Universidad Autónoma de Madrid
Escuela Politécnica Superior

Work to obtain master degree in
Computer and Telecommunication
Engineering
by Universidad Autónoma de Madrid

Master thesis advisor:
Dr. Francisco Javier Gómez Arribas

Development and evaluation of a low-cost scalable architecture for network traffic capture and storage for 10Gbps networks

Víctor Moreno Martínez

This work was presented on 2012
Tribunal:

Dr. Francisco Javier Gómez Arribas
Dr. Iván González Martínez
Dr. Jorge E. López de Vergara

© 2012 by UNIVERSIDAD AUTÓNOMA DE MADRID
Francisco Tomás y Valiente, n^o 1
Madrid, 28049
Spain

Víctor Moreno Martínez
Development and evaluation of a low-cost scalable architecture for network traffic capture and storage for 10Gbps networks

Víctor Moreno Martínez
Escuela Politécnica Superior. High Performance Computing and Networking Group

IMPRESO EN ESPAÑA – PRINTED IN SPAIN

Final Master's Thesis evaluators:

Dr. Francisco Javier Gómez Arribas
(Chairman)

Dr. Iván González Martínez

Dr. Jorge E. López de Vergara

Reviewer:

Dr. Francisco Javier Gómez Arribas

Table of Contents

Acknowlegments

To my family:

To my mother Teresa for fighting to be here today and, with my father Fernando and my brother Fernando, for supporting me all those years and being always there. A special acknowledge to my grandparents Teresa, Víctor, María and José who are no longer here. I also want to thank all my uncles, aunts and cousins. I wouldn't be the one I am without any of them.

To professor Francisco J. Gómez Arribas:

for guiding and enlightening my work all those years. I really appreciate his knowledge, patience, time and valuable comments. I have learned a lot working with him all those years.

To the people of the C113 lab:

Javier Ramos, Pedro Santiago, Jaime Garnica, José Luis García, Felipe Mata, David Muelas, David Madrigal, Marco Forconesi, Pedro Gómez, Álvaro García, Miguel Cubillo, for their help and for making the C113 lab a nice place to work at. Thanks as well to the former members of the lab: Diego Sánchez, Jaime Fullaondo, Víctor López, Rubén Nieto, ...

To the High Performance Computing and Networking Group from UAM:

for giving me the chance to work with them four years from now. Thanks to all of the senior researchers of the group for their teachings and valuable comments along those years: Javier Aracil, Francisco Gómez, Iván González, Sergio López, Gustavo Sutter, Jorge E. López and Luis de Pedro.

To my friends:

for helping me laugh and clearing my mind.

To the Spanish Ministry of Education:

as this work has been partially financed by the FPU Scholarship from this organism.

To Eloy Anguiano Rey:

for this document's LaTeX format.

Abstract

The last years have witnessed an undoubtedly explosion of the Internet users' demands for bandwidth. To manage such new demand and, especially, provide the adequate quality of service, ISPs have understood the importance of accurately monitoring their traffic, investing a great deal of effort in terms of funds and time. Novel packet I/O engines allow capturing traffic at multi-10Gbps using only-software and commodity hardware systems. This is achieved thanks to the application of techniques such as batch processing.

Nevertheless, such novel I/O engines focus on packet processing throughput while shifting to the background the ability of recording the incoming packets in non-volatile storage systems. In this work the storage capabilities of novel I/O engines will be evaluated, and a scalable solution to this problem will be developed.

Moreover, the use of batch processing involves degradation in the timestamp accuracy, which may be relevant for monitoring purposes. Two different approaches are proposed in this work to mitigate such effect: a simple algorithm to distribute inter-batch time among the packets composing a batch, and a driver modification to poll NIC buffers avoiding batch processing. Experimental results, using both synthetic and real traffic, show that our proposals allow capturing accurately timestamped traffic for monitoring purposes at multi-10Gbps rates.

1 INTRODUCTION

The Internet users' demands for bandwidth are drastically increased every year. To manage such growing demands, ISPs require new tools and systems allowing to accurately monitor their traffic in order to provide the adequate quality of service. Few years ago, traffic monitoring at rates ranging from 100 Mbps to 1 Gbps was considered a challenge, whereas contemporary commercial routers feature 10 Gbps interfaces, reaching aggregated rates as high as 100 Tbps [1]. As a consequence, specialized hardware-based solutions such as NetFPGA or Endace DAG cards, as well as other ad-hoc solutions, have been used to such a challenging task.

Alternatively, in recent years, the research community has started to explore the use of commodity hardware together with only-software solutions as a more flexible and economical choice. This interest has been strengthened by multiple examples of real-world successful implementations of high performance capturing systems over commodity hardware [2–4]. Such approaches have shown that the keys to achieve high performance are the efficient memory management and low-level hardware interaction. However, modern operating systems are not designed with this in mind but optimized for general purpose tasks such as Web browsing or hosting. Studies about Linux network stack performance, as [5], have shown that the major flaws of standard network stack consists in:

- per-packet resource (de)allocation and
- multiple data copies across the network stack.

At the same time, modern NICs implement novel hardware architectures such as RSS (Receive Side Scaling). Such architectures provides a mechanism for dispatching incoming packets to different receive queues which allows the parallelization of the capture process. In this light, novel capture engines take advantage of the parallelism capacities of modern NICs and have been designed to overcome the above mentioned deficiencies. That is, these capture engines have tuned the standard network stack and drivers to implement three improvements:

- per-packet memory pre-allocation,
- packet-level batch processing and
- zero-copy accesses between kernel and user space.

The use of novel I/O packet engines focuses on obtaining high throughput in terms of packet processing. Nevertheless, many kinds of monitoring applications, e.g. forensic analysis or traffic classification, need access to a traffic trace history. Thus, recording the incoming traffic into non-volatile storage systems becomes a capital importance task for such systems. Works such as [6] have paid attention to the packet capturing and storing problem using commodity hardware together with standard software. Such solutions have only succeeded in capturing packet rates below 1 Mpps (Millions of packets per second).

While the improvements introduced by novel packet I/O (Input/Output) engines boost up the packet capture performance, surprisingly, packet timestamp capabilities have been shifted to the background, despite their importance in monitoring tasks. Typically, passive network monitoring requires not only capturing packets but also labelling them with their arrival timestamps. Moreover, the use of packet batches as a mechanism to capture traffic causes the addition of a source of inaccuracy in the process of packet timestamping.

1.0.1 Goals of this work

This work studies the two major issues of both traditional and novel I/O packet engines just mentioned.

With regard to the packet capture and storage problem, this work focuses on:

- studying how can the use of novel I/O engines can outperform the packet capture throughput of traditional approaches,
- proposing a new capture approach to achieve a scalable capture rates to work at 10Gbps (Gigabit per second) rate, and
- assessing our approach's capabilities for non-volatile packet storage.

Regarding the packet timestamp issue, this work does:

- study the timestamping accuracy of batch-oriented packet capture engines, and
- propose several techniques to overcome the novel I/O packet engines accuracy problem.

1.0.2 Document structure

This document is structured as follows: chapter 2 contains an overview of the state-of-the-art regarding how do several packet capture engines work, focusing on both their strengths and weaknesses. The development of our proposed packet capture engine is exposed in chapter 3.

In chapter 4 we focus on the analysis of the packet capture and storage problem, while chapter 5 focuses on the timestamp accuracy problem.

Finally, conclusions and future work are included in chapter 6.

2 State of the art

2.1 The Intel 82599 NIC

The Intel 82599 [7] is a 10 Gbps capable NIC. Novel I/O packet engines [2–4] make use of such general-purpose NIC. This section describes how this network card works in conjunction with. Most of the concepts that will be exposed along this sections are valid for other NICs.

2.1.1 Descriptor rings and memory transfers

The NIC has a set of ”rings” used to store both incoming and transferred packets, consisting on a circular buffer of data structures called descriptors. Each descriptor contains information regarding the state of the packets and a pointer to a memory area. Note that, as shown in Fig. 2.1, consecutive descriptors can have pointers to non-consecutive memory

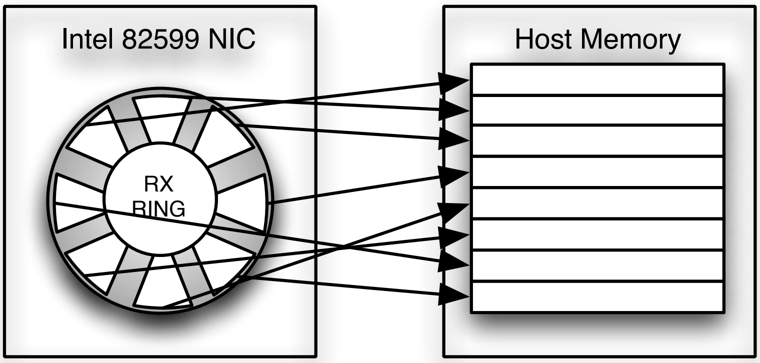


Figure 2.1: Intel$^{®}$ 82599 descriptor ring

When a new packet arrives from the network to the NIC, the hardware will probe the corresponding RX-ring for free descriptors. If there is any descriptor available, the incoming packet will be copied from the NIC to the host memory via a DMA transfer and then the NIC will update the flags of the descriptor involved.

Reversely, when the host is to send a packet through the network it will probe the TX-ring for available descriptors. If there is any descriptor available, the outgoing packet will be copied from the network stack to the host memory area pointed by the TX descriptor and then the corresponding flags will be updated by the host process. Once those flags have been updated, the packet will be transferred to the NIC via a DMA transfer and then sent through the network.

2.1.2 Multi-queue capabilities

In order to achieve high bit rates throughput, advanced NICs allow information exchange between host and network using multiple queues. Each RX/TX instantiates an isolated descriptor ring. This way, a MP (Multiprocessor) system could exploit parallelism by mapping each queue to a different processing core, thus improving the overall network throughput.

With the aim of maximizing parallelism the total amount of traffic has to be distributed amongst the different queues. Techniques such as RSS [8] allow distributing the traffic based on a hash value calculated from the packets five-tuple[1] as shown in Fig. 2.2.

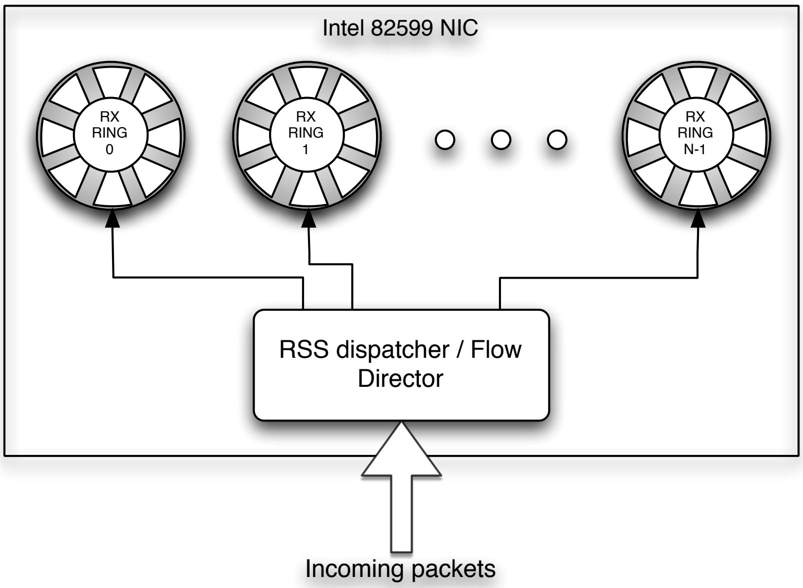


Figure 2.2: Incoming packet balancing at the Intel 82599

The Intel 82599 NIC [7] RSS policy guarantees that packets belonging to the same flow will be dispatched to the same queue. However, this may not be suitable for all kind of upper level applications, specifically, applications needing to keep track and correlate traffic related to a bi-directional connection. In such case, the uplink and downlink flow have a permuted five-tuple and thus could be delivered to different queues. If this happens, packet reordering could appear when collecting the traffic from different queues [9] and unfriendly situations could appear: the `ACK` message of a TCP connection establishment could be processed before the corresponding `SYN` message, $\cdots$

Nevertheless, Intel 82599 NIC provides and advanced mechanism called Flow Director that allows tying flow five-tuples to queues. Under some throughput constraints, this technique allows that applications working with bi-directional traffic to exploit multi-queue capabilities. Studies such as [10] propose an architecture in which a software hash function is applied to the incoming packets guaranteeing that both uplink and downlink flows are delivered to the same software queue.

[1] IP source and destination addresses, IP protocol field and TCP/UDP source and destination ports

2.1.3 Traditional packet receiving scheme

Fig. 2.3 shows how does the traditional packet receiving scheme work in a Linux system, which is true not only for the Intel 82599 but for most NICs. When a packet DMA transfer has finished, the host system will be notified via an interrupt so the driver in charge can carry out the corresponding action. At this point, the network driver fulfils a *sk_buff* (standing for socket buffer) data structure with the information of the incoming packet. As soon as the *sk_buff* structure is filled, the data structure is pushed into the network stack by means of the *netif_rx* function. This means that an interrupt needs to be risen for processing each of the incoming packets, whose processing overhead turns into the system bottleneck.

In order to reduce the amount of required interrupts, modern NICs follow the NAPI (New API) [11, 12] model. NAPI reduces the amount of interrupts needed to process all incoming packets at high rates by changing the driver I/O mechanism from interrupt-driven to polling. When the first packet arrives, the NIC (Network Interface Card) raises an interrupt in which such packet is processed and then polls the NIC rings to check whether there are more packets to be processed. If a poll check fails, the polling process stops and the next incoming packet will launch a new interrupt.

Regardless the use of NAPI, when the incoming packet is introduced into the network stack, its content will be copied every time it traverses a network layer. Finally, packet is kept in a queue until reclaimed by an application via a socket *recv()* call, or until the packet is deleted due to inactivity.

2.1.4 Linux network stack's performance flaws

Packet capture at multi-Gb/s rates making use of commodity hardware and only-software solutions is a challenging task. Standard drivers and operating system network stacks are unable to make the most of such capacity because they are not designed for this purpose but optimized for common desktop or server tasks (e.g., Web browsing or hosting). Studies about Linux network stack performance, as [2, 5, 13], have shown that the major flaws of standard network stack are: Mainly, the limitations of the network stack and default drivers are:

- **per-packet memory operations:** once a packet DMA transfer from/to the NIC finishes, the driver will free the memory area associated to the corresponding descriptor, just to allocate and map a new piece of memory. This means that the driver will be constantly allocating new DMA-capable memory regions, which are costly operations.
- **multiple copies through the stack:** as stated before, when a packet is pushed into the network stack, a new copy of that packet will be made every time it traverses a new network layer.

Those features give the network stack a high degree of isolation and robustness, but turn into a bottleneck when working at gigabit rates.

