Universidad Autónoma de Madrid
Escuela Politécnica Superior

Thesis to obtain PhD. degree in
Computer and Telecommunication Engineering
by
Universidad Autónoma de Madrid

Thesis advisor:
Dr. Francisco Javier Gómez Arribas

Harnessing low-level tuning in modern architectures for high-performance network monitoring in physical and virtual platforms

Víctor Moreno Martínez

This thesis was presented on May 2015

Tribunal:

Dr. Jaime H. Moreno
Dr. Sandrine Vaton
Dr. David Fernández Cambronero
Dr. Mikel Izal Azcárate
Dr. Iván González Martínez

Víctor Moreno Martínez
Harnessing low-level tuning in modern architectures for high-performance network monitoring in physical and virtual platforms

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

IMPRESO EN ESPAÑA – PRINTED IN SPAIN

A mis padres, Teresa y Fernando, por ser un ejemplo de vida.
A Merce, por compartir la suya conmigo.

Every man dies. Not every man really lives.
William Wallace.

Thesis evaluators:

Dr. Jaime H. Moreno
(Chairman)

Dr. Sandrine Vaton

Dr. David Fernández Cambronero

Dr. Mikel Izal Azcárate

Dr. Iván González Martínez

Defense date:

Calification:

Table of contents

1 Introduction **1**

1.1 Overview and motivation 1

1.2 Objectives 5

1.3 Thesis structure 7

2 Architectures for network monitoring **9**

2.1 Hardware components 10

2.1.1 Application-Specific Integrated Circuits 10

2.1.2 Field Programmable Gate Arrays 11

2.1.3 General-Purpose Graphical Processing Units 12

2.1.4 Commodity hardware 14

2.2 Background experience 15

2.2.1 ARGOS 15

2.2.2 Twin^{-1} 18

2.3 Conclusions 20

3 Packet capture using commodity hardware **21**

3.1 Commodity Hardware 22

3.1.1 NUMA architectures 25

3.1.2 Current and past operating system network stacks 26

3.2 Packet capturing 33

3.2.1 Limitations: wasting the potential performance 33
3.2.2 How to overcome limitations 36
3.3 Capture Engine implementations 41
3.3.1 PF_RING DNA 41
3.3.2 PacketShader 45
3.3.3 netmap 47
3.3.4 PFQ 48
3.3.5 Intel DPDK 51
3.3.6 HPCAP 52
3.4 Testing your traffic capture performance 53
3.4.1 General concerns 53
3.4.2 Captures engines performance evaluation 55
3.5 Use cases of novel capture engines 64
3.5.1 Creating a high-performance network application 64
3.5.2 Application examples 66
3.6 Conclusions 73

4 HPCAP implementation details and features **77**

4.1 HPCAP's design 78
4.1.1 Kernel polling thread 78
4.1.2 Multiple listeners 81
4.1.3 User-level API 82
4.1.4 HPCAP packet reception scheme 83
4.2 Packet timestamping 86

4.2.1 Accuracy issues 86
4.2.2 Performance evaluation 93
4.3 Packet storage 99
4.3.1 Motivation 100
4.3.2 Storing data on hard-drives 101
4.3.3 Network traffic storage solutions 109
4.4 Duplicates detection and removal 112
4.4.1 Accuracy 114
4.4.2 Performance 116
4.5 Conclusions 120

5 M^{3}OMon: a framework on top of HPCAP 123

5.1 Introduction 123
5.1.1 Novel features: multi-granular / multi-purpose 124
5.1.2 High-performance in off-the-self systems 125
5.1.3 Contributions 127
5.2 System Overview 128
5.2.1 M^{3}Omon 130
5.2.2 M^{3}Omon's API 132
5.3 Performance Evaluation Results 133
5.4 Industrial application samples 137
5.4.1 DetectPro 137
5.4.2 VoIPCallMon 139
5.5 Related Work 144

5.6 Conclusions 146

6 Network monitoring in virtualized environments 147

6.1 High-performance network processing 149

6.2 Virtualized environments and I/O processing 153

6.2.1 Full-virtualization 155

6.2.2 Paravirtualization and VirtIO 155

6.2.3 PCI passthrough 159

6.2.4 PCI virtual functions 160

6.3 Virtual network probe 164

6.4 Virtual network monitoring agent 168

6.5 Conclusions 171

7 Conclusions and future work 173

7.1 Results dissemination and publications 174

7.2 Industrial applications 179

7.3 Future work 180

8 Conclusiones y trabajo futuro 183

8.1 Diseminación y divulgación de los resultados alcanzados . . 185

8.2 Aplicaciones industriales 189

8.3 Trabajo futuro 190

Bibliography 193

Glossary 208

A Capture engines' usage examples **209**

A.1 Getting started 209

A.2 Setting up capture engines 213

A.2.1 Default driver 213

A.2.2 PF_RING DNA 214

A.2.3 PacketShader 215

A.2.4 netmap 215

A.2.5 PFQ 217

A.2.6 Intel DPDK 217

A.2.7 HPCAP 218

B HPCAP manual **221**

B.1 Using the HPCAP driver 221

B.1.1 Installing all the required packages 221

B.1.2 Configure your installation 221

B.1.3 Interface naming and numbering 223

B.1.4 Per-interface monitored data 223

B.1.5 Waking up an interface in *standard* mode 224

B.1.6 Sample applications 224

B.2 Working with the RAW file format 226

B.2.1 File data structures 226

B.2.2 Example code 227

B.3 Quick start guide 229

B.3.1 Launching `hpcapdd` 230

B.3.2 Checking traffic storage 230

B.4 Frequently asked questions 231

Lists **235**

ACKNOWLEGMENTS

The present document is the fruit of the work I've been carrying out for some years, and it could not have been completed without the support of many people in both professional and personal terms:

First of all, I want to thank my supervisor **Francisco J. Gómez Arribas** for his guide since I joined the research group seven years ago. You have always been a near supervisor and a great partner, and your opinions and annotations have always proven useful. Your help has played a fundamental role in the successful accomplishment of this goal, that seemed so far away a couple of years ago. Thank you for your friendly guide, and for helping me to develop as a researcher and a professional.

This work would have not been possible without the opportunity that the *High Performance Computing and Networking Group* from the Universidad Autónoma de Madrid offered me: they gave me the chance to prove myself when I was just an undergraduate student, and kept they support until the present day. Being part of this research group has also given me the chance to share moments and learn from highly skilled people such as Javier Aracil, Jorge E. López de Vergara, Iván González, Sergio López, Gustavo Sutter and Luis de Pedro. I also had luck of not only learning from but also sharing great times in the C-113 lab with incredibly valuable people, to whom I owe a special thank and maximum respect: Javier Ramos, Pedro M. Santiago, José Luis García, David Muelas, Felipe Mata, Víctor López, Joś Luis Añamuro, Jaime Garnica, Diego Sánchez and Jaime Fullaondo. I also want to thank the rest of the C113 group for the good moments shared: Rubén García-Valcárcel, Isabel García, Juan Sidrach, Rafael Leira, Carlos Vega, Paula Roquero and many others. I can't less but thank `irenerodriguez.ii.uam.es` for *her* support and great performance offered along all those years: I hope you get can some rest now!

