HIGH PERFORMANCE COMPUTING AND NETWORKING RESEARCH GROUP

Universidad Autónoma de Madrid
Escuela Politécnica Superior

HPCAP

Developer guide

Abstract

The HPCAP Developer Guide has all the necessary information for developers and maintainers of the HPCAP driver and client applications. It includes a description of inner workings of the driver, how the code is structured and the auxiliar tools created to support the development and the driver.

Guillermo Julián Moreno
guillermo.julian@estudiante.uam.es

November 1, 2016

Contents

Chapter 1

Introduction, basic information and practices

The HPCAP driver is a modification of the Intel IXGBE driver that allows capture of traffic at high rates. This section acts as a basic introduction to the driver code and development practices.

This document, along with the auto-generated code manuals, the user guide and Víctor Moreno's master's thesis [1], conform the documentation of the HPCAP driver.

1.1 Folder structure

The HPCAP driver code tree is structured as follows:

- *Doxyfile* The instruction file that generates Doxygen documentation. See section 1.5 for details.
- *Makefile* The file with instructions for the build system. See section 2.
- *Makefile-driver-vars.mk* Specific Makefile code to set the driver build configuration.
- *doc* The folder that contains the documentation for the HPCAP driver. The folders *doc/latex*, *doc/html* and *doc/man* contain the autogenerated Doxygen documentation in each respective format.
- *driver* Contains the source files for the driver. This folder contains three subdirectories:
  - *hpcap_ixgbe-4.1.2* IXGBE driver files.
  - *hpcap_ixgbevf-2.14.2* IXGBE driver files - virtual mod.
  - *common* Common files for both drivers (i.e., HPCAP-related files)
- *include* C header files.
- *install_hpcap.bash* Driver install script.
- *params.cfg* Driver configuration file.
- *samples* Sample applications. See section 5.1.
- *scripts* Helper scripts.
- *srclib* Library source codes. See section 5.2.
- *bin* Binary output folder. Samples binaries and modules will be stored here.
- *obj* Output folder for the compilation intermediate products.

1.2 Version control

The version control system used by HPCAP is SVN. The branching strategy is discussed below. To aid maintenance and development, the developer should follow usual recommendations regarding commit messages. Basically, ensure that commit messages describe what was done and why, not how (we already have the diff to see how it was done).

It's also helpful to include the "category" of the commit in the subject line, such as "hpcap: Fix reset of listener's buffer size" or "lib: Refactor listener operations in a common function". As HPCAP has several different parts, this helps to understand what has been changed without too much explaining (e.g., it's not the same to change listener code in the hpcap driver than in the library, and the prefix "hpcap:" or "lib:" takes care of that) and also helps when reading the commit log.

Commits should also be atomic, that is, one commit should reflect one change in the codebase. This helps a lot when reverting specific features or when searching where was a bug introduced.

1.3 Version numbers

From version 4.2.0 onwards, HPCAP follows the semver semantic versioning scheme. This means that version numbers are of the form *MAJOR.MINOR.PATCH*.

New versions should increment one of the tree numbers:

- *MAJOR* version when the API changes in incompatible ways.
- *MINOR* version when new functionality is added in backwards-compatible manner.
- *PATCH* version when there are new backwards-compatible bugfixes.

1.4 Code formatting

The HPCAP code is formatted using the Linux coding style guide, using tabs for indentation. To make formatting easier, the file *.astylerc* contains the options to format the code with the astyle tool. You can format manually the code executing `astyle --options=.astylerc -R "./*.c" "./*.h"`, or you can integrate this options file with editors that support automatic formatting, such as Sublime Text's AStyleFormatter or Vim's Autoformat.

1.5 Doxygen documentation

This document contains general documentation regarding the driver and related software. For more specific documentation about the code, Doxygen-style comments are used. The advantage of this comment style is that developer-friendly documentation can be generated automatically. The file *Doxyfile* controls this documentation output. Currently, HTML, LaTeX and Man versions of the documentation are generated. To explore the generated manpages, you should change your `MANPATH` variable running the command `export MANPATH=$MANPATH:hpcap-path/doc/man`. Alternatively, you can run `sudo make install` to install everything to your system, including the manpages.

Chapter 2

Build and installation system

The build system used for HPCAP is, as always, GNU Make. Apart from the traditional `all` target that builds everything, there are three other targets:

- `samples`. These are applications built for this driver, with the purpose of either aiding development and debugging or as simple utilities, such as the hugepage mapper or the *hpcapdd* dumper. The samples are described in section 5.1.
- `libs`. To allow using the HPCAP driver from userspace applications, two libraries act as layers between them and the driver. These are described in section 5.2.
- `drivers`. The *hpcap* and *hpcapvf* drivers, no need to say what are they. The targets `drivers-release` and `drivers-debug` may be used to create debug or release versions of the drivers.
- `install` and `uninstall`. These targets respectively install and uninstall the HPCAP from the driver. The installation procedure is explained in section 2.3.

To build all the targets, there is a Makefile placed in the root of the code tree. It builds the samples, libraries and drivers detecting automatically the dependencies. In order for it to work, it needs a strict folder structure as described in section 1.1.

The restrictions are the following:

1. Each sample should be in its own directory inside of *samples*. If there are multiple samples in one directory (that is, more than one file with a `main` function) compilation will fail.
2. Library header files should be kept in the *include* directory.
3. Library code files should be kept in the *srclib/libname* directory.
4. Kernel drivers must be placed in the *driver* directory. Each driver should be named *hpcap_ixgbe[extra]-ignored*. Valid folder names are, for example, *hpcap_ixgbe-3.7.17_buffer* or *hpcap_ixgbevf-2.14.2*. The driver files will be named based on the folder name: *hpcap[extra].ko*.

The Makefile will generate binaries and libraries automatically in the *bin* and *lib* directories respectively.

2.1 Build configurations

The build system supports multiple build configurations. A configuration is just a set of compiler flags and options. Currently, there are two different configurations:

- **debug**. This configuration builds the files with debugging symbols included, no optimizations and a `DEBUG` macro defined to enable debug-specific regions in the code.
- **release**. This is a configuration targeted for production binaries. All optimizations are enabled, the targets are stripped of all unnecessary information (such as debugging symbols) and debug-specific code regions are deactivated.

Each configuration has a Makefile rule with its name that will build both the samples and libraries with that configuration. For example, running `make debug` will build debug versions of everything.

2.2 Driver build system

In order to build the kernel drivers with separate configurations, a workaround was needed in the build system. As documented in the kernel, the Linux module build system must be used to build the modules, calling the Kbuild makefiles inside the Linux kernel code tree.

The Kbuild system doesn't support the modification of the output path: it is the same as that of the source code files. To bypass this limitation, the whole code of the drivers is automatically copied to subfolders inside the *obj/kernel* directory. A Makefile is automatically generated so Kbuild knows how to build the driver. The flags are automatically changed depending on the configuration used.

It's not a perfect system (code is copied one time per configuration) but at least works and allows the use of different configurations.

The Makefile contains a more detailed description of how this is achieved.

2.2.1 Changing build parameters

The build parameters of the driver can be changed in the first lines of the *Makefile*. This includes a parameter (`BUILD_KERNEL`) for building the driver against an specific kernel version (ensure that you have the corresponding headers installed).