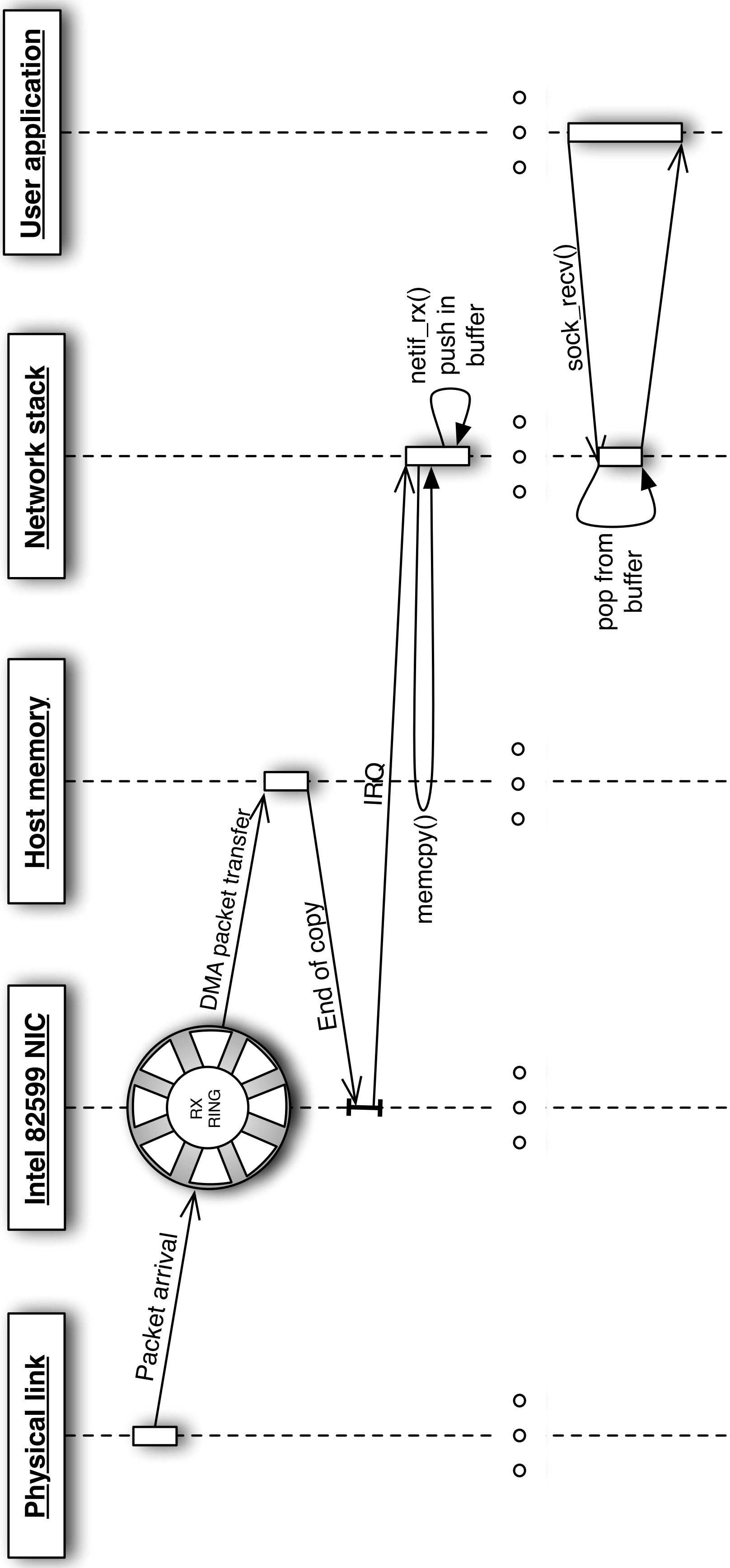


Figure 2.3: Traditional packet receiving scheme

2.2 Novel I/O engines

To overcome the Linux network stack deficiencies, novel packet I/O engines have been proposed [2, 14, 15] in the recent years. Such capture engines have modified the standard network stack and drivers to implement three enhancements, namely:

- memory pre-allocation: a pool of memory buffers is allocated when the driver is woken up, instead of freeing such buffers and allocating new ones every time a packet is received, the driver will reuse the previously existing buffers.
- packet-level batch processing: differently from traditional network stacks, incoming packets will be fed to upper level application in batches. This way, the overhead due to the required call chain is minimized.
- zero-copy: direct access between the application and the driver

Thanks to these modifications along with a careful tuning of the software design and critical hardware parameters, existing overheads in standard operating system network stack are removed, allowing packet sniffing up to 40 times faster [15], achieving wire-speed capturing, i.e., up to 10 Gb/s and more than 14 Mpps per interface.

Fig. 2.4 shows how does PacketShader [2] work in contrast to the behaviour of traditional network stacks shown in Fig. 2.3. Firstly, packets are copied into host memory via DMA transfers the same way as in a traditional system. The big difference is that no interrupt process will process the incoming packets, it will be the user level application the one that will ask for packets by calling the *ps_recv()* function from the PacketShader API. When issuing such call, the driver will poll the RX descriptor ring looking for packets that have already been copied into host memory. If there any packets available, all of them will be copied to a memory buffer that has been mapped by the user-level application thus acceding them in a zero-copy basis.

In order to achieve peak performance, both CPU and memory affinity issues must be taken into account. The user level application must be scheduled in the same NUMA node in which the memory buffer where the incoming packets will be copied at their arrival. Otherwise, memory access latency could damage capture performance.

Note that the RX descriptors won't be reused until their corresponding packet has been copied as a consequence of a *ps_recv()* request from the upper level application. This way, the system packet capture performance depends on how fast the users' applications are able of consuming the incoming packets in order to leave space for the new ones. If needed, packet capture performance can be improved by using several RX queues: different application threads/processes would ask for packets on their corresponding RX queue. In that case, affinity issues become even more relevant: the different capture threads must be scheduled in different cores, always keeping in mind that the core each thread/process is mapped to resides in the same NUMA node as its associated RX queue.

Netmap [15] is a different packet capture engine following a similar approach to PacketShader. Both solutions follow the same principles and provide a particular API for user-level applications. They are both capable of line-rate packet capture in 10 Gb/s network using just one RX queue [2, 15].

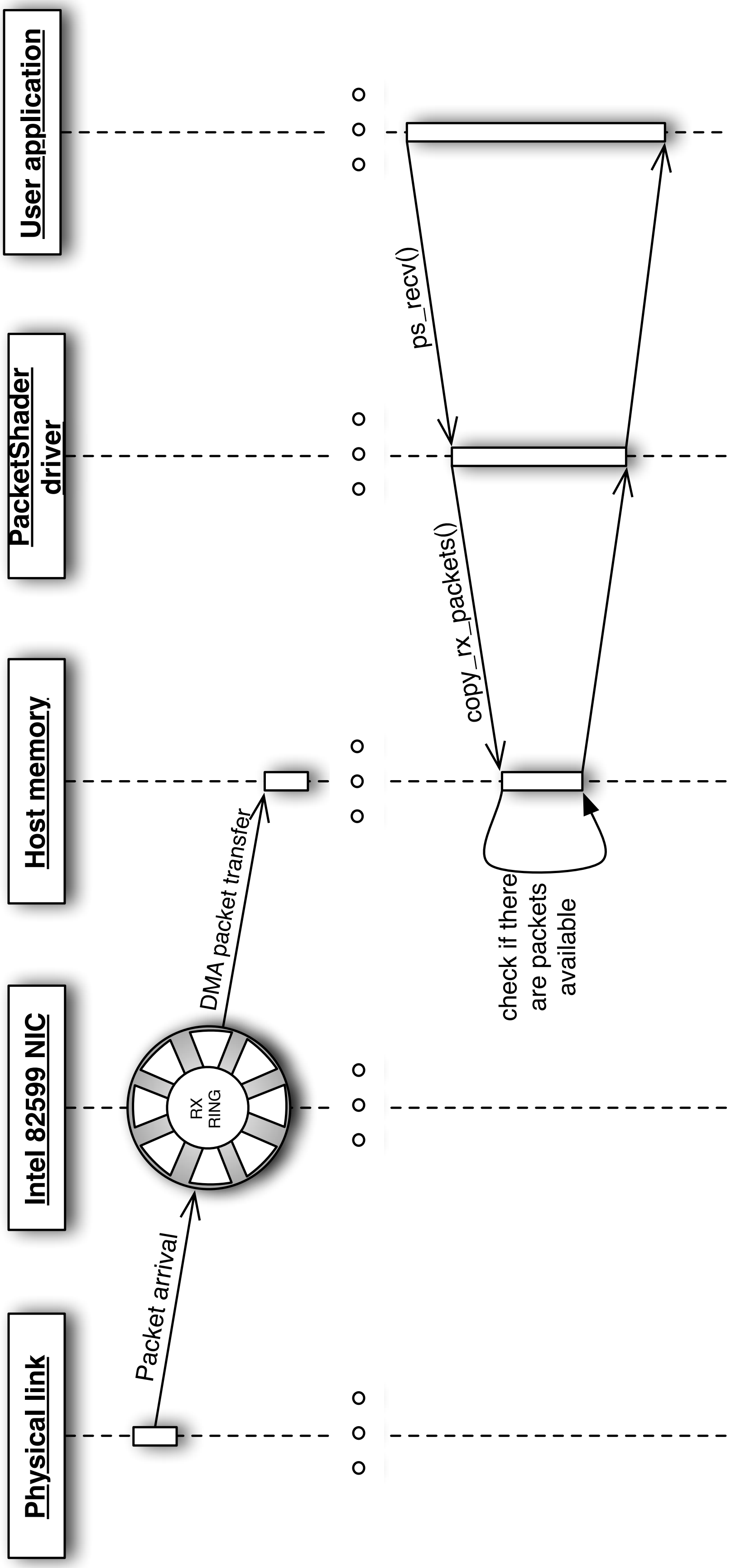


Figure 2.4: PacketShader packet reception scheme

Other existing approaches such as PFQ [4] or DashCap [10] bet for a socket-alike interface. Thus, those solutions provide a more user friendly interface then PacketShader or Netmap. Nevertheless, such approaches are not capable of achieving comparable performance than PacketShader or Netmap: Dashcap works at 1 Gb/s [10], while PFQ needs using more than 10 RX queues in order to achieve 10 Gb/s line-rate capture [4].

While novel I/O engines improvements boost up the performance of packet capture engines, surprisingly, packet timestamp capabilities have been shifted to the background, despite their importance in monitoring tasks. Typically, passive network monitoring requires not only capturing packets but also labelling them with their arrival timestamps. In fact, the packet timestamp accuracy is relevant to the majority of monitoring applications but it is essential in those services that follow a temporal pattern. As an example, in a VoIP monitoring system, signaling must be tracked prior to the voice stream.

In this light, there are two features that an ideal packet capture engine should meet, namely:

- with the aim of avoiding reordering effects [9], it should achieve line-rate packet capture using just one RX queue;
- it should emboss incoming packets with an as accurate as possible timestamp.

Up to date, none of the existing commodity hardware based capture engines do simultaneously meet both requirements. This work proposes a new packet capture mechanism that meets both two.

3 HPCAP PROBE DESCRIPTION

In this chapter we describe the implementation details of HPCAP (High-performance Packet CAPture): our packet capture proposal. HPCAP is designed with the aim of meeting the two features that an ideal packet capture engine should meet, as mentioned in section 2.2, namely:

- it should achieve line-rate packet capture using just one RX queue;
- it should associate incoming packets with an as accurate as possible timestamp.

Moreover, many kinds of monitoring applications, e.g. forensic analysis or traffic classification, need access to a traffic trace history. Thus, recording the incoming traffic into non-volatile storage systems becomes a capital importance task for such systems. In this light, there is one more feature than an ideal packet capture engine should meet:

- it should be able to store all the incoming packets in non-volatile volumes.

Up to date, none of the existing commodity hardware based capture engines do simultaneously meet the first two requirements, and none of them have paid attention to non-volatile traffic storage problems. This work proposes a new packet capture mechanism that meets all of the three previously explained features.

Although our goal is a capture solution working with just one RX queue, the system architecture has been designed so that more than one queue can be used. HPCAP architecture is depicted in Fig. 3.1. Three main blocks can be distinguished inside HPCAP:

- the Intel 82599 NIC interface: this block enables the communication between the HPCAP driver, and the NIC. It is comprised by the source C code files from the opensource ixgbe (Intel's 10 Gigabit Ethernet Linux driver).
- the HPCAP driver: this block consists on a set of C source code files that are compiled together with the ixgbe driver files to generate a new driver.
- user level applications: a user-level application using the HPCAP API, which consumes the incoming packets.

3.1 Kernel polling thread

A basic technique when trying to exploit your system's performance is the ability to overlap data copies and processing. With the aim of maximizing, we instantiate a kernel-level buffer in which incoming packets will be copied once they have been transferred to host memory via DMA (see Fig. 3.1).

Other capture engines, such as PacketShader [2] or Netmap [15], do copy incoming pack-

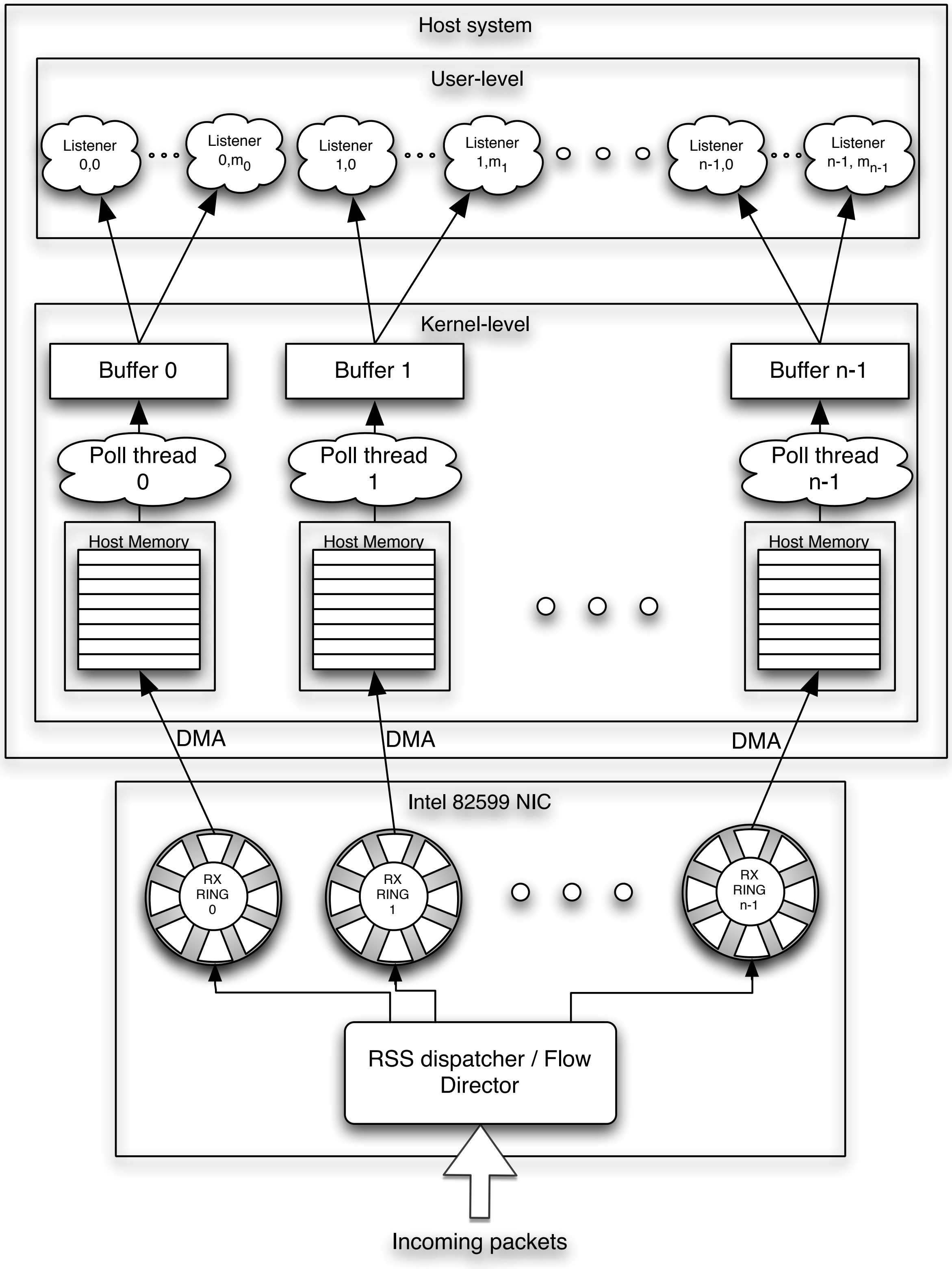


Figure 3.1: HPCAP kernel packet buffer

ets to an intermediate buffer. Nevertheless, this copy is made when the user-level application asks for new packets. This philosophy has two main drawbacks:

- as the incoming packets are copied to the intermediate buffer when the application asks for them, there is no copy and process overlap, thus limiting the capture performance,
- if the capture engine was to timestamp incoming packets, it could only be done when they are copied from RX ring memory to the intermediate buffer. As packets are not copied until the user asks for them, its timestamp accuracy is damaged. Whatsmore, packets copied due to the same user-level request would have a nearly equal timestamp value. A full discussion about that matter is included in chapter 5.