All my work would also not have been possible without the support of the Universidad Autónoma de Madrid and the *Departamento de Tecnología Electrónica y de las Comunicaciones* of the Escuela Politécnica Superior. I also want emphasize my gratitude to the Universidad Autónoma de Madrid and the Spanish Ministry of Education for funding this Ph.D. under the F.P.I. and the F.P.U. fellowship programs respectively. I hope that this kind programs keep existing in the future in order to promote the research and talent that so much we need.

I also want to express my gratitude to Dr. Khaled Benkrid for hosting me along the three months I spent in the University of Edinburgh. This interesting

stay allowed me to acquire experience with state-of-the-art architectures.

From the personal point of view, let me change to Spanish:

Quiero agradecer a mi familia por el apoyo y confianza que siempre han mostrado en mí. Esto incluye a mis padres, Teresa y Fernando, cuya educación y apoyo me ha permitido desarrollarme como persona y como profesional. Además, no puedo más que agradecerles el ejemplo a seguir que siempre han supuesto para mí, y que siguen demostrando ser día a día. También quiero agradecer a mi hermano Fernando por los buenos momentos que hemos compartido a lo largo de toda una vida juntos, y dejar por escrito que, aunque no se lo diga muy a menudo, me siento muy orgulloso de él.

También he de agradecer a mis abuelos Teresa, María, Víctor y José (aunque no tuviera la suerte de conocerle) la grandísima influencia que han tenido en mi vida. De ellos pude aprender cómo vivir la vida, y los tengo presentes y echo en falta cada día.

No puedo evitar tampoco expresar el orgullo que siento de pertenecer a mi familia, y el cariño que siento hace todos y cada uno de sus miembros: mis tí@s, prim@s, aquell@s que sin tener vínclulo sanguíneo son tan de la familia como los demás, e incluso nuestra familia más lejana de Andalucía.

En un lugar tan relevante como el que más, he de agradecer a Merce su compañía a lo largo de estos años. Además de algún que otro cambio de humor, ha tenido que sufrir los horarios y restricciones que el llevar a cabo este trabajo han supuesto, y siempre lo ha hecho con la máxima comprensión y ofreciéndome todo el apoyo y ayuda posible. ¡Gracias por ser cómo eres! También quiero agradecer mi familia política por haberme acogido como a uno más, y por los buenos ratos que he podido pasar con ellos.

No me olvido tampoco de todos aquellos amigos que me han acompañado a lo largo de todos estos años: amigos del barrio de toda la vida, hechos en la facultad, otros que he tenido la suerte de conocer en el extranjero, ... también a todos ellos de los que los años me han alejado pero han estado allí. Con todos ellos he podido compartir momentos excepcionales de chorradas, risas y otros completamente aleatorios que han amenizado estos largos años. En especial gracias a Vicky, Roberto, Saray, Josepa, Paula, Sara, Nieves, Edu, Inma y Ana.

Thank you all! ¡Gracias a todos!

Abstract

Over the past decades, the use of the Internet has rapidly grown due to the emergence of new services and applications. The amount of them available to end-users makes it necessary for their providers to deploy quality-assessment policies in order to distinguish their product among the rest. In this scenario, network processing and analysis becomes a central task that has to deal with humongous amounts of data at high-speed rates. Service providers must be able to accomplish such a challenging task using processing elements capable of reaching the required rates while keeping the cost as low as possible for the sake of profitability. This thesis analyses the main problems and provide viable solutions when applying commodity-hardware for high-performance network processing. Furthermore, diverse systems have been developed in this line, which have also been validated in industrial environments.

Traditionally, when the requirements were tight an eye was turned to the use of ASIC designs, reprogrammable FPGAs or network processors. This work is started with a study and evaluation of diverse architectural solutions for network processing. Those solutions offer great computational power at the expense of high levels of specialization. Consequently, they only address the performance half of the problem but they fail at solving the other half, which is the inexorably need to perform more diverse, sophisticated and flexible forms of analysis. Moreover, those solutions imply high investments: such hardware's elevated cost rises capital expenditures (CAPEX), while operational expenditures (OPEX) are increased due to the difficulties in terms of operation, maintenance and evolution. Furthermore, HW life cycles become shorter as technology and services evolve, which complicates the stabilization of a final product thus reducing profitability and limiting innovation. Those drawbacks turn specialized HW solutions into a non-adequate option for large-scale network processing. Nevertheless, this thesis has also evaluated the use of this possibility using FPGA technology. Specifically, a prototype has been developed for network packet capture with accurate timestamping, reaching a tenths of nanoseconds precision and GPS synchronization.

In this light, both industry and academia have paid attention to the use of solutions based on commodity-hardware. The advantages of those systems lay in the ubiquity of those components, which makes it easy and affordable to acquire and replace them and consequently reduces CAPEX. Those systems are not necessarily cheap, but their wide-range of application allows their price to benefit from large-scale economies and makes it possible to achieve great degrees

of experimentation. Additionally, such systems offer extensive and high-quality support, thus reducing OPEX. Unfortunately, the use of commodity hardware in high-speed network tasks is not trivial due to limitations on both hardware capacities and standard operating systems' performance. Essentially, commodity hardware is limited in terms of memory and internal bus throughputs. From the software side, limitations come from a general-purpose network stack that overloads communications due to a prioritization of protocol and hardware compatibility over performance.

It is in this context in which the main contribution of this thesis, HPCAP, is presented. HPCAP is a high-performance capture engine that has been designed to face the problems not yet solved by the state-of-the-art capture engines while keeping similar performance levels. The literature references capture engines which are capable of capturing 10 Gb/s network traffic, but do not pay attention to the vital tasks, e.g.: storing this traffic onto non-volatile storage systems, accurately timestamp the traffic, or feed this traffic to diverse processing applications. Those are in fact the most relevant contributions of HPCAP to the field. We have empirically verified that if the network packets are not accurately timestamped when carrying out network monitoring tasks, the analysis made can lead to wrong conclusions. Packet timestamping accuracy is not only affected by the accuracy of the time source used, but also by the moment in which packets are timestamped: the more code is executed between the packet's arrival and its timestamping moment, the more variability and error appears. On the other hand, there are many scenarios in which after a high-level analysis over the network traffic, it is required to access to low-level packet information to identify problem sources. Consequently, keeping the packets stored for their subsequent access becomes a relevant issue. In this line, it seem reasonable to instantiate several network traffic analysis applications while an independent application is in charge of storing the traffic in non-volatile place. Nevertheless, this requirement is difficult to reach without paying the performance loss price, and this is the reason for which HPCAP has been designed taking this into full consideration. Furthermore, M^{3}Omon has been created: a general-purpose network processing framework built on top of HPCAP, whose performance as also been exhaustively tested. M^{3}Omon pretends to be a reference point for easily developing high-performance network applications, which has already been applied in the development of several projects with a direct industrial application.