2.2.2 Optional/feature dependent files

The base *ixgbe* driver includes some code files depending on the available features of the kernel. The *Makefile* and related scripts are adapted to manage these situations.

Files that are always excluded from the build are in the variable `PRE_EXCLUDED_CFILES` in the *Makefile-driver-vars.mk* file. In this same file, depending on the system and wanted configuration, some files are added to the build, stored in the variable `EXTRA_CFILES`. For this to work correctly, the optional files must be defined in the *scripts/mkmakefile* script (the variable name is `OPTIONAL_CFILES`) that creates the *Makefile* that will be later used by the kernel build system.

2.3 Installing the driver to the system

The HPCAP driver can be installed to the system to avoid running everything from the source folder. The rule `make install` will create all necessary folders and files. The *hpcap.ko* and *hpcapvf.ko* will be installed to */lib/modules/your-kernel-version*, and the *modprobe* configuration will be modified so running `modprobe hpcap/hpcapvf` will install the *hpcap* / *hpcapvf* driver and create the configured interfaces. The configuration file will be placed in */etc/hpcap/params.cfg*, the libraries and headers in the standard POSIX directories (that is, */usr/lib* and */usr/include/hpcap*). Additionally, some binaries and scripts will be placed in */usr/bin*, such as `raw2pcap`, `hugepage_mount`[1] and `huge_map`[2].

The Makefile supports the installation to a custom prefix. That is, if you want the driver to be installed to */home/myuser/*, you can run `make install INSTALL_PATH=/home/myuser`. However, it is not guaranteed that modprobe and related kernel utilities will work with custom prefixes, so ensure you know what you're doing before using a custom installation prefix.

The scripts (except the ones related to modprobe) are ready to run from either the source folder or from their installation place. They source the bash library functions from the *hpcap-lib.bash* file[3] and probe for the configuration file either in the root of the source folder or in the */etc/hpcap/params.cfg* location.

2.3.1 Installing only the libraries for development

When developing programs that use the HPCAP API, it may be useful to install only the libraries and headers to the system. The Makefile target `install-libs` will install only the libraries to the system, without building nor installing the driver.

[1]Described in section 3.4.2.

[2]Described in section 5.1.1.

[3]The library should be placed either in the *scripts* folder or in */usr/bin*. This way, the `source` command will always find it independently of where the script is running from.

Chapter 3

Kernel driver

3.1 Architecture

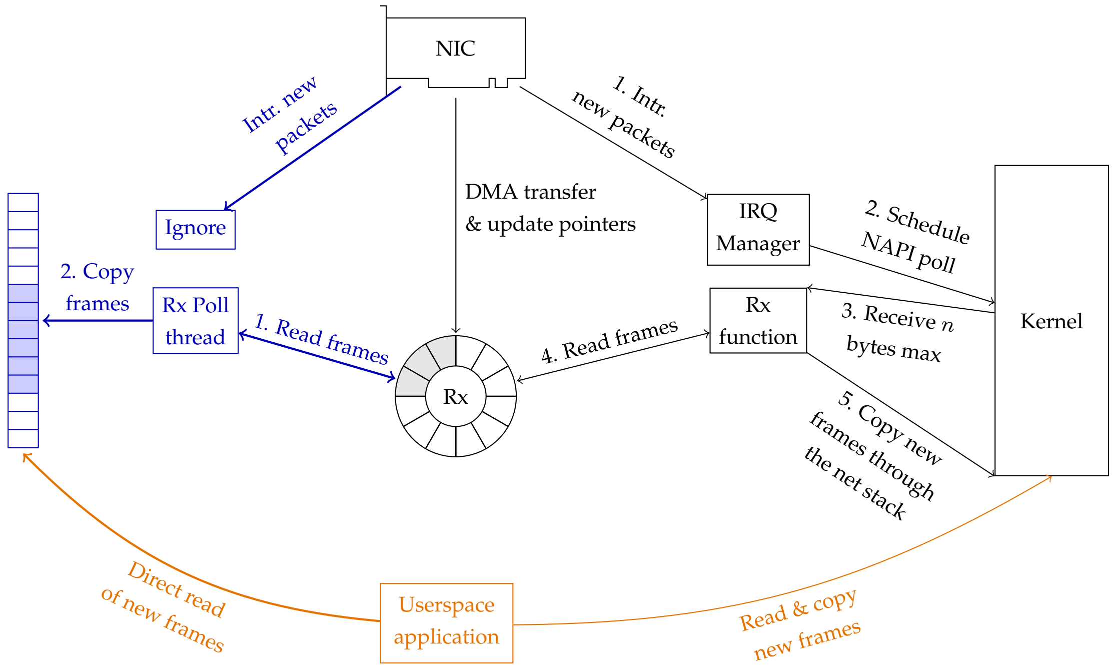

Figure 3.1: Architecture of the *ixgbe* driver (black) and HPCAP (blue), showing the different paths that each use to read frames from the reception ring filled by the network interface card (NIC).

Although Víctor Moreno's Master's thesis [1] already covers the architecture of the HPCAP driver, the main concepts behind it are explained here.

Usually, the manufacturer's driver (*ixgbe*) works via NAPI, a mixed push/poll approach. When the NIC copies new packets to the host memory via DMA, it sends an interrupt that is managed by the driver, which in turn notifies the NAPI subsystem[1]. Then, when the kernel is ready, launches the poll function from *ixgbe* with a certain budget n. This function will read up to n bytes from the Rx ring, filled by the NIC asynchonously via DMA, and will send the corresponding packets through the kernel's network stack, so they will end up reaching userspace applications.

However, given the motivation of the HPCAP driver (capture at high rates), most of those steps can be bypassed to allow higher performance. First of all, interrupts are disabled and/or completely ignored. Instead, a poll thread runs continously, checking for new packets in the Rx ring and copying them into a bigger buffer (usually, 1GB or more). When a userspace applications wants to read those frames (either to store them to disk or for further processing) it uses the HPCAP API, which will provide direct read access to that bigger buffer.

This architecture is simpler and avoids an unnecessary copy (from the network stack to the userspace application's buffer). It also avoids the extra latency present in NAPI polling systems, which is very useful to have accurate timestamps.

[1] "New API", a extension in the Linux Kernel designed to improve performance by mitigating interrupts. Documentation is available online.

3.2 Driver execution flow

To serve as an introduction to the driver, we explain how does it work from the moment of initialization.