In order to overcome such problems, HPCAP creates a KPT (Kernel-level Polling Thread) per each RX queue. Those threads will be constantly polling their corresponding RX descriptor rings, reading the first descriptor in the queue's flags to check whether it has already been copied into host memory via DMA. If the poll thread detects that there are any packets available at the RX ring, they will be copied to the poll thread's attached circular buffer. Just before this packet copy is made, the poll thread will probe for the system time by means of the Linux kernel `getnstimeofday()` function, whose accuracy will be evaluated in chapter 5.

All the incoming packets are copied into the buffer in a raw data format (see Fig. 3.2): first, the packet timestamp is copied (32-bit for the seconds field and other 32-bit field for the nanoseconds), after it, the packet's length (a 16-bit bit field), and in the last place the packet data (with a variable length).

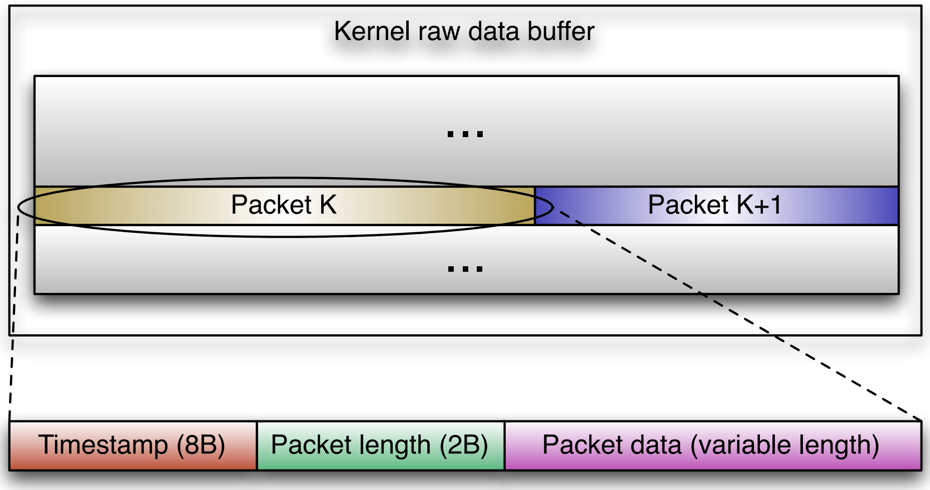


Figure 3.2: HPCAP kernel packet buffer

This raw buffer format gives a higher level of abstraction than the typical packet structure used by PCAP-lib or PacketShader, so upper level applications can efficiently access the data in both a packet oriented or a byte-stream basis. This byte-stream oriented access to the network allows using standard non-volatile storage applications like the Linux `dd` tool [16], thus benefiting incoming packet storage performance.

3.2 Multiple listeners

As shown in Fig. 3.1, HPCAP supports multiple applications, referred as *listeners*, to fetch packets from the same RX queue in a SPMC (Single Producer, Multiple Consumer) basis. This is achieved by keeping an array of structures (one for each listener thread/application plus a ”global listener”) keeping track of how much data has each listener fetched from the common buffer. In order to keep data consistency, the packet read throughput will be set by the slowest listener.

Each listener structure consists of four fields, namely:

- **a listener identifier:** a field used to identify the different active listeners for each RX queue. This field is needed to keep consistency between different packet requests coming from the same listener. The listener identifier field is filled when a new listener opens an HPCAP session, and cleared when this session is closed.
- **a read pointer:** this field is used to know the beginning of the buffer memory where the copy to user space transfer must be made when a user application issues a read request. When a new listener is registered for an HPCAP queue, this field is set to value set in the global listener field, guaranteeing that all the data residing in the buffer from this moment on will be accessible by this listener. After that, this field is only reader and updated by the listener application when it reads a new block of bytes, so no concurrency-proven mechanism must be applied to that field.
- **a write pointer:** this field tells the kernel-level poll thread, where a new incoming packet must be copied. Thus, this field is only read and updated by the kernel poll thread and again no concurrency-proven mechanism needs to be applied.

  atomic operations
- an available byte counter: this counter indicates the amount of data(in terms of bytes) currently available in each RX queue buffer. This value is increased by the kernel poll thread, and decreased by the slowest listener thread. Thus, concurrency-proven techniques must be applied to avoid inconsistency in this data field [17]. We have chosen to use the Linux atomic API.

This feature allows monitoring application to focus on packet processing, while a different application stores them into non-volatile volumes, thus overlapping data storing and processing and exploiting maximum performance.

3.3 User-level API

The "everything is a file" Linux kernel philosophy, provides a simple way to communicate user-level applications with the HPCAP driver. Following that philosophy, HPCAP instantiates a different character device [18] node in the `/dev/` filesystem for each to the different RX queues belonging to each of the different available interfaces. This way, a system holding `N` network interfaces with `M` RX queues each would see the following devices in its `/dev/` directory:

```
...
/dev/hpcap_xge0_0
/dev/hpcap_xge0_1
        ...
/dev/hpcap_xge0_<M-1>
/dev/hpcap_xge1_0
/dev/hpcap_xge1_1
        ...
/dev/hpcap_xge1_<M-1>
...
/dev/hpcap_xge<N-1>_0
/dev/hpcap_xge<N-1>_1
        ...
/dev/hpcap_xge<N-1>_<M-1>
...
```

Code 3.1: Contents of the `/dev/` directory in a system running HPCAP

A user level application that wants to capture the packets arriving to queue `X` of interface `xgeY` does only need to execute an operating system `open()` call over the character device file `/dev/hpcap_xgeY_X`. Once the application has opened such file, it can read the incoming packets by performing standard `read()` call over the previously opened file. This way, the user level application will be copied as much bytes as desired of the contents of the corresponding kernel buffer, as shown in Fig. 3.3. When the application wants to stop reading from the kernel buffer, it just has to execute a `close()` call over the character device file, and its corresponding listener will be untied from the listeners pool. Such an interface, allows the execution of programs over HPCAP that will read the incoming packets from the corresponding buffer, but it also allows using standard tools such as Linux's `dd` to massively and efficiently move the data to non-volatile storage volumes.

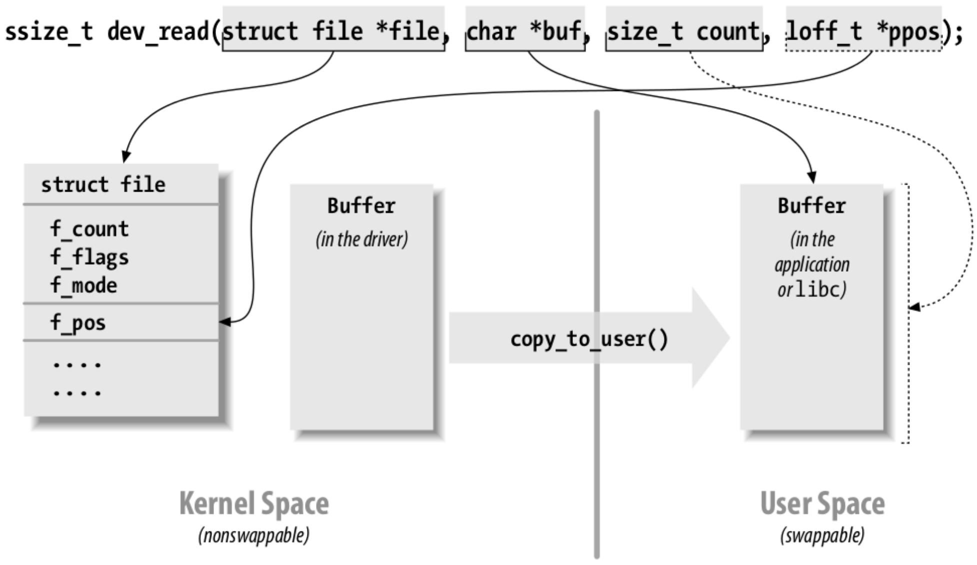


Figure 3.3: Character driver reading. Image taken from [18]

3.4 HPCAP packet reception scheme

Fig. 3.4 shows HPCAP's packet reception scheme, in contrast with the schemes regarding the packet capture engine of a traditional NIC and PacketShader, shown in Fig. 2.3 and Fig. 2.4 respectively.

As in PacketShader, the packet reception process no longer depends on the use of interrupts as the mechanism to communicate the hardware and the network driver. Instead, the already mentioned kernel poll thread will constantly copy (and timestamp) the available packets into its corresponding buffer. Note that this approach is different from PacketShader's, as the packet copies are not made when a user application asks for more packets, but always there are available packets.

Note that, just as mentioned in section 2.2, in order to achieve peak performance both the kernel poll thread and its associated listeners must be mapped to be executed in the same NUMA node. HPCAP kernel poll threads' core affinity is set via a driver load time parameter. Regarding listener applications, a single-threaded application can be easily mapped to a core by means of the Linux `taskset` tool [19]. Multithreaded application can make use of the `pthread` (POSIX Threads) library in order to map each thread to its corresponding core [20].

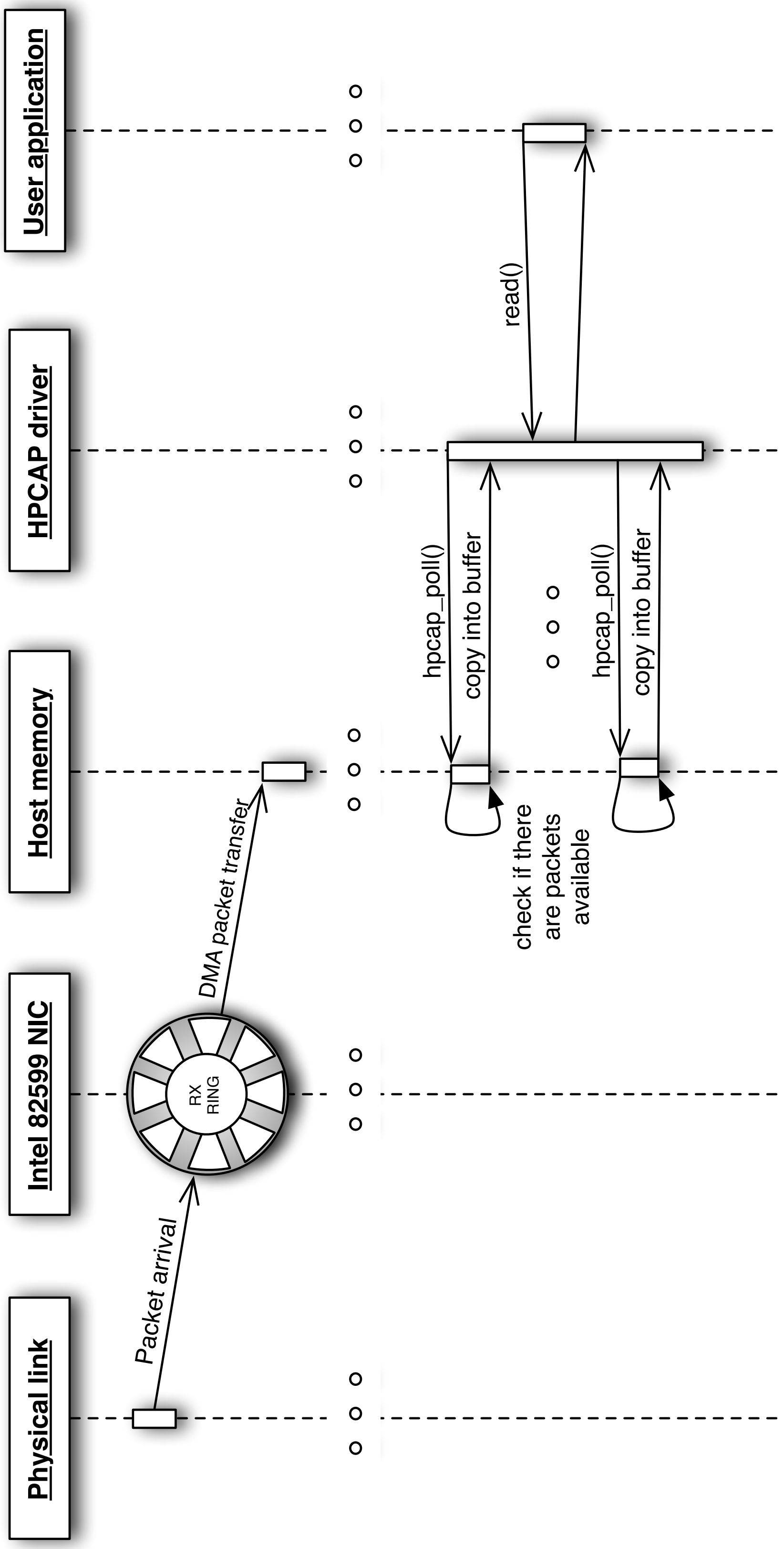


Figure 3.4: HPCAP packet reception scheme

4 Storage performance evaluation

As explained in chapter 3, one of the requirements of our approach was achieving full-rate packet storage in non-volatile volumes. Along this chapter, the validation and performance evaluation process regarding this issue will be exposed. All experiments described in this chapter have been carried out using just one queue of the Intel 82599 NIC, and taking the affinity issues mentioned in section 2.2 into account.

4.1 Validation of HPCAP

First of all, HPCAP's capture correctness has been verified. To accomplish such task, a FPGA-based traffic generator has been used, specifically, a HitechGlobal HTG-V5TXT-PCIe card which containing a Xilinx Virtex-5 FPGA (XC5VTX240) and four 10GbE SFP+ ports [21]. This card contains a VHDL design allowing to load PCAP network traffic traces into the device, and then sending such traffic both at the trace's original speed or at the link's maximum speed. On the receiver side, a two six-core Intel Xeon E52630 processors running at 2.30 GHz with 124 GB of DDR3 RAM machine, running a Ubuntu 12.04 LTS with a 3.2.0-23-generic kernel, was used.

This infrastructure has been used to validate HPCAP's packet reception of real traffic from a Tier-1 link (i.e., a CAIDA OC192 trace [22]), both sent at the trace's original speed and at the link's maximum speed. Details about the installation an launch of HPCAP are explained in appendix B.

Once HPCAP's packet capture correctness was validated, an analysis regarding its packet capture throughput was carried out. This analysis involves network traffic capture and copy into system memory. We have tested our capture system by sending constant-size packets at the maximum link's speed using the previously mentioned FPGA sender. Those tests can be easily carried using the following command,in which the Linux `dd` tool copies the incoming data into memory, and then writes them into the `null` device (a null write):

```
dd if=/dev/hpcap_xge0_0 of=/dev/null bs=12M count=10000000
```

Code 4.1: Command used to check HPCAP's memory packet capture performance

HPCAP's buffer size is a key point, the bigger the available buffer is, the more robust will be the capture when network or disk write performance peaks appear. The maximum buffer size that the Linux 3.2.0-23-generic kernel allows to allocate in 1 Gbyte. It must be pointed out that this buffer allocation has not been dynamically achieved (i.e. by means of the `kmalloc()` function), but statically (i.e. by declaring a `buffer[BUFF_SIZE]` object in the driver's code).