Other application domain with undeniable interest is the world of virtualized platforms. If those solutions based on commodity-hardware are to be applied in realistic highly-demanding scenarios, the increased demands for network processing capacity could be translated into a big number of machines even though if off-the-shelf systems were used. Such an amount of machines means high expenses in terms of power consumption and physical space. Moreover, the presence of commodity servers from different vendors empowers the appear-

ance of interoperability issues. All those drawbacks damage the profitability that networked service providers may experience. This situation has pushed towards the applications of virtualization techniques in network processing environments, with the aim of unifying existing computing platforms. Techniques such as PCI-passthrough allow the migration of the results obtained in the physical world to the virtual one in a very direct way. This work has carried out a study regarding the impact of this technique on the previously presented systems, concluding that the performance loss experienced is appealingly low. Another approach is the one propose by an alliance composed of network operators and service providers which has introduced in the last years the concept of Network Function Virtualization (NFV). This new paradigm aims to unify the environments where network applications shall run by means of adding a virtualization layer on top of which network applications may run. This novel philosophy also allows merging independent network applications using unique hardware equipment. However, in order to make the most of NFV, mechanisms that allow obtaining maximum network processing throughput in such virtual environments must be developed. The contribution of this work in the NFV field is the evaluation of performance bounds when carrying out high-performance network tasks in NFV environment and the development of HPCAPvf as a NFV counterpart for the formerly presented HPCAP.

Keywords: High-speed networks, packet capture, packet storage, network processing, network drivers, network function virtualization, virtual machines, performance assessment.

Resumen

En las últimas décadas, el uso de Internet ha crecido vertiginosamente debido a la aparición de nuevos servicios y aplicaciones. La cantidad de los mismos obliga a sus proveedores a llevar a cabo políticas de asesoría y aseguramiento de calidad con el fin de diferenciarse de la competencia. En este contexto, el procesamiento y análisis del tráfico de red se convierte en una tarea primordial, que tiene que hacer frente a enormes cantidades de datos de diferente tipo y a tasas de velocidad muy elevadas. Los proveedores de servicio deben por tanto, superar este reto utilizando elementos de procesos capaces de alcanzar las tasas requeridas, a la vez que mantienen un coste tan bajo como sea posible, con vistas a maximizar los beneficios obtenidos. En esta tesis se estudia y analiza los principales problemas y se aportan soluciones viables aplicando hardware estándar en el procesamiento de red de alto rendimiento. Se han desarrollado diversos prototipos siguiendo estas líneas y se han validado en entornos de producción industrial.

Tradicionalmente, cuando los requisitos computacionales eran elevados o muy estrictos, se venía haciendo uso de diseños basados en Circuitos Integrados de Aplicación Específica (ASICs), hardware reconfigurable (FPGAs) o procesadores de red. Este estudio comienza haciendo una evaluación de distintas arquitecturas de cómputo aplicadas al problema del procesamiento de red. Estas soluciones ofrecen una gran potencia de cómputo a cambio de un grado de especialización muy elevado. Es decir, sólo abordan la mitad del problema relativa al rendimiento, pero se olvidan de la otra mitad: la inexorable necesidad de llevar a cabo nuevas, diversas y más sofisticadas formas de análisis. Lo que es más, estas soluciones implican unos niveles de inversión muy elevados: estos elementos hardware tienen un alto coste que incrementa notablemente la inversión inicial (CAPEX), y las complicaciones asociadas al mantenimiento, operación y evolución de estos sistemas disparan también los costes operacionales (OPEX). Además, los ciclos de vida del HW son cada vez más cortos a medida que la tecnología avanza, lo que complica la estabilización de un producto. Estas ventajas hacen de las soluciones basadas en hardware específico una opción poco factible para procesamiento de red a gran escala. No obstante, en esta tesis también se ha evaluado el uso de esta aproximación utilizando tecnología FPGA. Concretamente se ha realizado un prototipo de captura de red con marcado de tiempo sincronizado por GPS alcanzando una precisión de decenas de nanosegundos.

A la luz de estos datos, tanto la industria como el mundo académico han

dirigido su mirada hacia el uso de soluciones basadas en hardware estándar. La ventaja de estos sistemas reside en la omnipresencia de sus componentes, que hace que su adquisición o reemplazo sea sencilla y asequible, implicando una reducción en el CAPEX. Estos sistemas no son necesariamente baratos, pero su amplitud y universalidad de uso permite que su producción se beneficie de políticas económicas a gran escala, así como de elevados grados de testeo y experimentación. Además, estos sistemas ofrecen soporte extensivo y de alta calidad, lo cual supone reducciones en el OPEX. Desafortunadamente, la aplicación del hardware estándar en tareas de red a alta velocidad no es trivial debido a limitaciones inherentes al hardware y a los sistemas operativos convencionales. Esencialmente, las limitaciones a nivel hardware son la tasa de transferencia de memoria y la de los buses del sistema. Por otro lado, a nivel software los sistemas operativos utilizan una pila de red que sobrecarga las transferencias porque su diseño está orientado a la compatibilidad con un gran número de protocolos y dispositivos.

Es en este contexto en el que se presenta la principal contribución de este trabajo de tesis: el motor de captura HPCAP. Éste ha sido diseñado con el objetivo de resolver problemas que hasta el momento no resolvían otros manteniendo las mismas prestaciones. Actualmente, la bibliografía referencia motores que son capaces de capturar el tráfico de red entrante a 10 Gb/s, pero que no consideran hacer un procesamiento más exhaustivo como puede ser: guardar dicho tráfico en sistemas de almacenamiento no volátiles, asociar marcas de tiempo precisas al tráfico, o permitir que diversas aplicaciones puedan alimentarse del mismo tráfico con el mínimo sobrecoste posible. Estas son precisamente las aportaciones más relevantes de HPCAP al área de conocimiento. Hemos podido comprobar empíricamente que el hecho de no marcar con la precisión adecuada los paquetes de red entrantes puede llevar fácilmente a realizar análisis incorrectos. A la precisión del marcado de paquetes no le afecta únicamente la precisión de la fuente de tiempos utilizada, sino también el momento en el que se realiza: cuanto más código se ejecute entre la llegada del paquete y su marcado, mayor variabilidad y error existirá. Por otro lado, son numerosos los escenarios en los que tras realizar un análisis de alto nivel sobre el tráfico de red es necesario acceder a bajo nivel al mismo para obtener la causa del problema, por lo que tener almacenado el tráfico adquiere una gran relevancia. En esta línea, parece razonable la instanciación de diversas aplicaciones, independientes o no, de análisis del tráfico de red mientras que una aplicación diferente se encarga de almacenar el tráfico de forma no-volatil. Sin embargo, este requisito es dificil de alcanzar sin pagar un alto coste en la tasa de procesamiento y es por eso que el diseño de HPCAP ha sido realizado teniéndolo muy en cuenta. Adicionalmente, se ha creado M^{3}Omon: un marco de trabajo o *framework* sobre HPCAP para facilitar el desarrollo de aplicaciones de procesamiento de red, cuyo rendimiento ha sido también probado exhaustivamente.