3.2.1 Initialization

1. The *install_hpcap.bash* script installs the driver with *insmod*, with the corresponding arguments as configured in the *params.cfg* file.
2. The kernel executes the entry point `ixgbe_init_module`, in the file *ixgbe_main.c*. In this function, HPCAP code ensures that the options are correct (`hpcap_precheck_options`) and assigns the interface numbers (see section 3.6).
3. IXGBE registers itself as a PCI driver.
4. For each compatible PCI device found:
   a) The kernel calls `ixgbe_probe` with the PCI structure, so IXGBE can initialize the device.
   b) IXGBE prepares the network devices and other structures.
   c) HPCAP assigns to the device the corresponding fixed interface number.
   d) The device is configured with the corresponding arguments that were passed to the module with *insmod* (see `ixgbe_check_options`).
   e) HPCAP saves a reference to the IXGBE adapter structure in the `adapters` array, in the file *driver_hpcap.c*.
   f) IXGBE finalizes the initialization of the device (hardware init, network device registration, etc).
5. After IXGBE has finished with the initialization of all devices, `hpcap_register_adapters` (file *driver_hpcap.c*) is called. For each device working in HPCAP mode, the function `hpcap_register_chardev` is called. This function does the following:
   a) Creates the `hpcap_buf` structure and saves it in the IXGBE device structure.
   b) Registers the character device (see section 3.3) with the system. This character device will be used for the communication with user-space.
6. After the module has been initialized, Linux wakes up the network interfaces, calling `ixgbe_open`. IXGBE prepares its private configuration, and then calls `hpcap_launch_poll_threads` if the interface is in HPCAP mode. This functions ensures that the interface is named correctly (see the corresponding bug description in section 3.8.4) and then launches the RX poll thread. This thread just copies the new buffers from the NIC ring to the internal HPCAP buffer (see figure 3.1) if there is a listener. See the ssubsection below for a description of traffic reception.
7. If there are hugepages configured, the *install_hpcap.bash* scripts allocates and registers them with the system. See section 3.4.

3.2.2 Traffic reception - kernel side

The traffic reception is divided in two sections: first, how does the driver receive the data and then how does a client application read it. All the functions for this are defined in *hpcap_rx.c*.

The entry point for the RX thread is `hpcap_poll`. This function loops continously until it is requested to stop. In each iteration:

1. Checks whether there are client applications reading the traffic. If there are no clients, the read/write offsets are reset (see bug 3.8.5) to avoid unaligned memory access.
2. Calls the reception function `hpcap_rx`. While the NIC ring is non-empty, it copies new frames to the internal HPCAP buffer. This function ends when the NIC ring is empty or the internal buffer is filled.
3. Updates the listeners write pointers to notify them of the new data available, calling `hpcap_push_all_listeners`.
4. Updates the global read offset according to the slowest listener. Thus the new global read offset will point to the last byte read by all listeners. All the bytes between the write offset and the read offset (take into account that the HPCAP buffer is circular) are ready to be written to with new data.

3.2.3 Traffic reception - userspace side

1. A client application, such as *hpcapdd* or *monitor_flujos*, is launched.

2. The client opens the HPCAP character device */dev/hpcap_N_Q* calling `hpcap_open` in *libhpcap.c*. This opens the character device with the standard Linux call `open`.
3. The kernel calls `hpcap_open` in *hpcap_cdev.c*, notifying the driver of the new client. HPCAP assigns it a fixed ID and registers the corresponding listener.
4. The client application maps the HPCAP buffer into its address space, calling `hpcap_map`, which in turns does an `ioctl` call that returns the necessary buffer information.
5. The client application calls in a loop to `hpcap_ack_wait_timeout`, acknowledging how many bytes as it read (this updates the listener read offsets) and updating the count of available bytes (updating its write offset). Then, it can process the new frames, reading them from the previously mapped buffer.
6. Once everything has finished, the client applications closes the HPCAP handle to allow the driver to free the allocated resources and structures.

3.3 Character device

As it has been mentioned in the previous section, HPCAP uses character devices to manage the communication between client applications and the driver. Each interface gets assigned a file */dev/hpcap_N_Q*, where N is the interface number and Q the queue number[2].

The use of a character device allows the driver to bypass the system's network stack, and gives client applications raw access to the buffer where all the new frames are copied. It also allows using arbitrary `ioctl` calls, easier to develop and manage, to control the driver. For example, the assignment of hugepages (section 3.4) is done via `ioctl` calls.

3.4 Hugepages

To improve performance, the HPCAP driver supports the use of huge memory pages: each adapter may replace its regular buffer by a *hugepage*-backed one. This allows the use of bigger buffers, so traffic peaks can be accommodated easily.

The applications do not need to change anything in order to use *hugepage*-backed buffers. The only required actions are done automatically by the installation script.

The next paragraphs describe the inner workings of the *hugepage*-backed buffer.

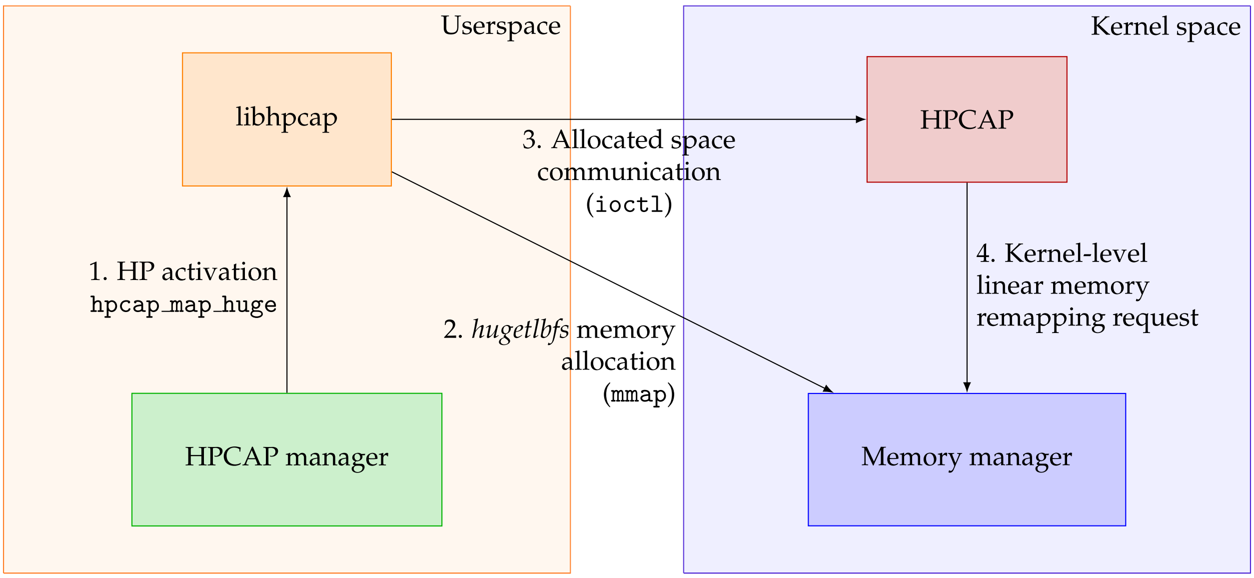

Figure 3.2: Schematic of the hugepage allocation and buffer assignment to the driver.

To use the driver with hugepages, first it should be activated calling the function `hpcap_map_huge`. The *huge_map* application, in the *samples* folder (see section 5.1.1), does this call easily from user-space. The install script `install_hpcap.bash` does this automatically when an interface is configured in mode 3 (although the mode is always 2 in the buffer).

[2]The queue number is usually 0, as the driver does not correctly support more than one queue.

Once the call to `hpcap_map_huge` is done, the *libhpcap* library takes control. The function `_hpcap_mmap_hugetlb` will get a memory region of the requested size backed by the system's hugepages. In order to do this, it will create a file in the *hugetlbfs*[3] filesystem and will call `mmap` to obtain the memory address.

Once the memory region pointer has been obtained, it will be transferred to the HPCAP driver by means of a `ioctl` call. The `ioctl` controller in the driver will call `_hpcap_use_huge`. This function will obtain from the kernel the actual pages that make up the buffer.