The reason is that `kmalloc()` does not allow to get bigger buffers than 32 Mbyte unless a full kernel tuning and recompilation is made. This modification is highly not recommended by the the kernel developing experts, as low-level factors like memory page size and others must be taken into account. Thus, the size of the HPCAP's buffer used along all the experiments is 1 Gbyte.

Table 4.1 shows the results of those experiments with a 500 seconds duration each. For each packet size, we have tested the system's packet capture performance with both the timestamp mechanism on and off. Note that when the packet size is small enough, disabling packet timestamping turns into a throughput improvement.

Pack. size(bytes)	Tstamp	Mpps sent	Mpps Captured	Gbps Captured	%Lost
60 (min)	off	14.88	14.88	7.14	0
	on		14.23	6.83	4.58
120	off	8.69	8.69	8.33	0
	on		8.69	8.33	0
500	off	2.39	2.39	9.54	0
	on		2.39	9.54	0
1000	off	1.22	1.22	9.77	0
	on		1.22	9.77	0
1500	off	0.82	0.82	9.84	0
	on		0.82	9.84	0

Table 4.1: HPCAP packet capture to system memory throughput (2.30 Ghz machine)

In the light of such results, it can be inferred that when the incoming packets are very small, the per-packet process and copy procedure becomes the system's bottleneck. This hypothesis has been confirmed by carrying out equivalent experiments in a faster machine, specifically in a Intel Xeon X5660 running at 2.80 Ghz. In table 4.2 the results of running HPCAP in such faster machine are exposed, showing a three orders of magnitude decrease in packet loss rate. Such table shows as well a lossless capture scheme for minimum sized packets due to the reduction of the total number of timestamp requests. Some results regarding how this partial timestamp policy affect accuracy are explained in chapter 5.

Pack. size(bytes)	Tstamp	Mpps sent	Mpps Captured	Gbps Captured	%Lost
60 (min)	off	14.88	14.88	7.14	0
	half		14.88	7.14	0
	on		14.88	7.14	$4.88 \cdot 10^{-3}$
1500	off	0.82	0.82	9.84	0
	on		0.82	9.84	0

Table 4.2: HPCAP packet capture to system memory throughput (2.80 Ghz machine)

Table 4.3 shows the results for equivalent experiments carried out using the PacketShader I/O engine. It must remarked that PacketShader does not timestamp the incoming packets, so a code modification have been made in order to include packet timestamping inside the

PacketShader. This code modification has been added at the most ”natural” point: the packet is timestamped at the same moment that it is copied from the RX descriptor ring memory to the host’s memory. The application used to carry such experiments is the `rxdump` tool provided within the PacketShader source code [23], with a slight modification in order to avoid performance penalties due to printing the incoming packets into an output file.

Pack. size(bytes)	Tstamp	Mpps sent	Mpps Captured	Gbps Captured	%Lost
60 (min)	off	14.88	14.63	7.03	1.63
	on		14.13	6.78	5.06
120	off	8.69	8.69	8.33	0
	on		8.66	8.33	0.25
500	off	2.39	2.39	9.54	0
	on		2.39	9.54	0
1000	off	1.22	1.22	9.77	0
	on		1.22	9.77	0
1500	off	0.82	0.82	9.84	0
	on		0.82	9.84	0

Table 4.3: PacketShader packet capture to system memory throughput

Comparison between tables 4.1 and 4.3 shows that HPCAP slightly outperforms PacketShader when it comes to capturing the incoming packets into system memory. When it comes to capturing incoming packets without an associated timestamp, HPCAP can achieve line-rate capture regardless packet size. If minimum-sized packet timestamp is required, HPCAP is only capable of capturing 95% of the incoming packets, just as PacketShader does. Nevertheless, not as PacketShader, the use of HPCAP allows line-rate packet timestamp and capture if the packets are greater than 120 bytes.

4.2 Optimizing non-volatile write performance

The amount of data that our system needs to keep is huge, specifically 10 Gb/s means 4.5 Terabyte per hour, and thus 108 Terabyte of data per each day of capture. Those figures limits the usage of high-throughput storage volumes such as SSD (Solid-Sate Drive), due to the economical cost that the amount of such disks would mean in order to store not even a full day data, but a couple of hours. We have used instead a set of HDD, whose characteristics are detailed in table 4.4.

Vendor:	Hitachi
Model number:	HUA723030ALA640 (0F12456)
Interface:	SATA 6Gb/s
Capacity:	3TB
Max. sustained sequential transfer:	152 MB/s

Table 4.4: Hard disk drives specifications

We have checked the write throughput that such disks offer, by means of the following command:

```
dd if=/dev/zero of=<HDD mount point>/testXX.zeros oflag=direct bs=1M
     count=1000000
```

Code 4.2: Command used to test a HDD's write throughput

In that terms, we have obtained an average 112 MB/s throughput. Thus, we estimate that at least 12 of such disks would be needed with the aim of creating an storage system capable of consuming incoming packets at a 10 Gb/s rate[1]. This number of disks is precisely the maximum amount of storage drives that the host chassis can handle. Further than 12 disks have been temporary mounted in such chassis in order to carry the experiments described in this section.

As our aim is maximizing the storage volume's write throughput and capacity, we have joined the amount of HDDs into a RAID 0 volume. In order to choose the right file system to be mounted over the RAID volume, several options have been taken into account: etx4, JFS and XFS. Those three file systems where taken into account as a result of being the best choices according a file system performance study that members of the HPCN group carried out under the MAINS European project [24]. Nevertheless, the already proven scalability and support for both large files and large number of files of the XFS file system [25, 26] made us choose such file system. Moreover, by means of the `logdev` mount-time parameter, XFS supports sending all the fyle system journalling information to an external device, which improves the overall file system performance.

To create such a RAID volume, two different RAID controller cards were available, whose characteristics are included in table 4.5 and 4.6.

[1] $\frac{10Gb/s}{152MB/s} = \frac{1.25GB/s}{112MB/s} = 11.16$

Vendor:	Intel
Model number:	RS25DB080
Cache memory:	1GB 800 MHz ECC DDR3
Max. internal ports:	8
Interface:	PCIe 2.0 x8

Table 4.5: Intel's RAID controller card specifications

Vendor:	Adaptec
Model number:	6405
Cache memory:	512MB 667 MHz DDR2
Max. internal ports:	1
Interface:	PCIe 2.0 x8

Table 4.6: Adaptec's RAID controller card specifications

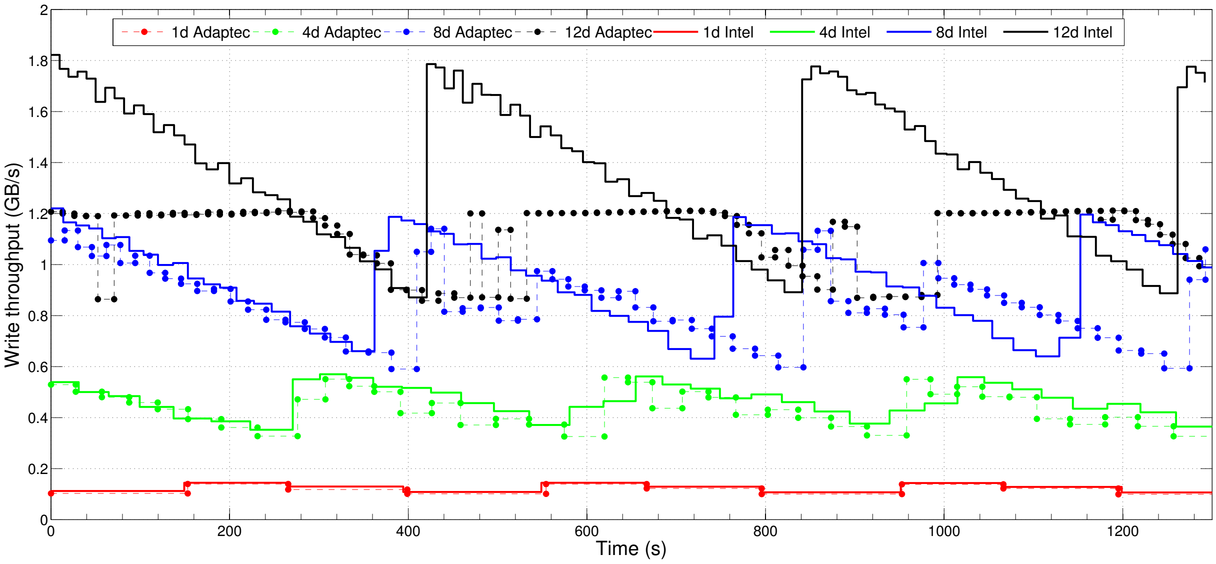


Figure 4.1: Write throughput for the Intel and Adaptec RAID controllers

The performance profile for both RAID controller cards has been tested. To accomplish such task, the previously described `dd` based command to write several have been repeated over time, writing the results into different files. Some of those results are shown in Fig. 4.1, for detailed results see appendix C. Such results in conjunction with the ones shown in Fig. 4.3(a) expose that the Adaptec RAID controller suffers from a channel limitation of 1.2GB/s. Intel RAID controller offers better write performance, and thus is the one chosen to be included in the final system. Consequently, the rest of the write performance experiments have been carried out using the Intel RAID controller card.

Additionally, Fig. 4.1 shows a periodic behaviour for both controller cards. The writing process throughput begins at its maximum level and then this throughput is successively reduced until reaching its minimum value. Once this minimum throughput is reached, the system goes back to the maximum and then this cycle is indefinitely repeated.

On appendix C it is explained that several experiments have been carried out in order to mitigate such spurious effect, those experiments reveal that this spurious effect may be due to some file system configuration parameter. A better understanding of this effect is left as future work. Mounting and umounting the file system every time a number of files is written (e.g. 4) has proven to mitigate such variation. Fig.4.2 shows the effect of such action in the system's write throughput over time. Moreover, in Fig. 4.3(b) and 4.3(c) the oscillation reduction in the write throughput due to the application of such mount/umount technique is shown.

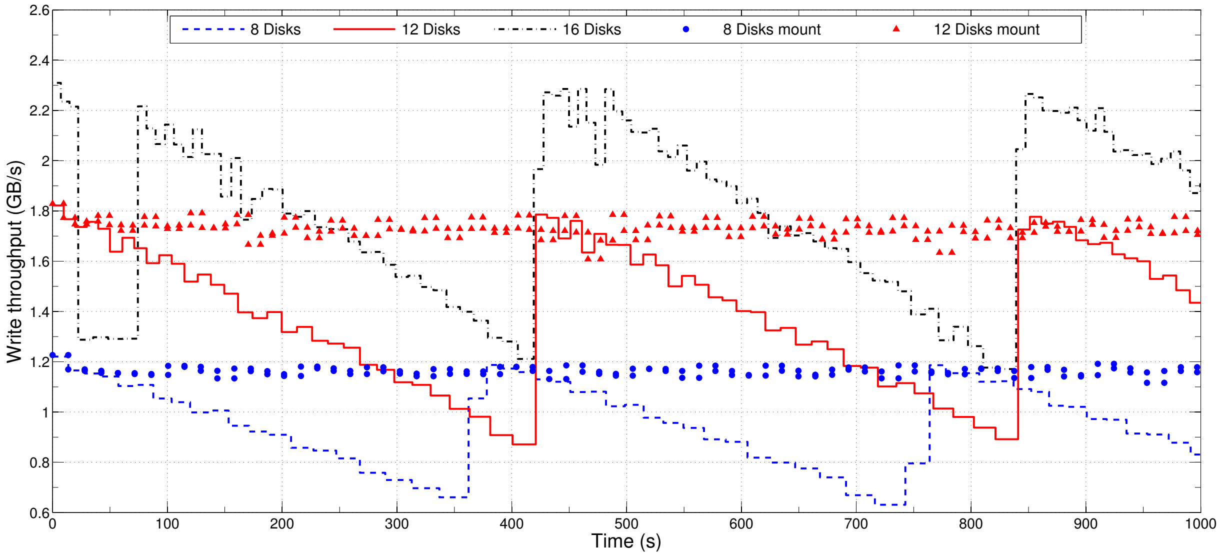
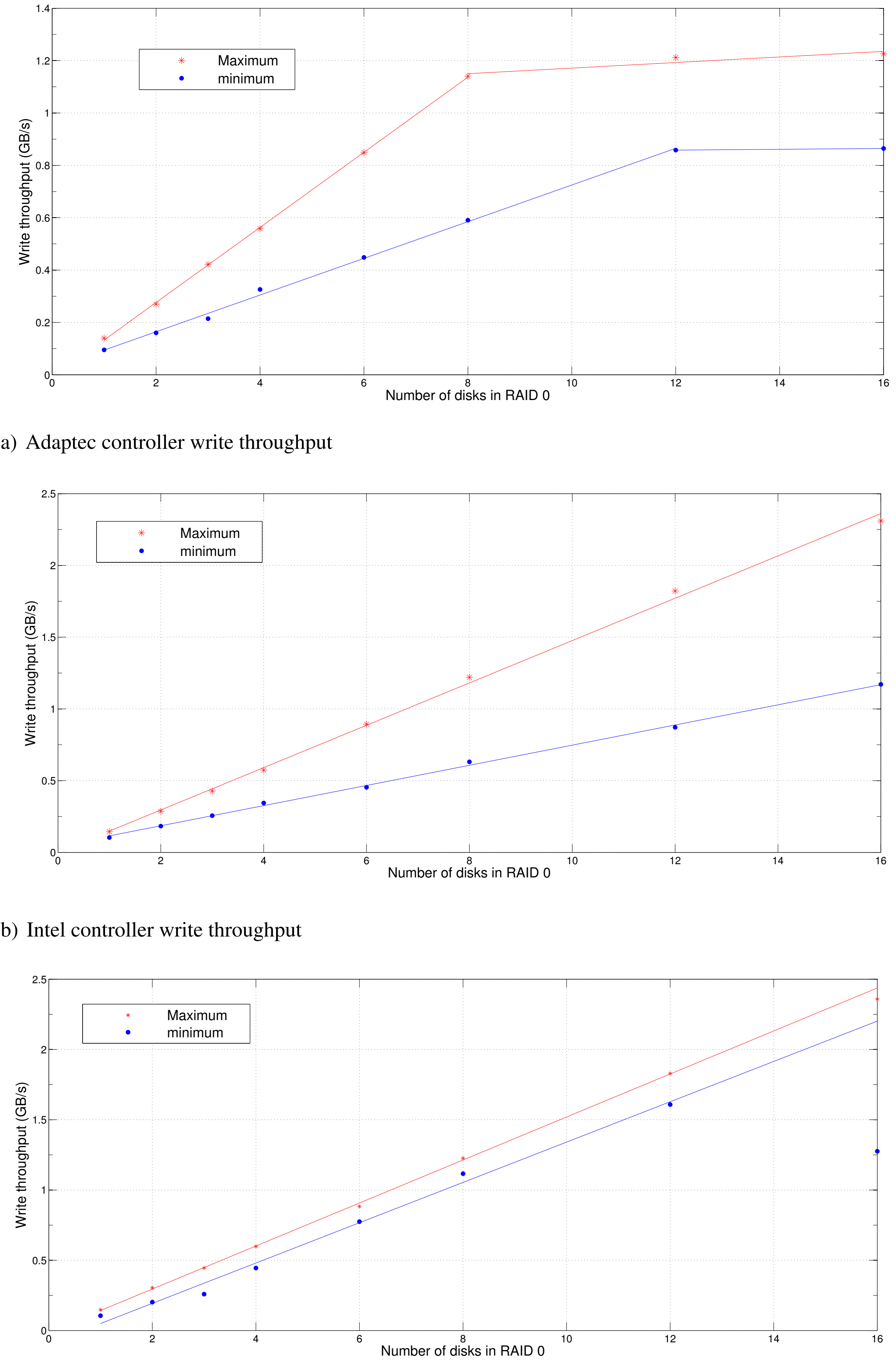


Figure 4.2: Effect of mounting the filesystem every 4 written files

We have found no evidence of this spurious effect in the related bibliography. However, we consider that such effect is quite relevant and should be taken into account in any write throughput oriented system. We believe that the use of some premium features offered by the Intel RAID controller, such using SSD drives as intermediate cache, may help to mitigate this effect. Nevertheless, the lack of resources has not allowed us to carry such experiments.