M^{3}Omon pretende ser un punto de referencia para desarrollar de forma ágil y sencilla aplicaciones de red de altas prestaciones. De hecho, ya ha sido aplicado satisfactoriamente para el desarrollo de diversos proyectos con aplicación industrial directa.

Otro dominio de aplicación que tiene indudable interés es el mundo de las plataformas virtualizadas. A la hora de aplicar soluciones basadas en hardware estándar en redes de altas prestaciones complejas, los elevados requisitos de cómputo pueden llegar a traducirse en un gran número de máquinas. Esta cantidad de máquinas implica costes elevados en términos tanto de consumo eléctrico como de espacio físico. Además, la existencia en la misma infraestructura de elementos hardware estándar de diferentes fabricantes no hace sino aumentar la probabilidad de aparición de problemas de interoperabilidad. Dicha situación ha fomentado precisamente la aplicación de técnicas de virtualización en entornos de procesamiento de red con el objetivo de unificar las plataformas de procesamiento. Técnicas como el baipás-PCI (*PCI-passthrough* en su terminología nativa) permiten migrar de forma directa los resultados obtenidos en el mundo físico a entornos virtuales. En este trabajo se ha llevado a cabo un estudio sobre el impacto en el rendimiento de estas técnicas a los sistemas de procesamiento de red previamente presentados, concluyendo que la pérdida de rendimiento es atractivamente baja. Otra aproximación es la propuesta por una alianza formada por diversos proveedores de servicios en red, operadores, e incluso fabricantes de hardware que han introducido el concepto de Virtualización de Funciones de Red (*Network Function Virtualization*, NFV). Este nuevo paradigma pretende unificar los entornos en los que las aplicaciones de red serían ejecutadas por medio de la inserción de una capa de virtualización. Esta novedosa filosofía permite consolidar aplicaciones de red independientes utilizando un único equipamiento hardware. Sin embargo, de cara a sacar el máximo partido a la filosofía de NFV, se han de desarrollar mecanismos que permitan obtener la máxima tasa de rendimiento de red en entornos virtuales. La contribución de este trabajo en este campo, es precisamente la evaluación de los límites de rendimiento de tareas de red en entornos NFV y el desarrollo de HPCAPvf como el homólogo en el mundo NFV del motor de captura presentado, HPCAP.

Palabras clave: Redes de alta velocidad, captura de tráfico de red, almacenamiento de tráfico de red, procesamiento de red, drivers de red, virtualización de funciones de red, máquinas virtuales, evaluación de rendimiento.

1 INTRODUCTION

This chapter has the purpose of providing an overview of this Ph.D. thesis as well as introducing its motivation, presenting its objectives, and, finally, describing its main contributions and outlining its organization.

1.1 Overview and motivation

Users' demands and the capacity of both backbone and access links almost since the Internet was born, have played a daily game of cat and mouse. On the one hand, the widespread availability of the Internet is a fact. Moreover, users tend to use the Net more intensively as new applications gain significant popularity in a matter of weeks [GDFM+12]. In line with such an increase demand, users' quality of service expectations have also strengthened turning the Internet into a truly competitive and mature market.

On the other hand, the operators' answer has been more investments in terms of both CAPEX (CAPital EXpenditures) and OPEX (OPErational EXpenditures). That is, backbone links witness continuous upgrades, and probes have been deployed across operators' infrastructures to allow them to perform measurement campaigns and careful monitoring, which helps satisfy quality demands from users but also entails costly management.

Nonetheless, it is not only operators who face the challenge of handling and monitoring high-speed networks, but also other entities such as banking institutions, content providers, and networked application developers [LSBG13]. In these cases, the task may not be only to deliver bytes but also to dig into applications' behavior, achieve the best performance, or inspect what traffic looks like. As an example, in a bank network where security is a key element, network managers must collect and study humongous sets of data often with aggregated rates of several Gb/s to identify anomalous behavior and patterns. Indeed, things may be even worse as malicious traffic often shows bursty patterns

profiles [KKH+04]. The story is not very different for content providers and networked applications developers. They are certainly also interested in monitoring traffic and other forensic tasks, but additionally they have to assess that the performance of both their software components and their infrastructures scaling at least as fast as capacity and demands of Internet links do.

In short, the increase of the user's demands and subsequent link capacities have forced the different players in the Internet arena to deal with multi-Gb/s rates. To put this into perspective, we note that, for instance, packet-traffic monitoring at rates ranging from 100 Mb/s to 1 Gb/s was considered very challenging only a few years ago [MLCN05, PMAO04, FML+03], whereas contemporary commercial routers typically feature 10 Gb/s interfaces, reaching aggregated rates as high as 100 Tb/s [YZ10].

As a consequence, network operators have entrusted specialized hardware devices such as FPGA (Field Programmable Gate Array) [DKSL04, AGMM12], Ternary Content Addressable Memories (TCAMs) [YKL04, MPN+10], or high-end commercial solutions with their networked tasks [End14b]. These solutions give answer to high performance needs for a very specific task, e.g., lossless packet handling in a multi-Gb/s link while routing or classifying traffic [Sys13].

However, the initial investment is high and such specialization, as with any custom-made development, lacks both extensibility and flexibility, which in turn also have an indirect cost impact. As an example, in the case of large-scale networks featuring numerous Points of Presence (PoP), extensibility and ease of update are the key. Equivalently, in the case of a smaller network, it is desirable to have hardware flexible enough to carry out different tasks as specifications change. Additionally, in any scenario, network managers must prevent their infrastructure from being locked in to a particular vendor. As a workaround to these limitations, some initiatives have provided extra functionalities in network elements through supported API (Application Program Interface) that allow the extension of the software part of their products—e.g., OpenFlow [MAB+08].

It has been only recently that the research community has proposed, as a real alternative, the use of software-based solutions running on top of commodity general-purpose hardware to carry out network tasks [BDKC10, GDMR+13]. The key point is that this provides flexibility and extensibility at a low cost. Additionally, leveraging commodity hardware to develop networked services and applications brings other advantages. All the components a commodity system is based on are well known and popular hardware. This makes these systems both more robust, due to extensive validation, easy to replace, and cheaper, as the development cost per unit is lower thanks to the economies of scale of large-volume manufacturers.

Unfortunately, the use of commodity hardware in high-speed network tasks

is not trivial due to limitations on both hardware capacities and standard operating systems performance. Essentially, commodity hardware is limited in terms of memory and internal bus throughputs. From the software side, limitations come from a general-purpose network stack that overloads communications due to a prioritization of protocol and hardware compatibility over performance. To give some figures about the size of the challenge, in a fully-saturated 10 Gb/s link the time gap between consecutive packets assuming an Ethernet link and minimum IP (Internet Protocol) packet size below 68 ns, while an operating system may need more than half a microsecond to move each packet from kernel to application layer [RDC12]. Therefore, it becomes evident that it is harder to deal with small-sized packets than with large ones, as there is an inevitable per-packet cost. Unfortunately, small-sized packets traffic profiles are not uncommon on the Internet as for example VoIP (Voice over IP) traffic, distributed databases or even anomalous traffic [KWH06]. This calls for a careful tuning of both hardware and the operating system stack to improve the figures. Nonetheless, even in the best hardware and tuned operating system combination, there will be packet losses if the application layer is not aware of the lower levels' implementation. Furthermore, the optimization of a network I/O module must be done taking into account more things that performance alone, or issues such as packet reordering [WDC11] or timestamping [MSdRR+12] inaccuracy may arise.