A small trick used when calculating the number of pages, is that the page size used is not 1GB (or whatever is the hugepage size in the system), but the default regular page size of the system (4KB usually). Despite this, the pages used will actually be *huge pages*.

The kernel function `get_user_pages` will be called with the page number calculated as explained. Here, a problem may occur. When the buffer size is greater than the *hugepage* size, the pages returned may be unordered and memory addresses will not be sequential[4]. To solve this, `vm_map_ram` is called, obtaining a virtual address that can be accessed sequentially.

Once the buffer address is ready, the buffer of the given adapter is replaced by the *hugepage*-backed one. Additionally, certain parameters are saved in the adapter structure in order to manage and free the buffer later.

3.4.1 Releasing *hugepage* resources

There are several parts of the system where references to the *hugepages* buffer are kept:

- Kernel driver.
- Userspace applications that allocated the buffer and transferred it to the driver.
- Userspace applications that map the adapter memory.
- *hugetlbfs* filesystem.

In order for the memory to be freed, no references should be left.

The application allocating the *hugepages* buffer loses the references once it is closed. The applications calling `mmap` lose them in the same way, although the handles can be released by calling `munmap`.

The driver releases the references once the adapters are unregistered, which can happen on module unloading or when changing the adapter configuration.

The references in the *hugetlbfs* filesystem are deleted when calling `hpcap_unmap_huge`. If they are still present, they can be released simply by deleting the corresponding files in the filesystem.

To see the statistics of the system's hugepages, you can use the `hp_info` function in the *hpcap-lib.bash* script. Just source that file in your shell (`source scripts/lib.bash`) and call `hp_info` to see the statistics. This function just extracts the information from the */proc/meminfo* file.

3.4.2 *hugetlbfs* filesystem

As explained in the kernel docs, an easy way to access and use *hugepages* is via the *hugetlbfs* filesystem. This is a virtual filesystem that can be accessed as any other part of the Linux folder tree. It is mounted with the *mount* application:

```
> mount -t hugetlbfs none /mnt/hugetlb
```

The *hugetlbfs* accepts several parameters, documented in the kernel docs:

```
> mount -t hugetlbfs -o uid=<value>,gid=<value>,mode=<value>,size=<value>,nr_inodes=<value>
none <hugetlfs_path>
```

Files created inside this filesystem are, from the user point of view, the same as every other file in the system. However, under the hood, these files are pointers to memory regions backed by *hugepages*. That means that if they are mapped with a call to `mmap`, what will be returned will be a pointer to a *hugepage*-backed memory buffer.

In order to use hugepages, your system must me configured properly at boot time. That means that you should add the corresponding parameters to the *linux* command in your *grub.cfg* file. These parameters are *default_hugepagesz*, *hugepagesz* and *hugepages*, as documented in the kernel documentation. For example, the parameters `default_hugepagesz=1G hugepagesz=1G hugepages=8` would enable hugepages of 1 GB, and then would allocate 8 hugepages of 1GB at boot time. In the end, your *linux* boot command could look like this:

```
linux /boot/vmlinuz-3.8.0-29-generic root=UUID=b2be13dc-7a60-4928-b61a-cc3ec95044a0 iommu=pt
intel_iommu=on isolcpus=0,1,2,3,4,5 ro default_hugepagesz=1G hugepagesz=1G hugepages=8
```

[3] See section 3.4.2 for a description of the *hugetlbfs* filesystem.

[4] See this StackOverflow question for further discussion of the problem and solution.

To make the hugepage mounting process easier, the script *hugepage_mount* in the *scripts* directory will mount the *hugetlbfs* filesystem in the default path */mnt/hugetlb* if no *hugetlbfs* filesystem is already mounted. It will also detect when the system does not have hugepages activated.

3.5 `ioctl` interface

To allow communication between client (userspace) applications and the driver, the `ioctl` system call is used.

For listener operations (acknowledging read bytes or waiting for new data to arrive), the `ioctl` code used is `HPCAP_IOC_LSTOP`, that sends a `hpcap_listener_op` structure with all the data. The driver will receive that structure and do the necessary operations. Refer to the code in *hpcap_cdev.c* for details.

For buffer operations (buffer information and hugepage mapping (section 3.4)), the structure used is `hpcap_buffer_info`. The operation to perform will be decided depending on the `ioctl` command used. The relevant operations are `HPCAP_IOC_BUFINFO`, `HPCAP_IOC_HUGE_MAP` and `HPCAP_IOC_HUGE_UNMAP`. Again, refer to the code in *hpcap_cdev.c* for details on how are these commands processed.

3.5.1 `ioctl` version compatibility

When changing the structures and `ioctl` commands, client applications compiled for previous versions may fail and, what's worse, do it without a easy-to-understand error message. To avoid this problem, apart from documentation and version number changes[5], the magic number (`HPCAP_IOC_MAGIC`) should be updated to stop old versions from issuing commands that the new driver will see as corrupted.

The other option (adding a version check field to the structure) was actually implemented in commit 4aba154098fe5d12019d4c65e7be477a98ad8735, but reverted later after it was noticed that increasing the magic number would have the same results, better reliablity and less complexity. If you want to go back to the version check field, just revert the revert commit (dc01554a5220d32d947ab9eedead03736d526805).

3.6 Interface numbering

The HPCAP interfaces receive a fixed name based on the PCI bus ID, to avoid renamings after a reboot of the monitoring system. All the functions related to this capability are in the file *hpcap_pci.c*.

To achieve this fixed naming, the *ixgbe* driver first calls `hpcap_fix_iface_numbers` with the list of recognized PCI devices. This function searches all devices matching those PCI IDs (that is, all the supported NICs) and assigns them a fixed number. After initialization, HPCAP will call to the other functions in the file to retrieve the fixed numbers assigned.

3.7 Debugging & testing

In this section we will describe several techniques to debug and test the HPCAP driver. Take into account that there are no debuggers like *gdb* (or at least they're not that useful).

3.7.1 Traffic generation

Usually, the first issue that appears when testing the driver is the need for generating traffic at high rates. There are several solutions, most based on Intel DPDK (Data Plane Development Kit).

To install DPDK, you should download it and uncompress it, and then follow these steps:

1. In the *dpdk* folder, go to *tools* and run *setup.sh* as root. A menu will appear.
2. Select the appropiate build option for your architecture (probably *x86_64-native-linuxapp-gcc*).
3. Set up the hugepage mappings for your architecture (NUMA/non-NUMA). I usually choose 1024 pages, but it is a flexible parameter.
4. Insert the IGB UIO module (for Intel IXGBE NICs).
5. Bind the desired Ethernet devices to the IGB UIO module. The ethernet devices appear with the PCI bus number and the interface they're bound to. If you want to bind an active interface (marked with **active**), first bring it down with `ifconfig iface down`. If you are not sure which interface corresponds to each physical link, you can use `ethtool -p iface` to make the LED lights on the corresponding physical interface blink for easy identification.

[5] As section 1.3 explains, backwards-incompatible changes such as changing the `ioctl` commands and/or arguments should reflect in an increase in the major version number.

Now, the available traffic generators.

3.7.1.1 pktgen-DPDK

pktgen-DPDK is a packet generator based on DPDK, very flexible but sometimes not predictable and not too easy to start configure.

