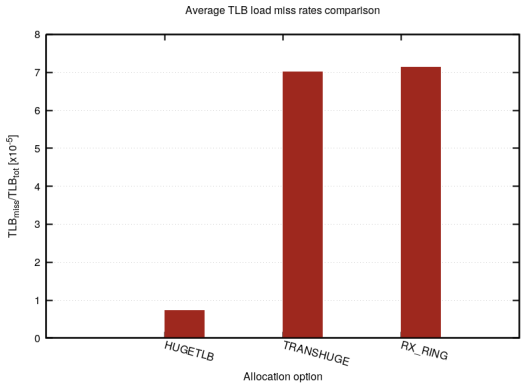
In Figure 6.11a, we compared the TLB miss rates for load operations, computed as the number of the recorded TLB load misses and the total number of TLB load accesses. Our interest goes to the load operations, because they are involved in the test application. But for testing purposes we keep also the store operation measurements.



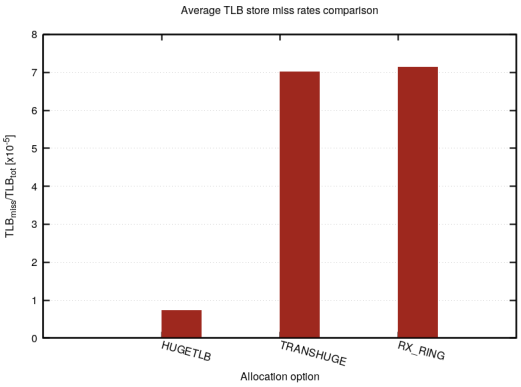
(a) TLB load miss rates.



(b) TLB store miss rates.



(a) TLB load miss rates.



(b) TLB store miss rates.

In the Figures 6.11a and 6.11b are compared all the allocation options in terms of TLB miss rates.

In the Figures 6.12a and 6.12b are shown the measures relating to the three main solutions: `PACKET_HUGETLB_RING`, `PACKET_TRANHUGE_RING`, `PACKET_RX_RING`. The limited comparison highlight that the solution using the HugeTLB filesystem is the most performing in terms of TLB miss rates. The HugeTLB solution gains a order of magnitude compared to the other ones. Both for loads and store operations.

In order to study the speed that is reached consuming the incoming packets, it is useful to know which amount of packet loss is registered and how it changes. The packet loss is the effect due to the fact that the ring buffer full. There is a value

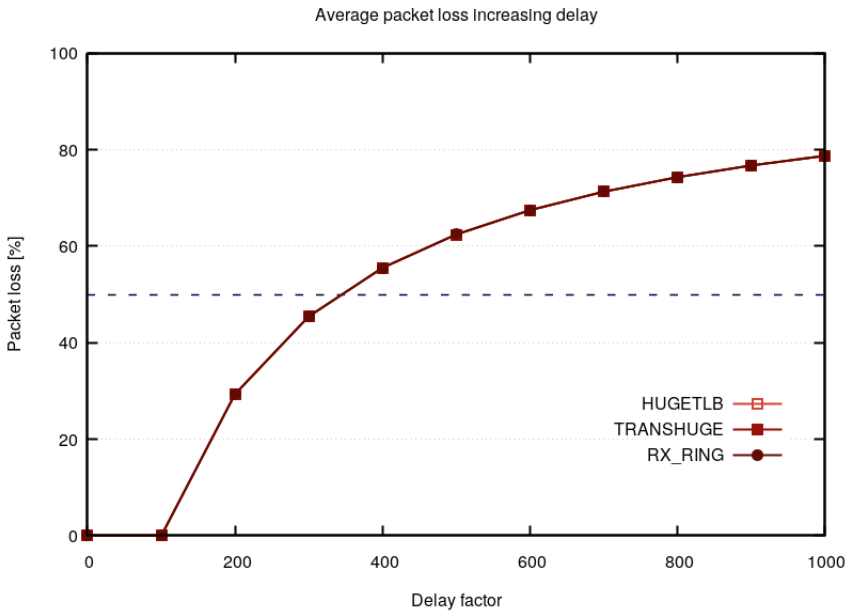


Figure 6.13: Packet loss trend increasing the delay factor per packet.

of delay factor when the driver begins to discard the incoming packets. The packet loss increases when the delay factor increases. This happens because growing the processing time per packet due to the delay factor, the packet is not removed from the buffer that keeps on filling util full.