The development of high-performance networked services and applications over commodity hardware should follow a four-layer model. The first layer comprises the NIC (Network Interface Card), that is, the hardware aimed at capturing the incoming packet stream. There are several major NIC vendors but it is Intel, and especially its 10 GbE model with chipset 82599 controller, that has received most attention from the community. This chipset provides performance at competitive prices, and more importantly, it offers novel characteristics that turn out to be fundamental in order to achieve multi-Gb/s rates at the application layer. The next layer includes the driver. There are of two kinds: standard or vanilla drivers, i.e. as provided by Intel; or user customized ones. The third layer moves packets from the kernel level to the application layer. This includes the standard way operating systems work, i.e. by means of a socket and network stack. In addition to this, there are different libraries that help application developers to interact with traffic by means of a framework. Among all these libraries, PCAP is considered the *de facto* standard [ALN12]. The combination of a driver and a framework is known as a packet capture engine and the literature gives several examples of high-quality engines using commodity hardware [GDMR+13]. These engines typically feature a new driver and performance-aware frameworks. Finally, we have the application layer, which encompasses any service or functionality built on top of a capture engine. As previously mentioned, examples can be found nowadays in software routers, firewalls, traffic classification, anomaly and intrusion detection systems, and many other monitoring applications [LLK14].

However, the increased demands for network processing capacity could be translated into a big number of machines even though if off-the-shelf systems were used. Such an amount of machines means high expenses in terms of power consumption and physical space. Moreover, the presence of commodity servers from different vendors empowers the appearance of interoperability issues. All those drawbacks damage the profitability that networked service providers may experience. This issues among others, have motivated an incremental trend in the recent years regarding the use of virtualization for computational purposes. Such trend has been empowered by the inherent advantages provided by virtualization solutions [YHB+11]. In this light, network operators and service providers have been working during the last years on the development of the concept of Network Function Virtualization (NFV). This new paradigm aims to unify the environments where network applications shall run by means of adding a virtualization layer on top of which network applications may run. This novel philosophy also allows merging independent network applications using unique hardware equipment. Consequently, the application on NFV can increase the benefits obtained by network service providers by (i) reducing their equipment investment by acquiring large-scale manufacturers' products and by reducing the amount of physical machines required, which also entails cuts in power consumption; (ii) speeding up network applications maturation cycle as all applications are developed in an unified environment; (iii) easing maintenance procedures and expenditures as testability is radically enhanced; (iv) opening the network applications' market for small companies and academia by minimizing risk and thus encouraging innovation; (v) the possibility to rapidly adjust network applications and resources based on specific clients requirements.

The development of this novel NFV philosophy has been favoured by other trending technologies such as Cloud Computing and Software Defined Networking (SDN). In the case of Cloud Computing, NFV can easily benefit from all the research carried out on virtualization management [AFG+10]. Furthermore, NFV is to reside inside Cloud providers' networks in order to carry out all the network-related management tasks. On the other hand, NFV is complementary to SDN but not dependent on it and vice-versa [MRF+13]. NFV enhances SDN as it provides the infrastructure on top of which SDN software can run. Nevertheless, in order to make the most of NFV, mechanisms that allow obtaining maximum network processing throughput in such virtual environments must be developed.

1.2 Objectives

The goal of this thesis is the creation and evaluation of a general-purpose network monitoring system, that users and practitioners in the field may use to develop their own high-performance network applications. Our proposal, HPCAP, has been designed after some preliminary experiences which provided valuable knowledge that have proven vital along the development of this work. Consequently, the partial goals established for this work are:

- Evaluating different **hardware architecture** possibilities for carrying network monitoring tasks. This involves experimentation with the diverse alternatives available, in order to comprehend the pros and cons offered by each of them.
- Supplying a wide background, knowledge, and both qualitative and quantitative reference that practitioners and researchers need to:
  - develop their own high-performance networked services and applications on commodity hardware starting from square one, or
  - to choose the most attractive option between the already existing ones to meet their specific requirements.

  This implies the development of a **sate-of-the-art** study on the existing solutions based on commodity hardware, which is the hardware architecture chosen for the main development. This study implies a deep understanding on the hardware and software elements involved in network monitoring tasks, as well as the main techniques documented to circumvent performance issues. Additionally, an extensive functional and performance evaluation of the available solutions must be carried out.
- Providing a new high-performance packet capture solution that includes some **important features** that other approaches have not taken into account. Those features are:
  - accurate packet timestamping,
  - line-rate packet storage, and
  - high-performance duplicated packets detection and removal.

  HPCAP is our proposal to meet this requirements, and a detailed description of this solution is provided so practitioners and researches can understand the points that differentiate HPCAP among other solutions. HPCAP, as well as each of its distinguishing features will be extensively tested and evaluated, both in terms of **performance** and **accuracy** when applicable. Furthermore, in order to favour the generality of our approach, the following features are also important to be addressed:

  - supporting the coexistence of network interfaces with dual functionality, i.e., one interface working in high-performance mode and the rest working as standard interfaces,
  - in order to favour porting to different vendors and drivers, stablish a clear hierarchy that eases the migration of HPCAP to new devices.

- Creating **M^{3}OMon** as a general-purpose programming **framework** built on top of HPCAP over which users can easily develop their own applications. This framework allows to instantiate user-level applications feeding from one or more **data granularity** levels, namely:
  - the network packets themselves,
  - flow records automatically generated by the framework from the incoming traffic, or
  - time-series containing the packets, bits and active flows for each second the system is running.

  This framework must be extensively tested, in order to establish a set of performance bounds to be taken into account by potential users in order to develop their applications over it.

Virtualization techniques are being applied to a diverse set of computing problems in diverse ways. Specifically, in the world of network traffic processing, virtualization has appeared as an attractive alternative towards the consolidation of an homogeneous hardware-independent environment. Nevertheless, the migration if high-performance network tasks to the virtual world implies optimizing not only the computational part of the task, but even more importantly the I/O. For this reason, this work also pretends to achieve the following goals:

- evaluating those virtualization techniques available for exporting physical I/O devices to a virtual environment both in terms of performance and functional implications, and
- establishing the feasibility of **migrating** the previously acquired knowledge and experience to an even more general scenario that includes the instantiation of network monitoring applications from **virtual** environments.

All the experiments described along this thesis work pretend to be repeatable and reproducible for any researcher interested. For this reason, all the code developments carried have been published following an Open-Source policy. In order to favour the widest accessibility for researchers and industry professionals to the systems and tools described, HPCAP, M^{3}OMon or HPCAPvf are available for download on GitHub.