To build it, you should build DPDK as explained above (even if it's already built) and then, in the same terminal session, run `make` in the *pktgen-dpdk* folder. If DPDK is not built, some environment variables are not exported and the pktgen compilation will fail.

Once compiled, you can run it with a command line such as this one:

```
./app/app/x86_64-native-linuxapp-gcc/pktgen -c FF -n 4 -- -T -P -m "[1-7:1].0"
```

These arguments are separated on two sections: the part before the `--` is related to DPDK and its "EAL" options. The `-c FF` is the core mask, and selects the cores to use. In this case, `FF = 1111 1111`, that is, use all 8 cores. `FE = 1111 1110` would use only 7, although it is not clear which core (the first or the last one) is unselected. The `-n 4` is the number of memory channels to use. More gives better performance, but there's usually a limit on how many can be used.

For the *pktgen* options (the part after the `--`), `-T` gives colored output, `-P` enables promiscous mode and, the most important part, the `-m "[1-7].0"` defines the core assignment. In this case, it assigns cores 1 to 7 to the interface 0. Running `pktgen --help` will give more examples of the syntax, just remember to keep all the specifiers between quotes. A tip: the first core (0) is usually assigned to the display thread, so assigning it to an interface could make the program fail.

When the core assignment is not correct, *pktgen* does not show an error message, it just stops. However, in the middle of all the output it produces there's a table with the cores and ports: if it's empty, you know your core assignment was wrong.

Once *pktgen* starts, it has a nice interface where you can configure the generator. Typing `help` will describe all the available options, including changing the IP or MAC destination.

For a quick start, set the frame size with `set [port-number] size X` and the rate with `set [port-number] rate R`, where `R` is a percentage between 0 and 100: actual transmission rate will be the $R\%$ of the maximum rate of the port. Then, start the transmission with `start [port-number]`. In all of the previous commands, `port-number` can be *all* to act on all the available ports at the same time.

If you want to send PCAP files with *pktgen-DPDK*, you have to add arguments to the command line and then activate the PCAP sending in the interface[6]. The command line arguments should look like this: `-s I:path`, where `I` is the index of the port from which you want to send the PCAP, and `path` is the path of the PCAP file. Once *pktgen-DPDK* has started, activate the PCAP sending with `pcap [port-number] enable`.

3.7.1.2 MoonGen

MoonGen, also based on DPDK, is a packet generator specialized in scripting. It uses Lua as an scripting language, is fast and more predictable than *pktgen*, but less useful if you just want to do a quick test. The installation instructions on its page are complete.

This is the generator used for automated testing. There's a script in the HPCAP folder, *scripts/fixed-rate.lua*, which can generate traffic at a fixed rate with custom frame sizes. It can be used as a base for writing other scripts for MoonGen. Their API documentation is also decent.

3.7.2 ethtool counters

It's really easy to add new counters that will show up with *ethtool* when running `ethtool -S hpcapX`. In the file *ixgbe_ethtool.c*, modify the `ixgbe_gstrings_stats` and add new entries. These entries need to have a string identifying the counter, and the corresponding field in the `ixgbe_adapter` structure. You should only worry about modifying that field when appropiate: the *driver* will take care of transmitting that information to the *ethtool* system.

Currently, there are two extra counters in *ethtool*: `rx_hpcap_client_lost_frames` and `rx_hpcap_noclient_frames`. The first one marks the frames that were discarded because the intermediate buffer was full. The second one counts the frames discarded because there were no clients ready to receive them. Note that these counters are note 100% precise and may not count all the losses of the driver: it is expected behaviour, as the driver will exit the reception loop when it is not able to copy the packets to the intermediate buffer.

[6] And have a little bit of luck. We have not been able to test this feature as it requires a huge amount of memory (the allocations are not efficient at all).

3.7.3 Automated testing

In order to make testing easier, a script for automation is provided: *scripts/hpcap-test*. The test parameters (e.g., generator port, generator host, traffic rates or tests to execute) can be configured in the file *hpcap-test.cfg* (you can create it renaming *hpcap-test.cfg.sample*). All the options are documented in the configuration file.

The test script ensures that the driver, samples and libraries compile correctly, that the driver can be installed and removed from the system without problems and with a coherent status, and then starts with extra tests.

These extra tests are located in the *scripts/tests* folder. To add new tests, the only thing needed is to create another file in that folder, and to activate the corresponding test in the configuration file, adding an option `test_filename=true` in the *hpcap-test.cfg*, where `filename` is the name of the file that was just created in the *scripts/tests* folder.

The tests can use functions defined in the *scripts/hpcap-test* file, such as `reinstall_hpcap` or `install_hpcap`; `begin_test, test_ok, test_fail` to mark the beginning and results of a test and `test_fail_die` to mark the test as failed and aborting the session (for example, when the driver is corrupted or any other thing goes horribly wrong).

The functions in *scripts/hpcap-lib.bash* can also be used. Of special interest are `start_generator`, which receives as arguments the generation rate in Mbps and the frame size and starts the traffic generator in the background; and `stop_generator` whose result is obvious.

All of the configuration of the test (e.g. receiver interface, generator configuration) is read from the *hpcap-test.cfg* file, in bash-like format. Most options are documented in the file. All the variables are read and are available with the same name in all of the test scripts.

3.7.3.1 Test cases

Apart from the basic test cases described above, the following extra tests are available:

- **bad_padding**: Tests for a edge case in the padding insertion, where the space left in the current file is exactly the size of a RAW header. This test was written to ensure that bug 3.8.8 did not appear again.
- **small_buffer**: A test of reception with small-ish buffers, not using hugepages. This may discover issues with overflowing counters.
- **traffic_rx**: Tests that the driver can support a high rate of reception without losses. All the traffic is discarded by *hpcapdd*.
- **traffic_store**: Tests whether the driver can support a decent rate of traffic reception without losses while saving the traffic to disk. Also ensures that the generated RAW files are correct and there's no malformation.

All of the parametres of these tests can be configured in the same *hpcap-test.cfg* files.

3.7.4 *Post-mortem* debugging

When the driver crashes or stops working, there are several options to try and see what has happened. We will see them from best-case to worst-case.

The best case occurs when the driver does not work appropiately but it is still loaded in the system and responding to some commands. The kernel log (run `dmesg -H`) may show error messages that may or may not give some clues. If you want extra information about the driver, installing a debug build (`make debug; ./install_hpcap.bash bin/debug/hpcap.ko`) will show more messages (make sure of enabling the messages for the categories you need in *driver/common/hpcap_debug.h*, see next section 3.7.6) that may help.

The application *statusinfo* can also be used to show information about a HPCAP interface in the console, such as read/write offsets, connected listeners and thread states. It is pretty useful when the driver has stopped receiving traffic but there are no messages telling why, and a debug build cannot be installed.

The next case is when the driver stops working and crashes but does not crash the system. Usually, there's a backtrace in the kernel log that shows the address of the instruction that crashed, although in hexadecimal format (e.g., `hpcap_rx+0x32c/0x660`), which is not very useful. *addr2line* can be used to see the actual line in the file, but usually *gdb* is more reliable. Run `gdb bin/debug/hpcap.ko`, making sure that the binary is the same that was installed when the crash happened. Then, the command `l *(hpcap_rx+0x32c/0x660)` will show the exact file and line of the instruction that crashed the driver.