(a) Adaptec controller write throughput

(b) Intel controller write throughput

(c) Intel controller write throughput mounting the FS every 4 files

Figure 4.3: Maximum and minimum write throughput for the Intel RAID card running an XFS file system

4.3 HPCAP write performance

Once both the RAID and file system configurations have been established as in the previous section, we proceed to experiment the full system's capture throughput. The infrastructure used to carry such experiments is the same as described is section 4.1

Firstly, the packet capture and storage performance of PacketShader has been tested. Tables 4.7 and 4.8 show the results of experiments with a duration of 500 seconds. For table 4.7 we have tested the capture performance of the system storing all the captured traffic in a standard PCAP file format. The results shown in table 4.8 refer to the system capture performance when storing the incoming traffic in a raw file format (similar to the one used by HPCAP, explained in section 3.1).

Pack. size(bytes)	Tstamp	Mpps sent	Mpps Captured	Gbps Captured	%Lost
60 (min)	off	14.88	7.26	3.48	51.22
	on		7.04	3.37	52.68
120	off	8.69	5.41	5.19	37.66
	on		5.38	5.15	38.17
500	off	2.39	20.01	8.06	15.54
	on		1.99	7.99	16.27
1000	off	1.22	1.09	8.76	10.28
	on		1.09	8.72	10.73
1500	off	0.82	0.74	8.98	8.84
	on		0.74	8.99	8.82

Table 4.7: PacketShader packet capture to RAID0 (12 disks) in PCAP format

Pack. size(bytes)	Tstamp	Mpps sent	Mpps Captured	Gbps Captured	%Lost
60 (min)	off	14.88	11.71	5.62	21.29
	on		11.51	5.53	22.56
120	off	8.69	7.18	6.89	17.26
	on		7.13	6.84	17.88
500	off	2.39	2.23	8.94	6.32
	on		2.21	8.83	7.44
1000	off	1.22	1.17	9.28	4.4
	on		1.16	9.22	4.91
1500	off	0.82	0.78	9.26	4.5
	on		0.77	9.21	6.1

Table 4.8: PacketShader packet capture to RAID0 (12 disks) in RAW format

Those figures show the improvement experienced due to the usage of a raw file format in comparison with the standard PCAP file. In order to be able to process the stored traffic with standard tools, a raw to PCAP file conversion may be needed. However, we consider that this should not be a common situation, as those traces should only be retrieved when a monitoring application pops an alarm event.

In addition to the performance boost experienced due to the use of a raw file format, there is another key flaw that the PCAP file format suffers from: it does not support timestamp accuracy below one microsecond. This point is of capital relevance in 10 Gbps networks, in which a packet can be transferred in between 67.2 and 1230 ns (see Eq. 5.2).

Table 4.9 shows the write performance results obtained under the same conditions by HPCAP.

Pack. size(bytes)	Tstamp	Mpps sent	Mpps Captured	Gbps Captured	%Lost
60 (min)	off	14.88	14.76	7.08	0.73
	on		1.34	6.43	9.91
120	off	8.69	8.49	8.15	2.15
	on		8.47	8.14	2.33
500	off	2.39	2.30	9.18	3.76
	on		2.28	9.15	3.9
1000	off	1.22	1.17	9.33	4.34
	on		1.17	9.33	4.41
1500	off	0.82	0.79	9.40	4.52
	on		0.79	9.36	4.93

Table 4.9: HPCAP packet capture to RAID0 (12 disks)

Results show once again that HPCAP slightly improves the performance profile given by a PacketShader application writing in a raw file format. Nevertheless, it must me noticed that the packet loss experienced by HPCAP present a bursty behaviour. That is, all the packets lost by HPCAP are lost contiguously.

It is worth remarking the non-intuitive behaviour that HPCAP's capture presents as the packet size grows. The bigger the incoming packets are, the bigger the packet loss rate is. Notice that this effect did not appeared when fetching the incoming packets into system memory (see Table. 4.1), so now the RAID writing becomes the system bottleneck. The reason for which this effects is more evident as the packet size increases, is that the bigger the packets are, the more bandwidth demanding the write process will be in terms of bytes per second.

We consider that a bigger driver-level buffer would help to protect the capture system from packet loss when the write throughput falls. As explained in section 4.1, Linux does not allow to allocate a buffer bigger thab 1GB. We propose the addition of ram-based file system as intermediate storage solution to behave as a virtually bigger memory buffer an thus reduce packet loss. Some preliminary results of such approach using a TMPFS file system [27] are exposed in table 4.10, showing a decrease in the packet loss rate up the level shown by the packet capture to system memory.

Pack. size(bytes)	Tstamp	Mpps sent	Mpps Captured	Gbps Captured	%Lost
60 (min)	off	14.88	14.88	7.14	0
	on		14.23	6.83	4.58
1500	off	0.82	0.82	9.84	0
	on		0.82	9.84	0

Table 4.10: HPCAP packet capture to RAID0 (12 disks) using a in intermediate TMPFS storage

5 TIMESTAMP ACCURACY EVALUATION

As it has been stated in the previous chapters, Novel I/O capture engines try to overcome the flaws of general-purpose network stacks by applying techniques such as:

- per-packet memory pre-allocation,
- packet-level batch processing and
- zero-copy accesses between kernel and user space.

While these improvements boost up the performance of packet capture engines, surprisingly, packet timestamp capabilities have been shifted to the background, despite their importance in monitoring tasks. Typically, passive network monitoring requires not only capturing packets but also labelling them with their arrival timestamps. In fact, the packet timestamp accuracy is relevant to the majority of monitoring applications but it is essential in those services that follow a temporal pattern. As an example, in a VoIP monitoring system, signalling must be tracked prior to the voice stream. Moreover, the use of packet batches as a mechanism to capture traffic causes the addition of a source of inaccuracy in the process of packet timestamping. Essentially, when a high-level layer asks for packets the driver stores and forwards them to the requestor at once. Therefore, all the packets of a given batch have nearby timestamps whereas inter-batch times are huge, not representing real interarrival times. This phenomenon has not received attention to date.

Consequently, this chapter assesses the timestamp accuracy of novel packet capture engines and proposes two different approaches to mitigate the impact of batch processing. Specifically,

- two simple algorithms to distribute inter-batch time among the packets composing a batch (UDTS/WDTS), and
- a driver modification using a kernel-level thread which constantly polls NIC packet buffers and avoids batch processing (KPT).

Finally, our results, using both synthetic traffic and real traces, highlight the significant timestamping inaccuracy added by novel packet I/O engines, and show how our proposals overcome such limitation, allowing us to capture correctly timestamped traffic for monitoring purposes at multi-10Gbps rates.

5.1 The packet timestamping problem

Dealing with high-speed networks claims for advanced timing mechanisms. For instance, at 10 Gbps a 60-byte sized packet is transferred in 67.2 ns (see Eq. 5.1 and 5.2), whereas a 1514-byte packet in 1230.4 ns. In the light of such demanding figures, packet capture engines should implement timestamping policies as accurately as possible.

$$\boxed{TX_{time} = \frac{(Preamble + Packet_{size} + CRC + Inter\_Frame\_Gap)}{Link_{rate}}} \tag{5.1}$$

$$\boxed{TX_{time} = \frac{(8 + 60 + 4 + 12)\frac{bytes}{packet} \times 8\frac{bits}{byte}}{10^{10}\frac{bits}{second}} = 67.2 \times 10^{-9}\frac{seconds}{packet}} \tag{5.2}$$

All capture engines suffer from timestamp inaccuracy due to kernel scheduling policy because other higher priority processes make use of CPU resources. Such problem becomes more dramatic when batch timestamping is applied. In that case, although incoming packets are copied into kernel memory and timestamped in a 1-by-1 fashion, this copy-and-timestamp process is scheduled in time quanta whose length is proportional to the batch size. Thus, packets received within the same batch will have an equal or very similar timestamp. In Fig. 5.1 this effect is exposed for a 100%-loaded 10 Gbps link in which 60-byte packets are being received using PacketShader [2], i.e., a new packet arrives every 67.2 ns (black dashed line). As shown, packets received within the same batch do have very little interarrival time (corresponding to the copy-and-timestamp duration), whereas there is a huge interarrival time between packets from different batches. Therefore, the measured interarrival times are far from the real values. We notice that PFQ [4] does not use batch processing at driver-level and this source of inaccuracy does not affect its timestamping. However, timestamp inaccuracy may be added due to the per-packet processing latency.

At the same time, other sources of inaccuracy appear when using more than one hardware queue and trying to correlate the traffic dispatched by different queues. On the one hand, interarrival times may even be negative due to packet reordering as shown in [9]. On the other hand, the lack of low-level synchronism among different queues must be taken into account as different cores of the same machine cannot concurrently read the timestamp counter register [28]. PFQ suffers from these effects because it must use multiple queues in order to achieve line-rate packet capture. However, batch-oriented drivers, such as PacketShader, are able to capture wire-speed traffic using just one hardware queue.

Although Linux can timestamp packets with sub-microsecond precision by means of kernel `getnstimeofday` function, drift correction mechanisms must be used in order to guarantee long-term synchronization. This is out of the scope of this chapter as it has already been solved by methods like NTP (Network Time Protocol), LinuxPPS or PTP (Precision Time Protocol) [29].

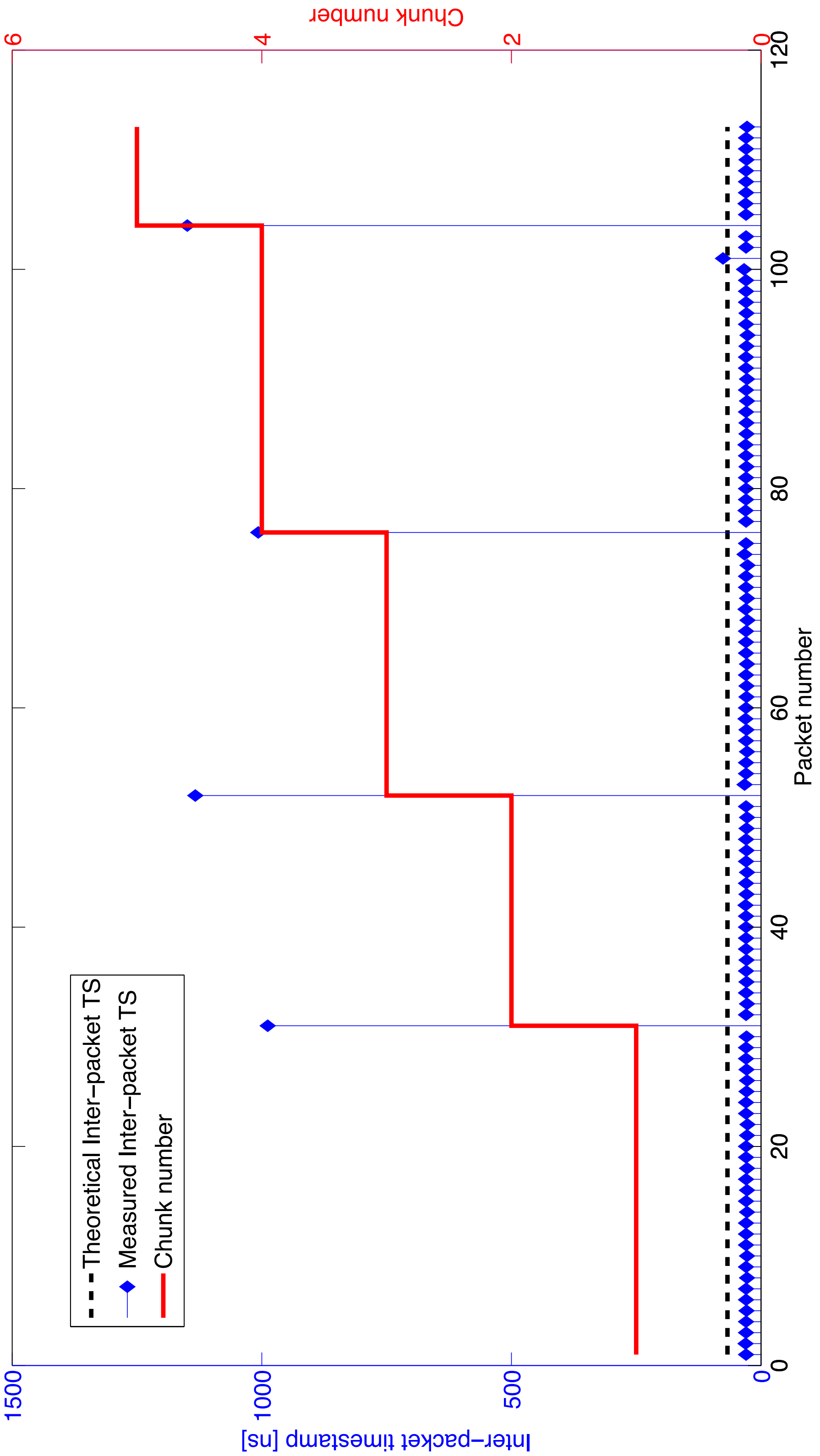


Figure 5.1: Batch timestamping

5.2 Proposed Solutions

To overcome the problem of batch timestamping, we propose three approaches. The first two ones are based on distributing the inter-batch time among the different packets composing a batch. The third approach adopts a packet-oriented paradigm in order to remove batch processing without degrading the capture performance.

5.2.1 UDTS: Uniform Distribution of TimeStamp

The simplest technique to reduce the huge time gap between batches is to uniformly distribute inter-batch time among the packets of a batch. Equation 5.3 shows the timestamp estimation of the i-th packet in the $(k+1)$-th batch, where $t_m^{(j)}$ is the timestamp of the m-th packet in the j-th batch and n_j is the number of packets in batch j.

$$\boxed{\tau_i^{(k+1)} = t_{n_k}^{(k)} + \left(t_{n_{k+1}}^{(k+1)} - t_{n_k}^{(k)}\right) \cdot \frac{i}{n_{k+1}} \ \forall i \in \{1, \ldots, n_{k+1}\}} \tag{5.3}$$

As shown in Fig. 5.2a, this algorithm performs correctly when the incoming packets of a given batch have the same size. A drawback of this solution is that all packets of a given batch have the same inter-arrival time regardless of their size (see Fig. 5.2b). Note that the inter-packet gap is proportional to the packet size when transmitting packets at maximum rate.