1.3 Thesis structure

The rest of this thesis document is structured as follows:

- Firstly, Chapter 2 presents an overview with diverse architectures which could be applied to the high-performance network monitoring problem, outlining the pros and cons of each possibility.
- Chapter 3 focuses on the use of off-the-shelf systems for network processing. Furthermore, this chapter presents a survey of the current solutions with the different features and performance obtained by each of them. In order to ease the use of those solutions to any researcher or practitioner, a set of guided instructions and code examples is presented in Appendix A.
- After this state-of-the-art survey, Chapter 4 introduces the design goals and implementation of HPCAP, which is our proposal for the high-performance network monitoring problem. This chapter analyzes the main features offered by HPCAP both in terms of performance and accuracy, which are: packet timestamping, packet storage and duplicated packets removal.
- In Chapter 5, M^{3}OMon, a framework built on top of HPCAP is introduced. M^{3}OMon has been designed in order to make it as simple as possible for network applications developers to build and deploy new high-performance network applications. In conjunction with a detailed description of this framework, a complete and sound performance evaluation study has been carried out and presented, as well as a set o sample applications already built on top of it.
- In Chapter 6, an eye is turned to the Network Function Virtualization approach. This successful philosophy is widely extended nowadays, but no effort has been place yet in order to offer high-performance capabilities. This chapter studies all the possible techniques that could be applied to translate the performance obtained by physical machines into virtualized scenarios. In addition to a complete performance evaluation study, two concepts: Virtual Network Probe and Virtual Network Monitoring Agent are presented as our proposals for high-performance virtualized networking.
- Finally, Chapters 7 and 8 present the conclusions and fruits of this work, in english and spanish languages respectively.

2 Architectures for network monitoring

Users' demands have dramatically increased due to widespread availability of broadband access and new Internet avenues for accessing, sharing and working with information. In response, operators have upgraded their infrastructures to survive in a market as mature as the current Internet. This has meant that most network processing tasks (e.g., routing, anomaly detection, monitoring) must deal with challenging rates, challenges traditionally accomplished by specialized hardware components. This chapter gives a brief summary of diverse hardware approaches that have been used in the network processing field, emphasizing the pros and cons of each alternative.

Along the past decades, the evolution of society has been characterized by an increment in the usage of new technologies: users are more connected, in a more varied amount of ways, and through different services. In order to provide a proper and profitable service, operators and network services providers must deploy effective monitoring and quality-assurance policies. Nevertheless, the network processing tasks involved by these policies have to deal with three main challenges:

- **Huge amount of data:** there is a need to keep record of diverse data records (e.g., connection to servers, session-related data, network performance estimators, ...) for an humongous set of network users.
- **High data rates:** in order to give service to the current existing amount of users, ISP (Internet Service Provider)s' infrastructures is populated with high-speed networks and interconnection systems. Consequently, any network monitoring policy deployed on those infrastructures must be capable of acquiring the network data at high-speed rates, i.e., 10 Gb/s and above.

- **The speed of the data's validity:** the data that can be extracted from the network has to be processed as fast as possible, because those data are only valid if they are obtained and used before their value is expired.

In order to face those challenges, the network data extraction and processing must be carefully planned. Furthermore, the computational power required to carry out those tasks claims for very specialized or tuned computing architectures.

In this chapter, some architecture alternatives are analysed in terms of the computing capacities offered, their programmability, their ease of support and their interconnection mechanisms. Afterwards, some applications developed using this kind of equipments are presented, namely the `Argos` FPGA-based monitoring card, and `Twin`$^{-1}$ GPGU (General-Purpose Graphic Processing Unit)-based duplicate detection and load balancing system.

2.1 Hardware components

Along this section, the main features offered by different computing-capable hardware components is presented, with their main advantages a disadvantages highlighted, and summarized in Table 2.1.

	ASICs	**FPGAs**	**GPGUs**	**Commodity hardware**
Performance	Maximal	Very High	Very High	Good
Interconnection	Custom	Custom	Co-processor	Standard
Programmability Support	Low	Low/Medium	Medium/High	High
Extensibility	Very Low	Medium	High	Very High

Table 2.1: Pros and cons summary for different hardware computing alternatives

2.1.1 Application-Specific Integrated Circuits

ASIC (Application-Specific Integrated Circuit) are hardware elements that offer maximum potential of adaptability during their design phase. The computing architecture is completely opened, and designers are free to completely build it from scratch [KC00]. For example, an ASIC could be designed for optimizing decimal floating point operations [DBS06] if it is going to be applied in a high-frequency banking problem, modular-arithmetic blocks for cryptographic

processing [WOL02], to simultaneously access as many memory banks as required, or to obtain a minimum-overhead connection with an external component. As a consequence, ASIC-based computing always offer maximum computational potential.

In terms of interconnection between the data to be processed and the computing element, the flexibility offered by ASIC platforms allows the designer to choose the most suitable scheme: the ASIC could be a standalone processing unit, or it could be connected as co-processing unit in a bigger system. However, if data storage tasks (or other tasks requiring the connection with complex hardware elements) are required, developing custom connection will greatly increment the design's complexity and, probably, profitability.

On the other hand, all this performance potential is obtained at the expense of a very low degree of programmability. ASICs have to be programmed using a low-level hardware design techniques that, even though the existence of simulation techniques, imply the use of electronic design tools that must deal with physical-level details. This turns ASIC design into a task available to a small set of designed with very specific professional skills. Consequently, ASIC design has a slow and expensive life cycle.

In terms of expenses, creating an ASIC-based product is a very costly process, as it requires producing large sets of chips due to the way ASIC manufacturers function. Furthermore, once a product is commercialized, its extensibility and maintenance has similar problems. If an ASIC device is modified and need to be re-installed, this version changes implies starting the production and installation process from scratch.

All those characteristics that ASIC-based solution have, turn them into a very appealing option for solving very-specific and constrained problems. However, their application into a general range of network applications becomes unfeasible.

2.1.2 Field Programmable Gate Arrays

Field Programmable Gate Arrays (FPGAs) are a special kind of integrated circuits designed so they can be configured through a set of memory elements after the chip has been manufactured [HD10]. Once a specific FPGA chip is set, its programming can be modified from a HDL (Hardware Description Language) description through a set of synthesis, component mapping and routing processes. FPGAs allow digital designers to create almost any digital component combination, and as technology evolves they tend to incorporate optimized block to include the most common digital (and also analogical for some manu-

facturers) components. For this reason, they offer a trade-off between the rigid ASIC devices and a standard computing device such as a microprocessor. In terms of performance, a certain FPGA design might be below an ASIC equivalent only in terms of clock frequency supported, so their offered performance level is near to optimal.

The interconnection possibilities between the data and the FPGA as computing element are as wide as the amount of commercial products in the market: the FPGA could run as a standalone element, be used as a co-processor connected to a CPU (Central Processing Unit) socket [Xtr, DMLB+13], or be connected via a expansion slot as in [Com14]. As with ASICs, the more complex the systems to be directly connected to the FPGA are, the more complex the design and peripheral control process will become.