The worst case (and also the common one) is that the driver crashes and takes the system with it. If the *kdump* service is enabled, the memory image and the *dmesg* log are saved to */var/crash*, which can be useful to see what happened. See your distribution's instructions on how to enable *kdump* (e.g., this for RedHat based distros).

3.7.5 Diagnostic collection tool

To facilitate the diagnosis of failures and bugs in production environments, the script *scripts/diagnostics* can be used to collect information that can help to discover what happened. After collection, the tool generates a *.tar.gz* compressed file so it can be easily sent by mail or any other means.

Currently, this script collects the following information:

- **System information**: Installation path, kernel version, distribution version, PCI deices, kernel modules installed, mounted filesystems, system topology, CPU information, memory information, running processes and kernel log.
- **Crash logs**: Crash logs stored in */var/crash*. Not very useful if the system does not have *kdump* enabled[7].
- **HPCAP information**: Configuration, driver version, driver modinfo, installation status and monitor logs.
- **HPCAP interfaces data**: ifconfig and ethtool output, ethtool statistics, listener status and character devices.
- **Capture information**: Size and location of all capture files (the search path is the filesystem root, excluding system folders such as */etc*, */home*, */usr*, ...).

3.7.6 Outputting debug information to the kernel log

There are several macros that facilitate the output of debug information. These are defined in the file *hpcap_debug.h*. All print the corresponding driver prefix (defined in the `PFX` macro, containing *ixgbe* or *ixgbevf* depending on the driver) to allow their identification in the kernel log.

- `printdbg` Enabled only in *debug* configurations. Prints the information along with the tag "dbg" and the currently executing function.
- `adapter_dbg` Enabled only in *debug* configurations. Prints the same output that `printdbg`, but includes the current adapter name. This information is retrieved from the *adapter* variable.
- `bufp_dbg` Enabled only in *debug* configurations. Prints the same output that `printdbg`, but includes the current buffer (adapter and queue). This information is retrieved from the *bufp* variable.
- `BPRINTK` Basic kernel printing with the corresponding prefix and current function.
- `HPRINTK` Kernel printing with the corresponding prefix, current function and current buffer (adapter and queue). This information is retrieved from the *bufp* variable.
- `DPRINTK` Kernel printing with the corresponding prefix and current function, including adapter name. It also prints to the network interface log.

The question of which function to use is easy. If you want to print a debug statement use the `dbg` functions, preferably `adapter_dbg` if the *adapter* variable is accessible. If the statement should always be printed in the kernel log, even in production releases, use the `XPRINTK` variant that outputs all the information you have in the current scope. That is, `DPRINTK` if the *adapter* variable is in the scope, `HPRINTK` if we have *bufp*, or else use `BPRINTK`.

The *dbg* variants (`printdbg, adapter_dbg, bufp_dbg`) all receive a first argument that describes the type of debug statement. This should be one of the `DBG_XXX` macro that allows control of which kind of statements get printed out.

3.8 Bugs, system errors and known issues

This section shows the bugs that have been found during the development and use of the driver, hoping that it gives the reader some ideas when the driver and/or the kernel have turned crazy and decided not to work.

3.8.1 Driver/system corruption after system crash

Status:	fixed (*commit: 06b9ec8: insmod: Blacklist hpcap and ixgbe to avoid loading them at . . .*)
Affected versions:	4.2.0, 4.2.1

[7] See this for RedHat based distros.

During the testing of HPCAP 4.2.1, the system crashed because of unknown issues while HPCAP was receiving traffic. The driver was installed to the system. After a reboot, something was corrupted: both *ixgbe* and *hpcap* were loaded at boot (even though none of them was configured to do so) and could not be removed. Trying to install the module again resulted in either errors or *modprobe/insmod* hanging.

It seems that, after a crash, the Linux kernel tries to reload the modules in some weird fashion, reloading both *ixgbe* and *hpcap* and causing some weird conflicts, including errors in the Intel Direct Cache Access subsystem (in some instances, the driver crashed on install with the stack trace pointing to `dca_register_notify`).

The fix was to blacklist both drivers in the *modprobe* configuration that was already being generated on install. When the error happened, the system could be restored running `depmod -a && update-initramfs -u` with *root* permissions, or reinstalling the kernel.

3.8.2 Segmentation fault on client applications

Status:	fixed (*commit: 9f21f1f: Fix concurrency issue when adding listeners*)
Affected versions:	4.1.0 and previous

Sometimes, when multiple clients open the same HPCAP handle at the same time (for example, *monitor_flujos* opens two handles on the same file), a race condition may occur and the listener identifications will not be registered correctly. This causes messages in the kernel log warning about some "listener not found" and possibly a segmentation fault when issuing `ioctl` calls. The solution was to properly protect critical sections of the code that can be read and written concurrently.

3.8.3 modprobe/insmod: Cannot allocate memory

Status:	not fixed
Affected versions:	All

There are several possible causes. If the error is raised by the HPCAP driver, you should see a line in the kernel log explaining what happened and how to correct the error[8].

However, if the kernel log doesn't show any message from *hpcap*, it may be an issue with available memory in the kernel. The Linux kernel has a certain memory space assigned to load modules[9], usually about 1.5GB. It's more than enough for regular uses of modules, but not in the case of HPCAP. HPCAP allocates a static buffer of 1GB that will be shared between the different queues. This buffer is placed on that module mapping space[10] so, if several modules have already reserved more than 500 MB, the kernel may refuse to load our module.

The workaround for this is to reduce the buffer size (modify the macro `HPCAP_BUF_SIZE` in the *include/hpcap.h* file) and, if necessary, to use hugepages (see section 3.4) to get buffers of the necessary size. I haven't found any way to see the memory usage of each driver nor how to see how much of that module mapping space is already allocated.

3.8.4 Interfaces renamed automatically by udev

Status:	fixed (*commit: r715: insmod: workaround for udev renaming rules, r807*)
Affected versions:	4.2.2 and previous

In some systems, udev has some "smart" renaming rules that theoretically gives a predictable naming to the network interfaces. That interferes heavily with HPCAP. The workaround consists in a small check in the `interface_up_hpcap` bash function, that will check in the kernel log for *udev* renamings and will undo them.

An extra problem in some CentOS installations was that *udev* did not even print in the kernel log these renames. This required an extra fix (revision 807 in *mellanox* branch, merged into trunk in rev. 838), that consisted of a naming check (the actual function is called `hpcap_check_naming`) that is called before launching the poll threads. This functions compares the current name of the interface to the expected one (*hpcapX*, where *X* is the adapter index): if they are different, it prints a message to the kernel log that will be picked up by the installation script so it can rename the interfaces.

3.8.5 Corrupted capture files

Status:	fixed (*commit: r842, r869*)
Affected versions:	4.2.3 and previous

[8] If the message appears but does not tell how to fix the error or does not explain what's happening, tell the maintainer to change that error message.

[9] See the kernel documentation about the matter.

[10] It may be possible that I'm wrong here about this.

A bug was discovered where the captures were being corrupted if another program had been listening previously. For example, if a first instance of *hpcapdd* was launched and then closed, the resulting capture files were all correct. However, if a second instance was launched without reinstalling the drivers, all capture files would be corrupted.