As seen in 6.11a, from the point of view of the TLB misses there is a significant improvement given by the HugeTLB filesystem solution. But the gain does not express an equivalent reward concerning the packet loss trend.

Indeed, the lines plotted in the Figure 6.13 shows no meaningful difference among the three solutions. The plot shows the percentage of packet loss increasing the delay factor. The packet loss percentage is deduced from the difference between the number of packets received by kernel side and the number of packets analysed by the user.

Chapter 7

Conclusions and future developments

We started this study inspirated by the current solutions for Linux high-speed packet capture. They concern frameworks that are able to optimize the packet reception to face with high traffic rates.

We computed the impact of the adjustment of an existing Linux kernel networking domain (`AF_PACKET`) based on the choices implemented in the mentionated frameworks. The main feature that was exploited is the use of memory huge pages to achieve a performance improvement.

Our performance tests concerned packet capture at 10 GBit/s, submitting a variable synthetic traffic. The results shows a partial success. As expected, using buffers backed with huge pages translates into a decrease of the TLB miss rates compared to the original implementation. However, the improvement does not impact so much to affect the percentage of packet loss during the capture process.

The inspiring frameworks for high-speed traffic creates an own domain. Otherwise, in this case the attempt to change an existing domain does not came across such an high-performance guarantee, probably suffering a pre-designed subsystem.

In the best case, our type of tests are not the best suitable ones for the software.

Using other words, there can be a type of traffic or packet analysis that can exploit this solution.

However, the implemented solution is working and registers some kind of improvement. Future works could evaluate if the gain obtained in terms of TLB misses can translates into facilitation of contemporary applications running on the same CPU.

List of Figures

Bibliography

[1] Greg Kroah-Hartman, *"Linux kernel in a nutshell"*, O'Reilly, 2006

[2] W. Richard Stevens - Bill Fenner - Andrew M. Rudoff, *"UNIX Networking programming"*, Addison Wesley, 2003

[3] Alessandro Rubini - Jonathan Corbet, *"Linux Device Drivers"*, O'Reilly, 2001

[4] Daniel P. Bovet - Marco Cesati, *"Understanding the Linux kernel"*, O'Reilly, 2005

[5] Abraham Silberschatz - Peter B. Galvin - Greg Gagne,, *"Operating systems concepts"*, Wiley, 2012

[6] *https://lwn.net/Articles/*, Eklektix, Inc., 2018

[7] *https://man7.org/linux/man-pages/man7/packet.7.html*, Michael Kerrisk, 2018

[8] *https://man7.org/linux/man-pages/man7/raw.7.html*, Michael Kerrisk, 2018

[9] *https://man7.org/linux/man-pages/man3/dlsym.3.html*, Michael Kerrisk, 2018

[10] *https://man7.org/linux/man-pages/man2/socket.2.html*, Michael Kerrisk, 2018

[11] *https://kernel.org/doc/Documentation/networking/packetmmap.txt*, Linux Kernel Organization, Inc., 2018

[12] *https://kernel.org/doc/Documentation/vm/hugetlbfs.txt*, Linux Kernel Organization, Inc., 2018

[13] *https://kernel.org/doc/Documentation/vm/transhuge.tst*, Linux Kernel Organization, Inc., 2018

[14] *https://kernel.org/doc/Documentation/networking/pktgen.txt*, Linux Kernel Organization, Inc., 2018

[15] *https://kernel.org/doc/Documentation/kbuild/kconfig-language.txt*, Linux Kernel Organization, Inc., 2018

[16] *http://www.tcpdump.org/manpages/pcap.3pcap.html*, Van Jacobson - Craig Leres - Steven McCanne, 2018

[17] Josè L. Garcìa-Dorado - Felipe Mata - Javier Ramos - Pedro M. Santiago del Rìo - Victor Moreno - Javier Aracil, *High-Performance Network Traffic Processing Systems Using Commodity Hardware*, Springer Berlin Heidelberg, 2013

[18] *https://sites.google.com/site/packetmmap/*

[19] *https://www.securitynewspaper.com/2017/05/12/exploiting-linux-kernel-via-packet-sockets/*

[20] *https://www.kernel.org/doc/gorman/html/understand/index.html*

[21] *https://perf.wiki.kernel.org/index.php/Main_Page*

[22] *http://man7.org/linux/man-pages/man1/oprofile.1.html*

[23] *https://lwn.net/Articles/379748*

[24] *https://linux.die.net/man/1/nload*