5.2.2 WDTS: Weighted Distribution of TimeStamp

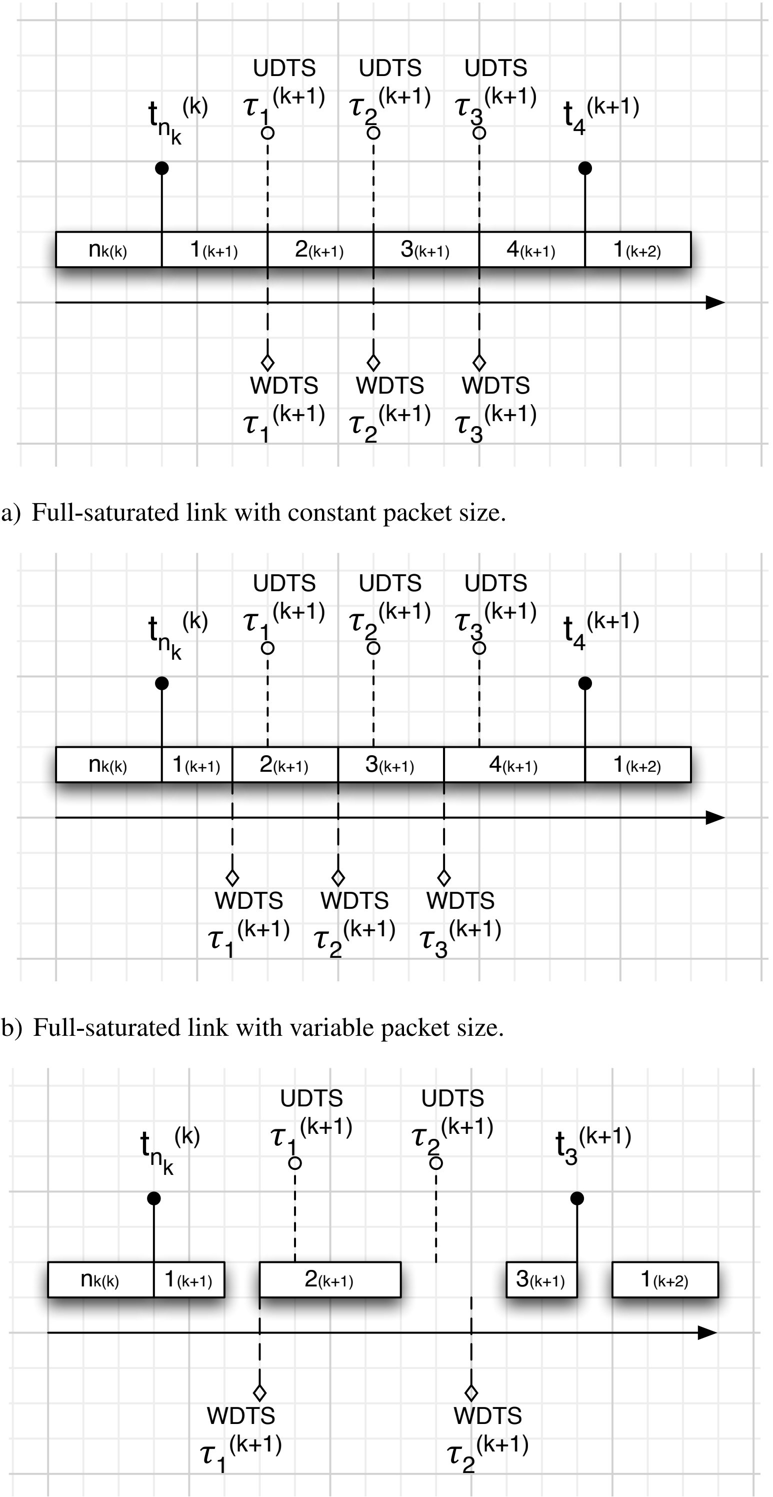
To overcome the disadvantage of previous solution, we propose to distribute time among packets proportionally to the packet size. Equation 5.4 shows the timestamp estimation using this approach, where $s_j^{(k+1)}$ is the size of the j-th packet in the $(k+1)$-th batch.

$$\boxed{\tau_i^{(k+1)} = t_{n_k}^{(k)} + \left(t_{n_{k+1}}^{(k+1)} - t_{n_k}^{(k)}\right) \cdot \frac{\sum_{j=1}^{i} s_j^{(k+1)}}{\sum_{j=1}^{n_{k+1}} s_j^{(k+1)}} \ \forall i \in \{1, \ldots, n_{k+1}\}} \tag{5.4}$$

WDTS is especially accurate when the link is completely loaded because there are no inter-packet gaps (excluding transmission time), regardless the packet size is variable, as shown in Fig. 5.2b. However, when the link load is lower, both UDTS and WDTS present poorer results as they distribute real inter-packet gaps among all the packets in the batch (see Fig. 5.2c). That is, the lower the inter-packet gap is, the higher the accuracy is.

5.2.3 KPT: Kernel-level Polling Thread

Towards a timestamping approach that performs properly regardless the link load, we propose a redesign of the network driver architecture. Novel packet capture engines fetch packets



(a) Full-saturated link with constant packet size.

(b) Full-saturated link with variable packet size.

(c) Not full-saturated link with variable packet size.

Figure 5.2: Inter-packet gap distribution

from the NIC rings only when a high-level layer polls for packets, then they build a new batch of packets and forward it to the requestor. This architecture does not guarantee when will the fetcher thread be scheduled and consequently, a source of uncertainty is added to the timestamping mechanism.

As explained in chapter 3, our proposal is to implement a kernel-level thread which constantly polls the NIC rings for new incoming packets and then timestamps and copies them into a kernel buffer. A high-level application will request the packets stored in the kernel buffer, but the timestamping process will no longer be dependent on when applications poll for new packets. This approach reduces the scheduling uncertainty as the thread will only leave execution when there are no new incoming packets or a higher priority kernel task needs to be executed. KPT causes a higher CPU load due to its busy waiting approach, but it does not degrade the performance to the point that packets are lost.

5.3 Accuracy evaluation

Our setup consists of two servers (one for traffic generation and the other for receiving traffic) directly connected through a 10 Gbps fiber-based link. The receiver has two six-core Intel Xeon E52630 processors running at 2.30 GHz with 124 GB of DDR3 RAM. The server is equipped with a 10GbE Intel NIC based on 82599 chip, which is configured with a single RSS queue to avoid multi-queue side-effects, such as reordering or parallel timestamping. The sender uses a HitechGlobal HTG-V5TXT-PCIe card which contains a Xilinx Virtex-5 FPGA (XC5VTX240) and four 10GbE SFP+ ports [21]. Using a hardware-based sender guarantees accurate timestamping in the source. For traffic generation, two custom designs have been loaded allowing: (i) the generation of tunable-size Ethernet packets at a given rate, and, (ii) the replay of PCAP traces at variable rates.

As first experiment, we assess the timestamp accuracy sending traffic at maximum constant rate. Particularly, we send 1514-byte sized packets at 10 Gbps, i.e., 812,744 packets per second and measure the interarrival times in the receiver side. Table 5.1 shows the error of the measured timestamp (i.e., the difference between the original and the observed interarrival times), in terms of mean and standard deviation, for a 1-second experiment (to make sure no packets are lost) for the different reviewed methods. Note that the lower the standard deviation is, the more accurate the timestamping technique is. The first two rows show the results for PacketShader, chosen as a representative of batch-based capture engines. We tested with different batch sizes and different timestamping points: at user-level or at driver-level. PFQ results are shown in the following row whereas the three last ones show the results of our proposed solutions. It can be observed that timestamping error grows with batch size. Even using one-packet batches, the error is greater than the one observed using our proposals. UDTS and WDTS methods enhance the accuracy, decreasing the standard deviation of the timestamp error below 200 ns. Both methods present similar results because all packets have the same size in this experiment. KPT technique reduces the standard deviation of the error up to ∼600 ns. Despite timestamping packet-by-packet, PFQ shows a timestamp standard error greater than 13 μs.

Solution	Batch size	$\bar{\mu} \pm \bar{\sigma}$ [ns]
User-level batch TS	1	2 ± 1765
	32	2 ± 3719
Driver-level batch TS	1	2 ± 1742
	32	2 ± 3400
PFQ	-	2 ± 13558
UDTS	32	2 ± 167
WDTS	32	2 ± 170
KPT	-	2 ± 612

Table 5.1: Experimental timestamp error (mean and standard deviation). Synthetic traffic: 1514-bytes packets

In the next experiments, we evaluate the different techniques using real traffic from a Tier-1 link (i.e., a CAIDA OC192 trace [22]). We perform two experiments: in the first one, the trace is replayed at wire speed (that is, at 10 Gbps), and then, we replay the trace at the original speed (i.e., at 564 Mbps, respecting inter-packet gaps). Due to storage limitation in the FPGA sender, we are able to send only the first 5,500 packets of the trace. Table 5.2 shows the comparison of the results for our proposals and the driver-level batch timestamping. We have used a batch size of 32 packets because 1-packet batches do not allow achieving line-rate performance for all packet sizes. In wire-speed experiments, WDTS obtains better results than UDTS due to different sized packets in a given batch. When packets are sent at original speed, WDTS is worse than KPT because WDTS distributes inter-packet gap among all packets. This effect does not appear at wire-speed because there is no inter-packet gap (excluding transmission time). In any case, driver-level batch timestamping presents the worst results, even in one order of magnitude.

Solution	**Wire-Speed**	**Original Speed**
	$\bar{\mu} \pm \bar{\sigma}$ **[ns]**	$\bar{\mu} \pm \bar{\sigma}$ **[ns]**
Driver-level batch TS	13 ± 3171	-26 ± 19399
UDTS	12 ± 608	-40 ± 13671
WDTS	5 ± 111	-42 ± 14893
KPT	-1 ± 418	-43 ± 1093

Table 5.2: Experimental timestamp error (mean and standard deviation). Real traffic: Wire-speed and Original speed

5.4 Conclusion

Batch processing enhances the capture performance of I/O engines at the expense of packet timestamping accuracy. We have proposed two approaches to mitigate timestamping degradation:

- UDTS/WDTS algorithms that distribute the inter-batch time gap among the different packets composing a batch and
- a redesign of the network driver, KPT, to implement a kernel-level thread which constantly polls the NIC buffers for incoming packets and then timestamps and copies them into a kernel buffer one-by-one.

In fact, we propose to combine both solutions according to the link load, i.e., using WDTS when the link is near to be saturated distributing timestamp in groups of packets and, otherwise, using KPT timestamping packet-by-packet. We have stress tested the proposed techniques, using both synthetic and real traffic, and compared them with other alternatives achieving the best results (standard error of 1 μs or below).

To summarize, we alert research community to timestamping inaccuracy introduced by novel high-performance packet I/O engines, and proposed two techniques to overcome or mitigate such issue.

6 Conclusions and future work

This work has presented how do standard NICs, such as the Intel 82599, work and how some novel I/O capture engines exploit some of their characteristics in order to obtain high packet capture performance. None of those existing capture engines meet all the features that we consider a good packet capture engine should do. This has motivated the development of HPCAP, a new packet capture engine that meets all those requirements.

A performance study has been made regarding the packet storage in non-volatile volumes of HPCAP, taking into account several configuration details that affects throughput. Preliminary results obtained running the HPCAP capture system in faster machines (2.80 Ghz against 2.30 Ghz of the machine used along this work) conform our hypothesis of a bottleneck in the packet processing scheme. As future work, HPCAP capture performance could be improved by means of applying techniques regarding user application directly mapping kernel memory to allow zero-copy access from the user to the network data, thus allowing the upper level applications to run heavier processes over the incoming packets without packet loss. Jumboframe support has also been left as future work.

In addition, the timestamp quality of our approach has been evaluated, a key point that had not received attention up to date. We show that HPCAP timestamp capabilities greatly outperform the existing solutions ones. However, HPCAP timestamp accuracy could still be improved by combining its timestamping procedure with techniques such as WDTS (Weighted Distribution of TimeStamp). Note that the application of such techniques would also reduce the processing requirements of the system and thus improve packet capture performance. This is left as future work.

HPCAP is a solution that is being currently used for traffic monitoring in industrial environments such as Telefonica, British Telecom and Produban (enterprise that belongs to ”Grupo Santander”) Data Centers.

Moreover, this work has lead to the creation of the following works:

- "Batch to the Future: Analyzing Timestamp Accuracy of High-Performance Packet I/O Engines". Sent to the "IEEE Communications Letters" magazine in June 2012.
- "On VoIP data retention and monitoring at multi-Gb/s rates using commodity hardware". Sent to the "Journal of Network and Computer Applications" in July 2012.

Bibliography

[1] Cisco, "Cisco carrier routing system." http://cisco.com/en/US/products/ps5763/index.html. 1

[2] S. Han, K. Jang, K. S. Park, and S. Moon, "PacketShader: a GPU-accelerated software router," *ACM SIGCOMM Computer Communication Review*, vol. 40, no. 4, pp. 195–206, 2010. 1, 2.1, 2.1.4, 2.2, 2.2, 3.1, 5.1

[3] L. Rizzo, M. Carbone, and G. Catalli, "Transparent acceleration of software packet forwarding using netmap," in *Proceedings of IEEE INFOCOM*, 2012. 1, 2.1

[4] N. Bonelli, A. Di Pietro, S. Giordano, and G. Procissi, "On multi-gigabit packet capturing with multi-core commodity hardware," in *Proceedings of Passive and Active Measurement Conference*, 2012. 1, 2.1, 2.2, 5.1

[5] G. Liao, X. Znu, and L. Bnuyan, "A new server I/O architecture for high speed networks," in *Proceedings of IEEE Symposium on High-Performance Computer Architecture*, 2011. 1, 2.1.4

[6] F. Schneider, J. Wallerich, and A. Feldmann, "Packet capture in 10-gigabit ethernet environments using contemporary commodity hardware," in *Proceedings of the 8th International Conference on Passive and Active Network Measurement*, vol. 4427, (New York, NY, USA), p. 207–217, Springer-Verlag Berlin Heidelberg, April 2007. 1

[7] Intel, "Intel ® 82599 10 GbE Controller Datasheet," *October*, no. December, 2010. 2.1, 2.1.2

[8] M. Corporation, " Receive-Side Scaling Enhancements in Windows Server 2008," 2008. 2.1.2

[9] W. Wu, P. DeMar, and M. Crawford, "Why can some advanced Ethernet NICs cause packet reordering?," *IEEE Communications Letters*, vol. 15, no. 2, pp. 253–255, 2011. 2.1.2, 2.2, 5.1

[10] M. Dashtbozorgi and M. Abdollahi Azgomi, "A scalable multi-core aware software architecture for high-performance network monitoring," in *Proceedings of the 2nd international conference on Security of information and networks*, SIN '09, (New York, NY, USA), pp. 117–122, ACM, 2009. 2.1.2, 2.2

[11] The Linux Foundation, "Napi." http://www.linuxfoundation.org/collaborate/workgroups/networking/napi. 2.1.3

[12] C. Benvenuti, *Understanding Linux Network Internals*. O'Reilly, 2005. 2.1.3

[13] W. Wu, M. Crawford, and M. Bowden, “The performance analysis of linux networking - packet receiving,” *Comput. Commun.*, vol. 30, pp. 1044–1057, Mar. 2007. 2.1.4

[14] L. Braun, A. Didebulidze, N. Kammenhuber, and G. Carle, “Comparing and improving current packet capturing solutions based on commodity hardware,” in *Proceedings of Internet Measurement Conference*, (Melbourne, Australia), pp. 206–217, Nov. 2010. 2.2

[15] L. Rizzo, “Revisiting network I/O APIs: the netmap framework,” *Communications of the ACM*, vol. 55, pp. 45–51, Mar. 2012. 2.2, 2.2, 3.1