The actual issue had to to with how the clients (listeners) acquire their reading offsets. A first listener would start reading from offset 0, transferring blocks of a fixed size. When the client closed, HPCAP saved the last reading offset, which would not necessarily point to the beginning of a frame.

Thus, when another new client tried to read from the HPCAP buffer, its reading offset was the last reading offset of the other client. This new client would begin saving the capture at the middle of a frame, so applications such as *raw2pcap* or *detectpro*, expecting to read first a correct packet header, would crash and/or output completely wrong results.

A first solution was to assign as read offset the last writing offset of the HPCAP producer. New frames are written starting from that offset, so new listeners would start reading frames with the correct headers. However, this caused another extra problem: misaligned access to the buffer that could hurt performance. A fix for this issue is simply resetting the read/write offsets when there are no listeners: if there are no client applications, HPCAP will forget the last read/write offsets and will start writing frames from offset 0, thus avoiding misaligned access for new clients.

A secondary bug appeared later, introduced by the first fix (r842): in some cases the global write offset would be advanced before the read offset of a new listener was configured. In these cases, the read offset would again be greater than 0 and would cause errors when writing the captures. This bug was fized in r869 by setting the read/write offsets in the intialization of the listener.

3.8.6 Connection failure / segfault on xgb interfaces

Status:	fixed (*commit: r826: ixgbe: Fix bug with ixgbe interfaces where next_to_use wasn't ...*)
Affected versions:	4.2.3. Possibly previous versions, not checked

Interfaces configured in the *xgb* mode were failing and/or crashing the system completely when they received data.

The cause was that the `rx_ring->next_to_use` pointer was not updated correctly. It should be updated in `ixgbe_release_rx_desc` to have the same value as the ring's tail pointer of the NIC, but the corresponding line was missing. This caused incoherencies between the ring status in software and in hardware, which in turn caused missing frames (thus the failing connections) and even crashes when the software tried to access to unallocated descriptors.

3.8.7 Connection reset after TX hang

Status:	fixed (*commit: r875, r876: ixgbe: Disable transmission and forced resets on TX hangs*)
Affected versions:	4.2.3 and previous

If an application in the system tried to send frames through interfaces configured in HPCAP mode, it could trigger a reset in the interface that would lead to corrupted captures.

The bug was that the *ixgbe* driver would store the packets to send in a queue which was not flushed. Then, a watchdog (either in the driver or in the kernel) would notice[11] and reset the adapter. This would in turn reset the poll thread and corrupt the capture.

The solution was to discard frames to transmit in HPCAP adapters, and to disable the code that resets the adapter when using HPCAP work modes.

3.8.8 Loss of all frames

Status: fixed (*commit: r963: hpcap: Fix bug where the driver was losing all frames*)

A misterious bug where at random times caused the driver to lose all frames. The cause was the padding check. When receiving a packet, if the space left in the capture file was exactly 2 RAW headers plus the incoming packet size, no padding would be written. The space left for the next frame was exactly the size of the RAW header, so the padlen, calculated as the remaining space in file *minus the size of a RAW header* would be zero and no padding would be inserted. The file size would be also incremented, surpassing the value of `HPCAP_FILESIZE` and either corrupting the memory space when writing with erroneous lengths or avoiding the reception of any packets as the driver would not have space to insert the padding with that size.

[11] The exact message was *initiating reset due to lost link with pending Tx work*.

3.8.9 IOCK and MNG_VETO bit enabled - capture thread stops

Status: not fixed
Affected versions: pre-5.0.0, at least

In a certain installation, the driver would stop capturing after the system received a non-maskable interrupt about an IOCK error. There is not much documentation about this error, but it seems to be related with the hardware.

The kernel log also showed a message from the base *ixgbe* driver about a `MNG_VETO` bit enabled. Reading the NIC datasheet [2], it seems to be a bit set up by the hardware when it is in low-power state that forbids link changes, so downed links would not be restored.

As of 27/4/2016, this bug seems to be caused by the hardware.

3.8.10 Concurrent listeners leads to losses and/or data corruption

Status: fixed (*commit: r959, r955*)
Affected versions: 4.2.3 and previous

Several related errors happened when there was more than one client listening for data in a HPCAP interface. For example, if a listener connected first but did not read anything, a second listener would receive an advanced read offset (the last write from the polling thread). So, even when the two listeners could start reading data from the buffer at the same time they would see different data. This was fixed in revision 959.

Another problem happened when two listeners connected to the driver and then the first one disconnected. The next listener to connect would receive handle ID 2, which was already being used. Data loss and/or corruption of the structures would then occur. This was fixed in revision 955.

3.8.11 Copies from/to user space can fail

Status: fixed (*commit: r956: hpcap: Fix copies from/to the user memory*)
Affected versions: 4.2.3 and previous

In the `ioctl` calls, the copies to and from user space memory were not being done with `copy_to/from_user` and, in strange instances where that memory was invalid, that could cause a segfault within the driver. With the fix, no invalid acceses are done and the correct error message is returned to the caller application.

3.8.12 Corrupted capture files after restarting a client

Status: fixed (*commit: r893: hpcap: Reset written buffer size when there're no listeners*)
Affected versions: 4.2.3 and previous

Related to bug 3.8.5, when restarting a listener, the captures would be subtly corrupted as the driver would not place the padding at the end of the file but in the middle instead. This happened because the filesize counter was not being reset if there were no listeners.

3.8.13 Problems when using buffers greater than 2GB

Status: fixed (*commit: r990: hpcap: Fix several problems with buffers greater than 2G*)
Affected versions: 4.2.X

When using hugepage-backed buffers with a size greater than 2GB, several variables would overflow and cause segfaults and memory errors.

There is a secondary problem: when allocating buffers of size 4GB, the function `vm_map_ram` used in the hugepage code will segfault due to a bug in the Linux kernel code. See the linux-mm mailing list for more information. The patch was included in kernel version 4.7.

3.8.14 Frames with VLAN tags disappeared

Status: fixed (*commit: r1025: ixgbe: Fix VLAN stripping*)
Affected versions: pre-5.0.0

During the update of the ixgbe base driver from 3.7.17 to 4.1.2, the VLAN stripping features were not correctly deactivated. This caused the card to strip the VLAN tags in hardware. The solution was to disable all hardware features in the correct location in the code.

3.8.15 RX Errors with no cause

Status: not fixed
Affected versions: All

In some environments, the *ethtool* counters for `rx_errors` increment, but without CRC or length errors. These may be caused by corrupted non-Ethernet packets, such as malformed Cisco Spanning Tree Protocol (STP) frames. The solution was to make the NIC stop counting these frames as errors, disabling a certain register. This required to add the line

```
hlreg0 &= ~IXGBE_HLREG0_RXLNGTHERREN
```

Chapter 4

NIC Interface and communication

Refer to the Intel controller datasheet [2] for more detailed information. Some relevant information is copied and explained here.

4.1 Statistics retrieval

The NIC fills several registers with different statistics regarding the adapter operation, such as missed frames or CRC errors. The NIC has 128 receive queues, but the amount of available registers does not allow to allocate one register for each queue and possible statistic. Thus, only a limited number of registers is allocated, and each register will store information for several queues. Section 8.2.3.23.71 on page 688 on the datasheet explains the storage configuration. It uses the `TQSM[n]` register, for `n` = 0-31. Each `TQSM[n]` register holds the mapping configuration for the four queues $4n$ to $4n + 3$.