From the programmability point of view, FPGAs are typically programmed from a description created using HDL languages. That opened FPGAs to a larger set of professionals than ASICs, but the design's life cycle is still long and expensive. Nevertheless, in the last decade there has been a important increase in the use of HLL (High-Level Language)s [PT05, CCA+11] for FPGA programming. This novel possibility for FPGA, although still in development and with the need of support from the chip manufacturer, has turned the task of FPGA programming into a more productive and effective task [EAMEG11, EAMEG13]. However, FPGA designers still need to have a deep knowledge on the FPGA technology in order to obtain reasonable results.

On the other hand, although a single FPGA chip raises the final price compared to a more standard solution, their price is still lower than the one from a tailored ASIC. The benefits of buying the same chip for diverse purposes, allows to deploy effective replacement policies.

Consequently, FPGAs provide an attractive trade-off between performance and programmability. Nevertheless, the long time-to-market cycles associated to FPGA development, even though the existence of HLLs approaches, complicates their application applied in a generalistic way. This thesis has evaluated the development of a FPGA-based network monitoring system with an emphasis on having an accurate packet timestamping mechanism, which was distributed and integrated in a European-level testbed.

2.1.3 General-Purpose Graphical Processing Units

Graphic Processing Units have been used for years as co-processors with the aim of accelerating the most common computations required for graphical processing. Due to the success of this approach, the main manufacturers of

those devices decided to go one step further and generalize their usage, so their computing capabilities could be used by any generic applications [OHL+08, ND10a]. The traditionally language used for programming GPU devices, OpenCL, proved useful for this new approach, and even manufacturers such as nVidia created their own approach, CUDA (Compute Unified Device Architecture) [SE10].

The most extended usage of GPGU computing elements is by connecting them to a CPU via an PCI (Peripheral Component Interconnect) expansion port. When using a GPGU, users must be aware of the resource sharing constraints or implement some resource allocation layer as in [GLBS+12], although some manufacturers are evolving their tools to circumvent this problem. Nevertheless, there has been recent efforts made, in which electronic designers haver merged CPU and GPGU computing functionalities in a single chip [DAF11, CGS14], with the aim of generalizing and increasing the amount of work done by the GPGU co-processors. Importantly, GPGUs can not run as a standalone computing element.

GPGUs offer a massively parallel computing approach, with a high amount of simple processors or Texture Processors (TP) grouped in Streaming Multiprocessors (SM). They also have a characteristic memory hierarchy, with diverse memory devices with diverse features depending on the volatility of the data to be stored. Although the programming approach is simple and provides a high-level abstraction, users still need to be aware of the peculiarities of the underlying hardware in order to exploit the performance possibilities.

In economical terms, GPGUs offer a very attractive performance-power ratio, which is one of the reasons for which they are being introduced in contemporary supercomputing systems [FQKYS04]. Manufacturers stablish several ranges GPGUs differentiated based on their features, so users can adjust their device to their own budget. Through being standardized devices, availability and component replacement issues are mitigated.

The use of GPGUs has proven a highly efficient solution for computational tasks fitting their programming paradigm. However, in the network processing field, although possible, leaving the GPGU the task independently fetch each incoming packet, processing it and moving the result elsewhere would be highly inefficient, as GPUs benefit from processing huge amounts of regular data units at the same time. A workaround to this problem is using the GPGU as an auxiliary processing element, so the CPU fetches the incoming traffic and pre-processes it so that it has a GPU-friendly format, which is the approach that has been followed along the development of Twin^{-1} (see section 2.2.2).

2.1.4 Commodity hardware

The term commodity hardware was forged to refer to those hardware equipments which are easily and rapidly available for their purchase. That is, hardware that is easy to for being sold by ubiquitous retailers, or by a big manufacturer with a well developed delivery infrastructure. Note that the term affordable is usually used too when describing commodity hardware, but we deliberatively not include it in our definition, because it may create confusion, as commodity hardware systems may not be cheap in absolute terms although they are compared to their alternatives.

Chapters 3 and afterwards of this thesis dissertation focus on the usage and optimization of commodity hardware for network monitoring tasks, so no further discussion is included in the present chapter.

2.2 Background experience

This section presents `Argos` and `Twin`$^{-1}$, which are two real-application implementations based on FPGA and GPGU respectively. Both prototypes were developed during the architecture-evaluation phase of this thesis.

2.2.1 ARGOS

`Argos` is a FPGA-based network monitoring card which was developed using Stanford's NetFPGA 1G platform [Com14], and completely developed using HDLs. This platform allows the programming of a Xilinx Virtex-II Pro 50 FPGA device, which is connected to four Gigabit Ethernet ports, and to another FPGA whose program is fixed and manages a simple PCI interface between the Virtex device and the system's PCI port. The design goal was the instrumentation of the ETOMIC experimentation testbed [CFH+10] with accurately-synchronized network monitoring equipment, in order to allow the testbed's researched to obtain fine-grain network measurements. This action was carried out under the *European Union OneLab2 FP7 Project (No. 224263)*.

More specifically, the design had to be synchronized via GPS, no all the devices geographically distributed along the testbed were globally synchronized. As the goal of this time synchronization was the timestamping of the processed network traffic, a local timing correction algorithm was developed. In order to simplify the interaction between the GPS antenna module and the network device, a MicroBlaze soft-processor was instantiated, with a simple peripheral to receive the GPS signal and global time messages. That MicroBlaze run a time-correction algorithm in order to correct the board's internal oscillator. The corrected timestamp value was made accessible to other design modules via a simplified peripheral connected to the processor's bus. The modular design scheme of the prototype is shown in Fig. 2.1. Importantly, the accuracy obtained by this time correction mechanism was in the order of tens of nanoseconds.

Once the accurate time source was established, it had two main purposes. First, all the incoming packets are assigned a timestamp at the moment in which the first byte of data was received through the MAC chip. Note that this guarantees a timestamping accuracy in two ways: (i) the accurate time source, (ii) the timestamp is not affected by processing delay. After their reception, packets are stored in a dual-port memory following a single-producer single-consumer policy with a custom header containing, among other fields, their arrival timestamp. Once packets are left in the memory device, they stay there until a custom kernel driver carried out a data request transaction and exports the packets' data to the CPU via the PCI bus. This custom driver translates the custom header informa-

tion and lifts the data to the Linux's network stack so any upper-level application feeding from this NIC would have access to the accurately timestamped traffic.