[16] The Single UNIX Specification, “dd: convert and copy a file.” http://www.gnu.org/software/coreutils/manual/html_node/dd-invocation.html. 3.1

[17] R. Love, “Kernel Locking Techniques.” http://james.bond.edu.au/courses/inft73626@033/Assigs/Papers/kernel_locking_techniques.html. 3.2

[18] G. K.-H. Jonathan Corbet, Alessandro Rubini, *Linux Device Drivers 3rd Edition*. O’Reilly, 2005. 3.3, 3.3, 6

[19] littledaemons.org, “CPU Affinity and taskset.” http://littledaemons.wordpress.com/2009/01/22/cpu-affinity-and-taskset/. 3.4

[20] L. P. Manual, “PTHREAD_SETAFFINITY_NP.” http://www.kernel.org/doc/man-pages/online/pages/man3/pthread_setaffinity_np.3.html. 3.4

[21] H. Global, “NetFPGA10G.” http://www.hitechglobal.com/boards/PCIExpress_SFP+.htm. 4.1, 5.3

[22] C. Walsworth, E. Aben, k. claffy, and D. Andersen, “The CAIDA anonymized 2009 Internet traces.” http://www.caida.org/data/passive/passive_2009_dataset.xml. 4.1, 5.3

[23] S. Han, K. Jang, K. S. Park, and S. Moon, “Packet I/O Engine.” http://shader.kaist.edu/packetshader/io_engine/index.html. 4.1

[24] T. M. Project, “Metro Architectures enablINg Sub-wavelengths.” http://www.ist-mains.eu/. 4.2

[25] R. Wang and T. Anderson, “xfs: a wide area mass storage file system,” in *Workstation Operating Systems, 1993. Proceedings., Fourth Workshop on*, pp. 71 –78, oct 1993. 4.2

[26] A. Sweeney, D. Doucette, W. Hu, C. Anderson, M. Nishimoto, and G. Peck, “Scalability in the xfs file system,” in *Proceedings of the 1996 annual conference on USENIX Annual Technical Conference*, ATEC ’96, (Berkeley, CA, USA), pp. 1–1, USENIX Association, 1996. 4.2

[27] P. Snyder, “tmpfs: A virtual memory file system,” in *In Proceedings of the Autumn 1990 European UNIX Users’ Group Conference*, pp. 241–248, 1990. 4.3

[28] T. Broomhead, J. Ridoux, and D. Veitch, “Counter availability and characteristics for feed-forward based synchronization,” in *Proceedings of IEEE Symposium on Precision Clock Synchronization for Measurement Control and Communication*, 2009. 5.1

[29] M. Laner, S. Caban, P. Svoboda, and M. Rupp, “Time synchronization performance of desktop computers,” in *Proceedings of IEEE Symposium on Precision Clock Synchronization for Measurement Control and Communication*, 2011. 5.1

[30] SuperMicro, “X9DR3-F.” http://www.supermicro.com/products/motherboard/xeon/c600/x9dr3-f.cfm. A

Lists

List of equations

List of figures

List of tables

Acronyms

API	Application Program Interface
CPU	Central Processing Unit
DMA	Direct Memory Access
FPGA	Field Programmable Gate Array
Gbps	Gigabit per second
HDD	Hard-Disk Drive
HPCAP	High-performance Packet CAPture
I/O	Input/Output
IP	Internet Protocol
ISP	Internet Service Provider
ixgbe	Intel’s 10 Gigabit Ethernet Linux driver
KPT	Kernel-level Polling Thread
Mbps	Megabit per second
MP	Multiprocessor
Mpps	Millions of packets per second
NAPI	New API
NIC	Network Interface Card
NTP	Network Time Protocol
NUMA	Non Uniform Memory Access
PCAP	Packet Capture API
PCIe	Peripheral Component Interconnect Express
PTP	Precision Time Protocol
RSS	Receive Side Scaling
RX	Reception
SMART	Self-Monitoring, Analysis and Reporting Technology

SPMC	Single Producer, Multiple Consumer
SSD	Solid-Sate Drive
Tbps	Terabit per second
TCP	Transmission Control Protocol
TS	Timestamp
TX	Transmission
UDP	User Datagram Protocol
UDTS	Uniform Distribution of TimeStamp
VHDL	Virtual Hardware Description Language
VoIP	Voice over IP
WDTS	Weighted Distribution of TimeStamp

TERMINOLOGY

batch processing
Processing together a group of packets, thus reducing the overhead of performing a system call for each packet in the batch.

commodity hardware
”Commodity” hardware is hardware that is easily and affordably available. A device that is said to use ”commodity hardware” is one that uses components that were previously available or designed and are thus not necessarily unique to that device.

flow
Cisco standard NetFlow version 5 defines a flow as an unidirectional sequence of packets that all share of the following 7 values: ingress interface, source and destination IP addresses, IP protocol, source and destination ports for UDP or TCP (or 0 for other protocols) and IP Type of Service (ToS).

zero-copy
”Zero-copy” describes computer operations in which the CPU does not perform the task of copying data from one memory area to another. For example, a user-space application that maps a kernel memory buffer can the data stored in such without explicitly copying its content into its memory address-space, thus reducing the access overhead and improving its memory accesses efficiency.

A

Architecture overview

All the experiments described in this work have been carried out using a SuperMicro machine with a X9DR3-F motherboard [30]. Details about such motherboard can be found in Fig. A.1 and A.2.

Information provided by such diagrams must be taken into account in order to achieve peak performance: the network card must be plugged in a PCIe slot attached to the CPU where the HPCAP driver will be later on executed (or vice-versa).

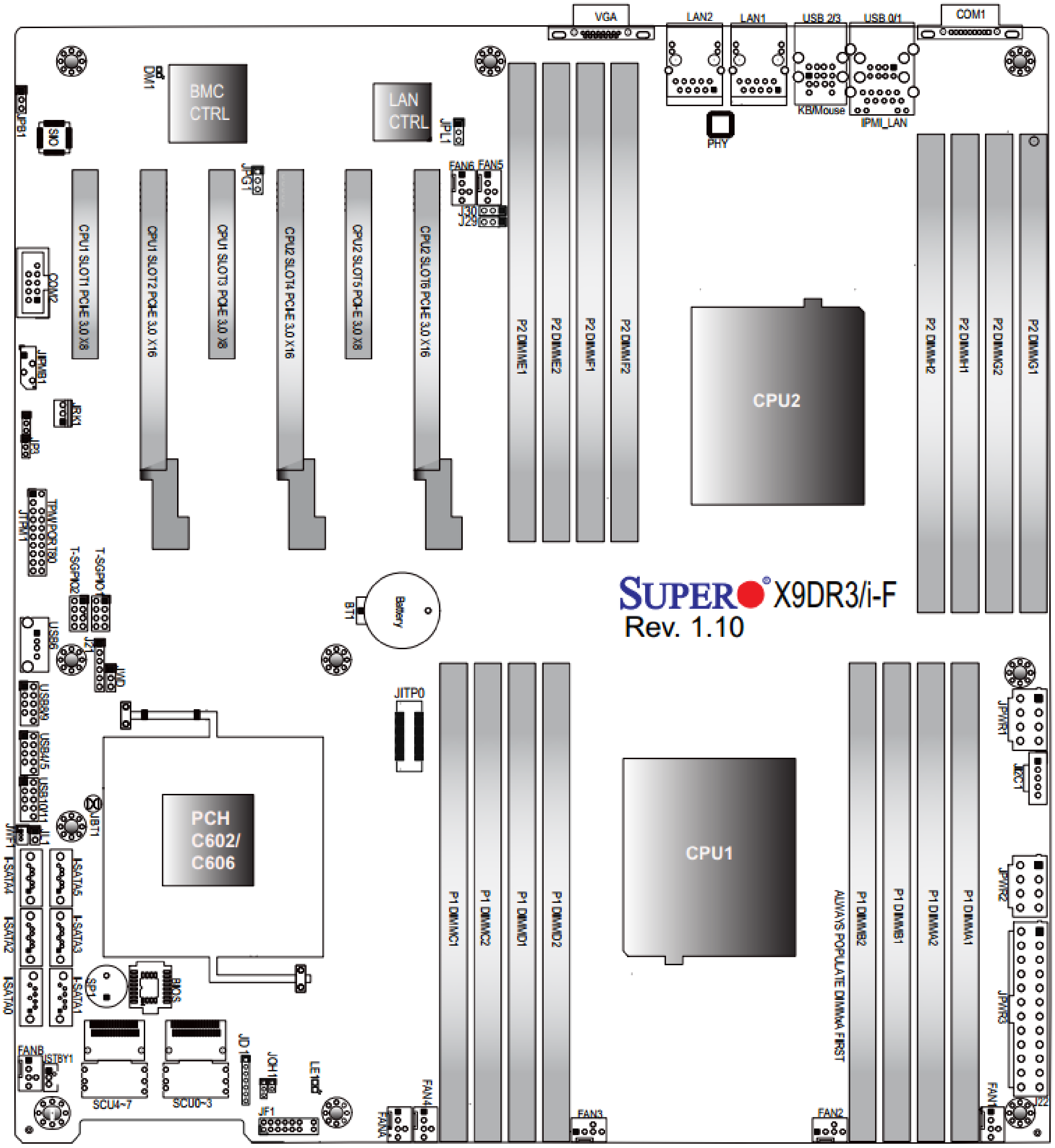


Figure A.1: Supermicro X9DR3-F motherboard

As an example, during our experiments we plugged our Intel 82599 NIC in the CPU1-slot2 PCIe port, and thus mapped our capture systems into the CPU1's cores (seen by operating system as cores from 0 to 5).

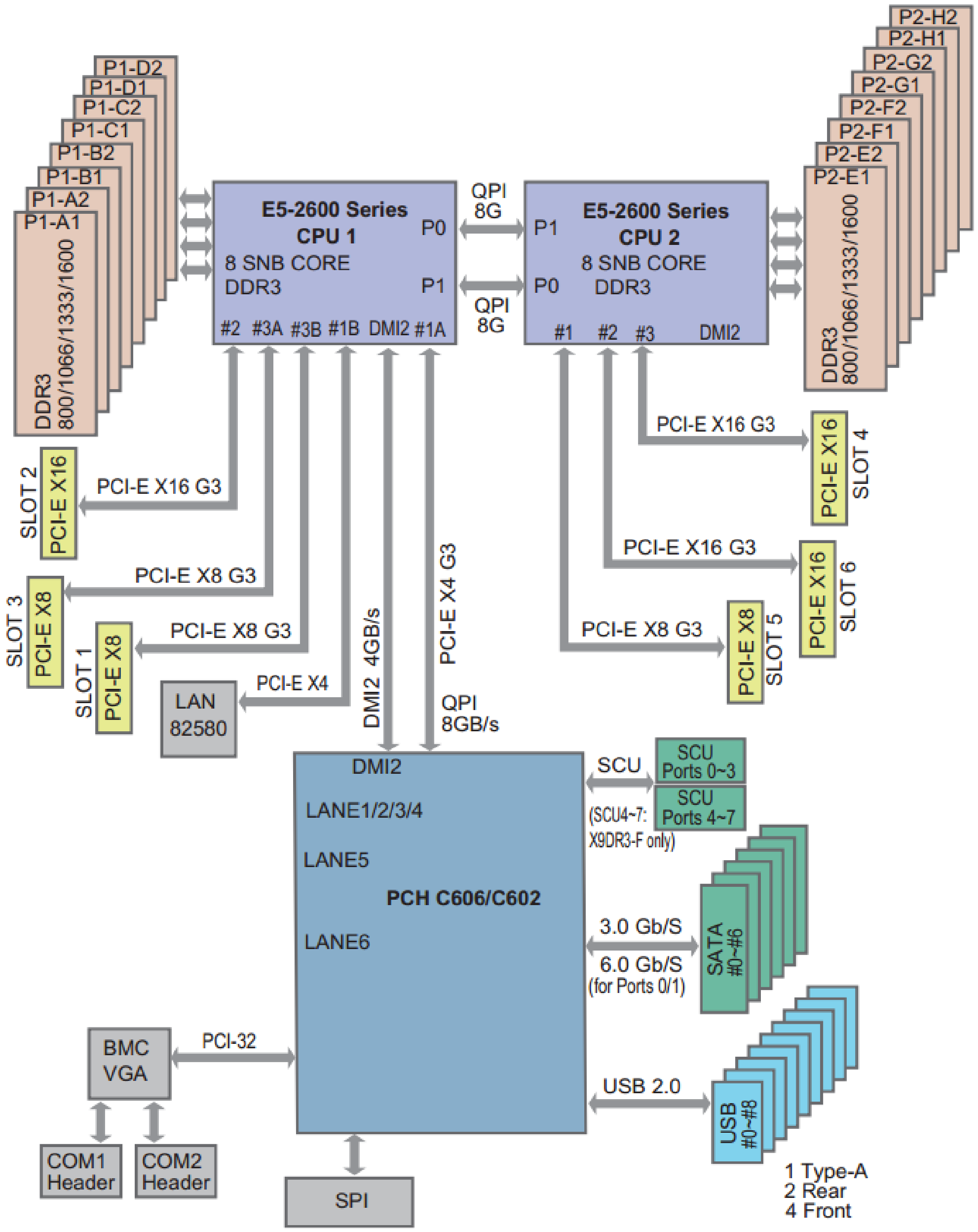


Figure A.2: Supermicro X9DR3-F's block diagram

B

How to run HPCAP

The installation of HPCAP follows a straightforward procedure: it only requires the user to copy the HPCAP folder into the host system. Once such folder has been copied, the user has to access to it (older contents are shown below) and execute the install_hpcap.bash script.

```
...
copy.bash
data
hpcap_ixgbe-3.7.17_buffer
      driver
      samples
install_hpcap.bash
params.cfg
...
```

Code B.1: Contents of the HPCAP folder

The previously mentioned installation script will compile the driver if required, and load into the system. Note that the following packets must be installed in your system in order to run HPCAP:

- numactl
- libnuma-dev
- libnuma1
- linux-libc-dev
- make

Some configuration parameters can be modified by means of writing into de params.cfg file (the interfaces to be used, number of queues per interface, core affinities, link speed,...). Before the installation script is launched, the user must make sure that the corresponding filesystem is created in the desired output mount point, and that this mount point is specified inside the params.cfg file.

The default behaviour of the installation script is launching an instance of the copy.bash script per each NIC's hardware queue.

C

RAID0 WRITE TESTS

C.1 RAID scalability

Fig. C.2 and C.1 show the results carried out to test the write scalability of a XFS RAID0 volume when adding disks to it. Those test contrast the results obtained for both the Adaptec and the Intel controller cards, whose characteristics have been already detailed in tables table 4.6 and 4.5 respectively. Those test expose the channel limitation that the Apactec controller suffers from. They furthermore expose a cyclic behaviour in terms of the write throughput. Section C.3 takes a deeper look to such effect.