The `TQSM[n]` register should be used to retrieve the wanted statistic from the correct register. For example, the `RXMPC[n]` register (section 8.2.3.23.4, page 674 on the datasheet) shows the missed packet count for `n=0-7`. The queues will store on one of those 8 registers the missed packet count depending on the mapping configuration.

Chapter 5

User-space applications

5.1 Samples

5.1.1 huge_map

The *huge_map* application is a small utility that allows the creation of hugepages-backed buffers in the driver. Its usage is the following:

```
huge_map adapter queue action buffer-size [hugetlb-path]
```

`action` is either `map` or `unmap`, depending on whether the desired action is memory allocation or freeing. `buffer-size` is the buffer size (can be human-readable file sizes, such as *1GB* or *3000MB*. Finally, `hugetlb-path` is the path to the *hugetlbfs* filesytem (see section 3.4.2). By default, */mnt/hugetlb* will be used if this option is not provided.

5.1.2 checkraw

To allow fast checking of the integrity of generated RAW files, the *checkraw* program is provided. It can receive one or more RAW files or directories where RAW files are stored, and will check all of the files found.

Optionally, the program can receive a minimum timestamp with *-t timestamp*. If it encounters files with an older timestamp, it will ignore them.

checkraw will output warnings and/or errors found in the RAW captures. If errors are found, they are printed together with the human-readable timestamp, to allow for easy identification.

5.2 Libraries

5.2.1 libhpcap

This library provides raw, direct access to the driver functions. The corresponding header file is *hpcap.h*. The methods are documented using doxygen, so you can find the working details in the man pages or the PDF/HTML manuals (see section 1.5).

5.2.2 libmgmon

This is a wrapper that offers a simple interface to the HPCAP drivers. It offers three loop functions, that will read the data from the given adapter either receiving frames, flows or mrtgs passed through a callback function. See the *libmgmon.h* header file for the documentation.

5.3 Scripts

In the *scripts* folder you will find several scripts created to facilitate the use of the HPCAP driver. Some of them are meant to be installed and used from your path (see section 2.3). The most relevant scripts are documented in the following sections.

Additionally, the *hpcap-lib.bash* file holds several auxiliar functions used by other scripts.

5.3.1 Monitoring scripts

To manage and control the interface monitors, fourh scripts have been developed: *hpcap-monitor, stop-hpcap-monitors* and *launch-hpcap-monitors*, and *hpcap-status*. The first and most important is the monitor itself, *hpcap-monitor*. Its first required argument is the interface to monitor. The second argument is optional: if present and equals to *vf*, the monitor will take into account that the interface has virtual functions.

The monitor script reads the *params.cfg* file from the corresponding location depending on whether the script is placed in the source folder (thus reading the configuration from the source folder too) or installed in the */usr/bin* system folder, thus reading the configuration from the */etc/hpcap/params.cfg* location.

If the script is running from the source folder, the log destination folder is the *data* folder. If installed, the destination will be the one set in the configuration (*monitor_basedir* parameter), which by default points to */var/log/hpcap*.

The script analyzes the output from the `ethtool -S iface` command, logging the number of bytes and frames received, the lost frames and estimates the bytes lost based on those numbers.

To manage the monitors automatically, the scripts *launch-hpcap-monitors* and *stop-hpcap-monitors* have been created. It's specially useful to use these two scripts to launch the monitors. The launcher reads correctly the parameters from the configuration file, setting the monitors in the correct core and putting them in the background to avoid them from stopping when the current console session is closed. It also saves the output from the scripts and their PID in files in the log directory. The PID will be later used in the stop script to stop the correct monitors. You can also use it to stop monitors manually executing `kill -TERM $(cat logdir/iface-monitor.pid)`.

Finally, *hpcap-status* is a simple script that continously shows the driver information: memory status, installation state, link states and interface traffic rates, including losses. It feeds on information from the monitors mentioned previously, and it can be useful to have a "status screen" when testing the driver.

5.3.2 Automated testing, benchmarking and configuration

To avoid inconveniences and ease development, some scripts have been developed to aid testing and configuration. The first one is *gen-hpcap-config*, which generates a *params.cfg* automatically, using common defaults and taking into account the NUMA architecture of the driver.

Automated tests An automated tester script, *hpcap-test*, has also been developed. It tests the correction of configuration files; correct compilation of the driver, samples and libraries; driver correct installation and configuration; and, optionally, it can perform automated traffic tests. Currently, the only supported system for traffic generation is MoonGen with Lua scripts. However, adding other generators such as DPDK should not be difficult.

All the parameters of the automated tester are managed in the *hpcap-test.cfg* file. The file *hpcap-test.cfg.sample* is provided as a sample, that should be modified to fit each scenario.

See section 3.7.3 for details on these scripts.

Automated benchmarking The script *hpcap-benchmark* servers as a standard tool for benchmarking the results of HPCAP. This script installs the driver and then sends traffic through the generator defined in *hpcap-test.cfg*, with different rates and frame sizes as specified in the configuration. It outputs the results in the screen (capture ratio, lost frames and generation traffic rate), and also saves them in a *.dat* file (the specific name is printed by the script at the end of the test). If gnuplot is present in the system, a graphic with the results is also generated (again, the specific filename is printed at the end of the test).

5.3.3 Modprobe helper scripts

When using *modprobe* to load the drivers in the system, there are several tasks that have to be completed apart from loading the driver, such as creating the node files and network interfaces. The scripts *hpcap-modprobe* and *hpcap-modprobe-remove* are meant to do these tasks. These scripts should not be called directly by the user.

Currently, the install script just loads the driver passed as argument (*hpcap* or *hpcapvf*) with the corresponding parameters read from the */etc/hpcap/params.cfg* file, and then set each interface up. This procedure (function `interface_up_hpcap` in *hpcap-lib.bash*) creates the queue nodes in the */dev/* filesystem, map the corresponding hugepages if required (that is, if the interface is in mode 3), sets the IRQ affinity, brings the interface up and negotiates the Ethernet connection.

The remove script unloads the driver and frees the resources used, deleting the existing hugepage buffers and deleting the nodes in the */dev/* filesystem.

5.3.4 Build system scripts

To improve the clarity in the *Makefile*, some functions have been placed in separated scripts.

mkmakefile is a script that automatically generates a Makefile for each driver configuration. The Makefile generated will be used by the kernel build system to compile the driver in the given configuration. See section 2.2 to see how this is accomplished.

generate_modprobe_conf just generates a simple configuration file for *modprobe*. For each driver, generates two lines *install* and *remove* that will make modprobe to call the helper scripts described in section 5.3.3. This script also generates *blacklist* lines to indicate the system that the *ixgbe* and *hpcap* drivers should not be loaded at boot automatically, thus avoiding conflicts that could result in nasty bugs such as the one in section 3.8.1.

The Makefile will generate this configuration file when installing the driver to the system (see section 2.3).

Index

Bibliography

[1] V. Moreno. Development and evaluation of a low-cost scalable architecture for network traffic capture and storage for 10Gbps networks. Master's thesis, Escuela Politecnica Superior UAM, 2012.

[2] Intel. Intel 82599 10 GbE Controller Datasheet. *October*, 2010.