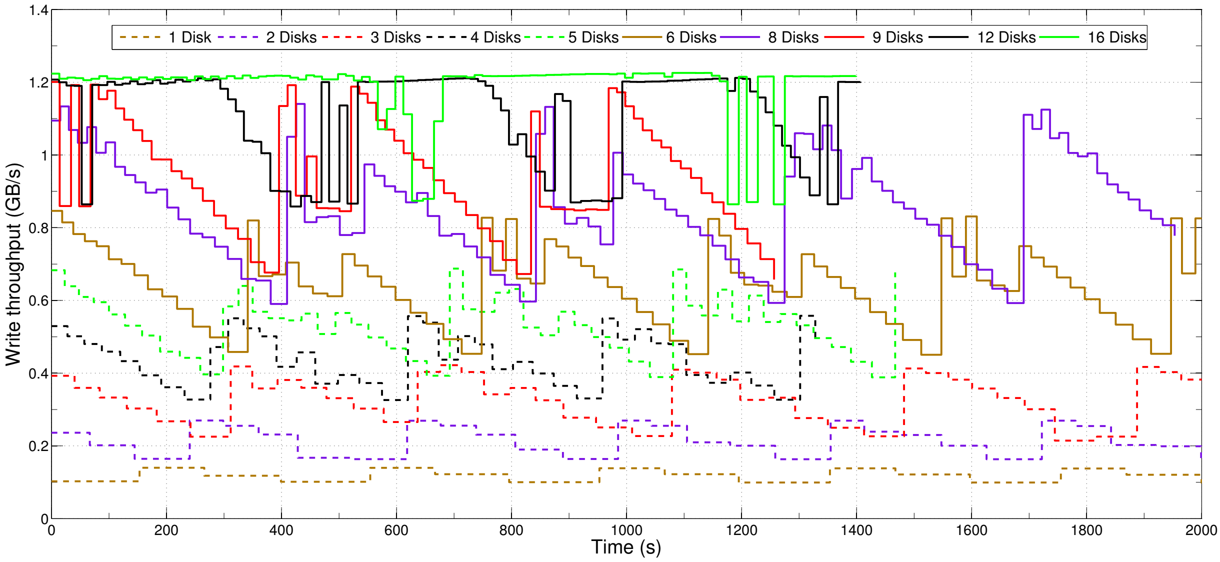


Figure C.1: Adaptec RAID0 write scalability

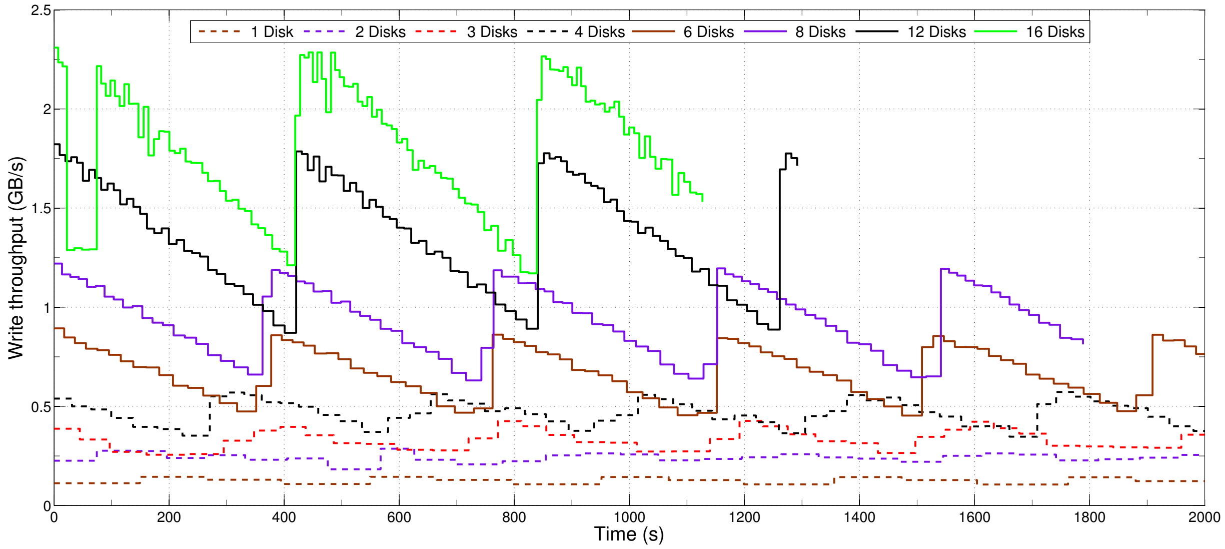


Figure C.2: Intel RAID0 write scalability

C.2 Optimal block size

Some tests have been carried out as well regarding the effect of the size of the written block (one of the arguments of the dd tool used in conjunction with HPCAP). Fig. C.3 shows the results of such experiments.

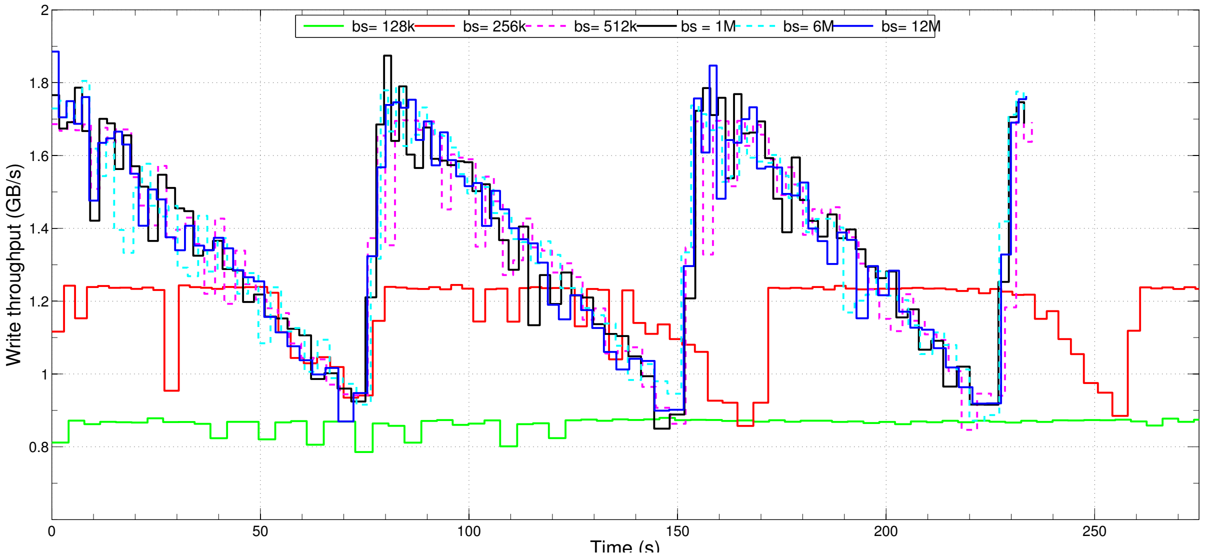


Figure C.3: Effect of the block size on write performance

Results show that best performance is obtained with a block size above 512KB. We have thus chosen a block size of 12MB to carry the rest of our experiments. This choice's justification is to keep both the RAID0 strips and the written block aligned (keep in mind that we have as 12 disks RAID0 volume, with a block size of 1MB and thus a strip size of 12MB).

C.3 Cyclic throughput variation

We have tried to eliminate the RAID write throughput periodic behaviour shown in Figs. 4.1, C.2 and C.1, by means of tuning the following configuration parameters of both the RAID controller and the file system:

- Rebuild rate: This field allows the user to determine the priority for the RAID rebuild operation. This option selects the amount of system resources devoted to rebuilding failed disk drives. A higher percentage rate rebuilds drives faster, but can degrade the system performance. Setting the rebuild rate to 0 will prevent the array from rebuilding. Its default value is 30.
- SMART (Self-Monitoring, Analysis and Reporting Technology) poll interval: A SMART drive monitors the internal performance of the motors, media, heads, and electronics of the drive, while a software layer monitors the overall reliability status of the drive. The reliability status is determined through the analysis of the drive's internal performance level and the comparison of internal performance levels to predetermined threshold limits. A smaller poll interval means that the disks will prioritize application I/O over SMART diagnostics. Its default value is 300 seconds.
- Additionally, the ability of the XFS file system to mount the journalling data to a ”ramdisk” has been tested.

Nevertheless, the modification of those configuration parameters had no effect on the write throughput variation. We have carried out a further analysis on this effect, obtaining how the write throughput varies regarding the size of the files written. The results of these tests, shown in Figs. C.4 and C.5. Note that the writing of each file is marked as a circle in Fig. C.4.

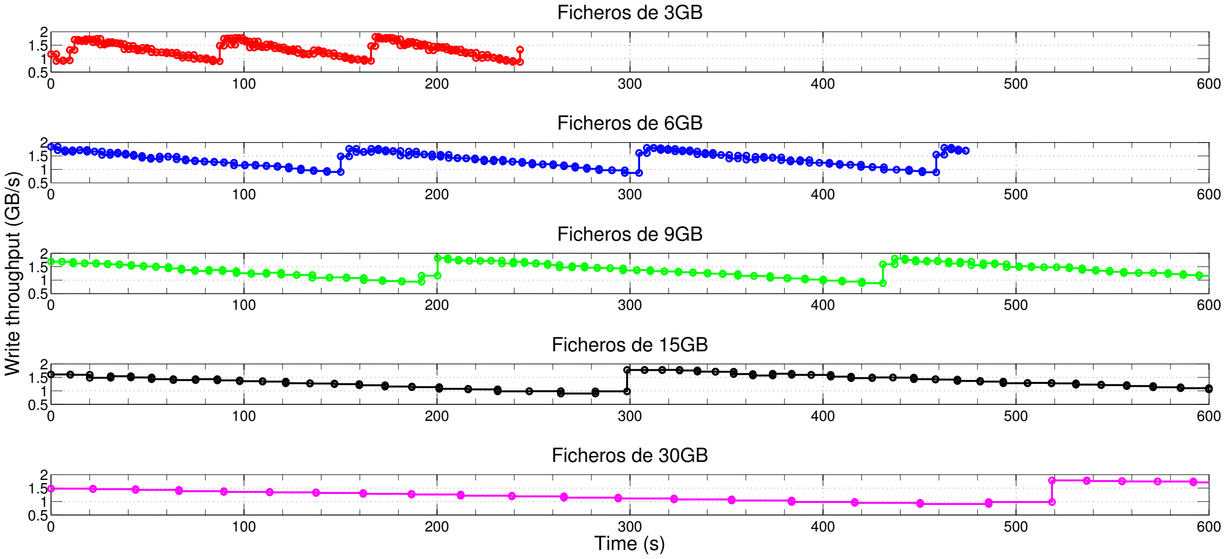


Figure C.4: Effect of the written file size on the write cycle

Fig.C.4 shows the write performance profile observer when writing different-sized files into our file systems. All those experiments have been made with a fixed block size of 12Mb over a

12-disk RAID 0 volume. Those results show that the performance oscillation does not follow a periodical pattern in terms of time elapsed, but in terms of number of files written. This effect can be easily appreciated in Fig. C.5, where the write throughput is painted versus the number of file being written. This graphic shows that regardless of the size of the files being written, the volume's write throughput experiences a cycle every 30 files.

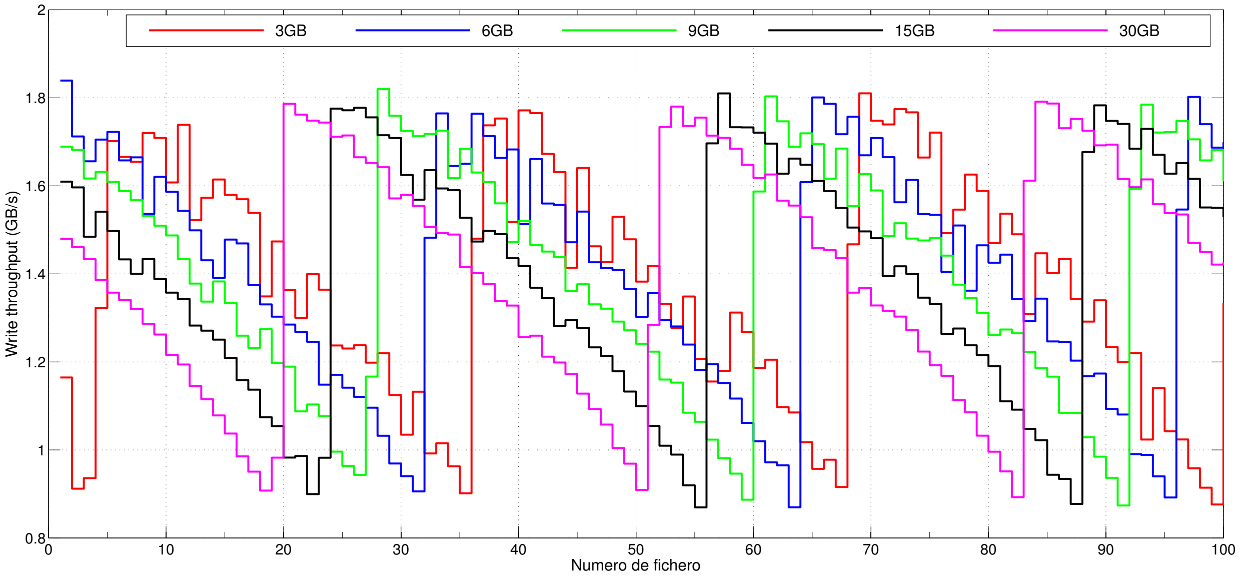


Figure C.5: Effect of the written file size on the write cycle

In the light of such results, we hypothesize that this effect is produced by some kind of reset/refresh in the XFS file system. We have only been able to mitigate that problem by means of forcing an file system mounts every time a certain number of files are written (see Fig. C.6).

Consequently, HPCAP writing experiments carried out along section 4.3 have taken this effect into account and applied the mount/umount measure in order to optimize overall performance.

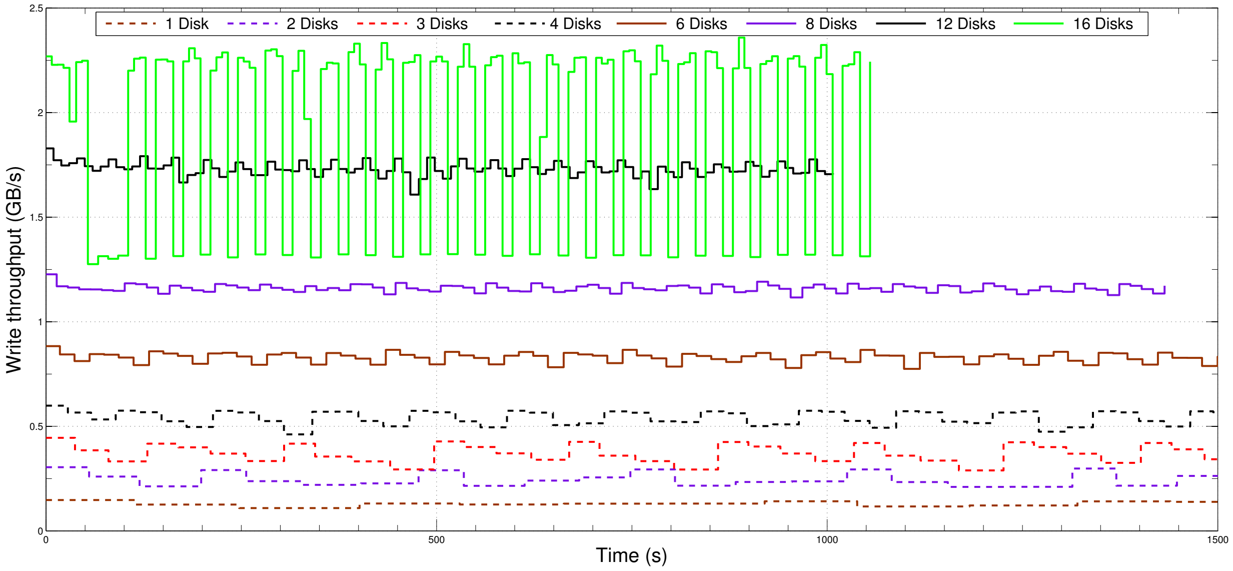


Figure C.6: Intel RAID0 write scalability(mounting the filesystem every 4 files)

Index