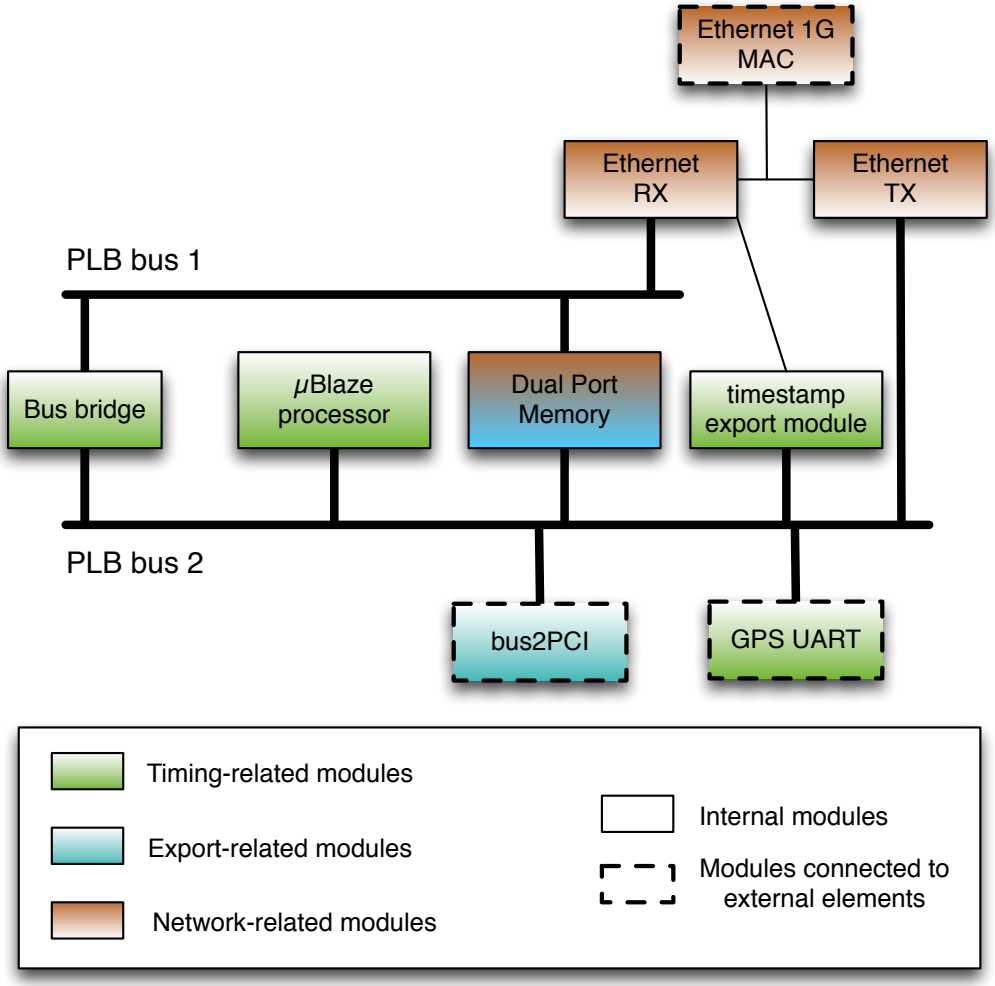


Figure 2.1: `Argos`' block scheme

Second, the timestamp value was also used to accurately control the transmission of traffic with a certain interarrival profile. Specifically, the `Argos` programming interface allowed creating a train of packets, which is composed by a configurable number of bursts of packets with a configurable amount of packets per batch, as in Fig. 2.2. More interestingly, the configuration allowed defining the interdeparture time of packets belonging to the same burst, and the time elapsed between the transmission of different bursts belonging to the same packet train (interburst time). The packets sent in a train are all UDP (User Datagram Protocol) packets sent with a configurable quadruple, so users may adapt them to their network configuration. Furthermore, inside each packet's UDP data field, the transmission module includes the timestamp in which the first byte of

the packet is transmitted through the network. Note that this implies an on-the-fly recalculation and modification of the Ethernet CRC value, but this can be easily done using a state machine inside the FPGA module.

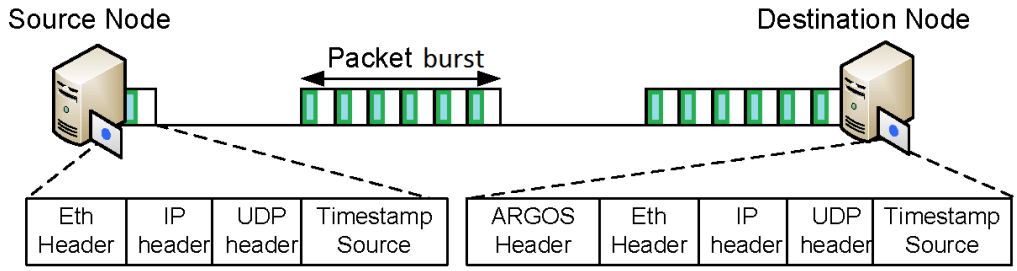


Figure 2.2: Packet burst train structure

Users may find this packet train feature useful by itself, but note that if the packets of a train sent with an `Argos` card are received by another `Argos` card, the receive side will have both the departure and arrival timestamps for each packet belonging to a train. This opens a venue for diverse network experimentation, in which final users may obtain very accurate information about their networks' behaviour. Fig. 2.3 shows an example of experiment, using two ETOMIC nodes one placed in Madrid and the other one in Budapest. The plot shows the one-way delay experienced by packets sent in each of the two possible directions. Results show latency differences between the two paths, that would be difficult to obtain with standard equipment.

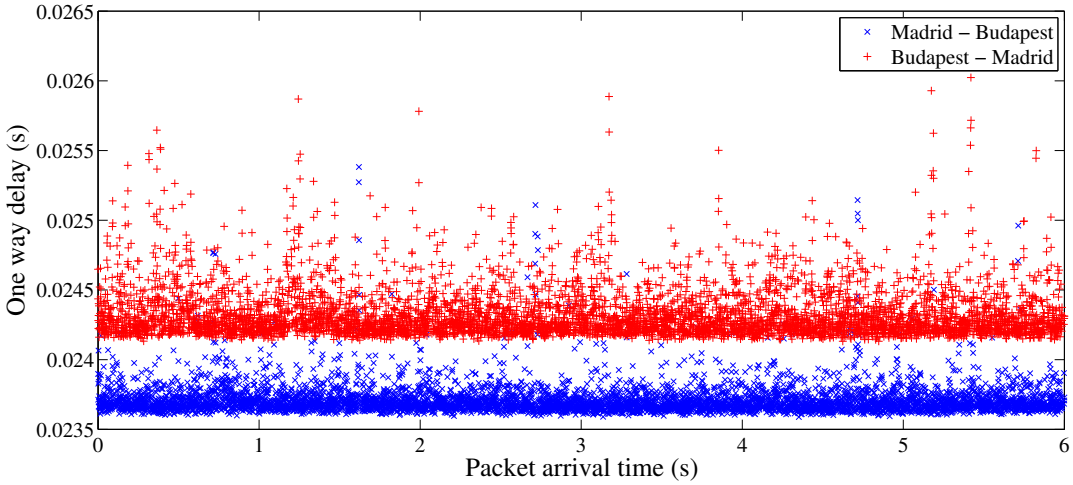


Figure 2.3: One-way delay measurements for a Madrid-Budapest connection

2.2.2 Twin^{-1}

Twin^{-1} was developed as a solution for a problem of high rates of packet duplication that a service provider was experience in its high-speed network infrastructure. The appearance of duplicated packets was a problem inherent to their network configuration [UMMI13] that could not be modified without damaging some important services. However, DPI analysis was required to be carried out over this network's traffic, and their analysis' was greatly disrupted due to the duplicity problem. Furthermore, the system was required to distribute the traffic between the servers carrying out the DPI analysis based on a set of IP and port numbering rules. The final architecture diagram of the Twin^{-1} system is showed in Fig. 2.4.

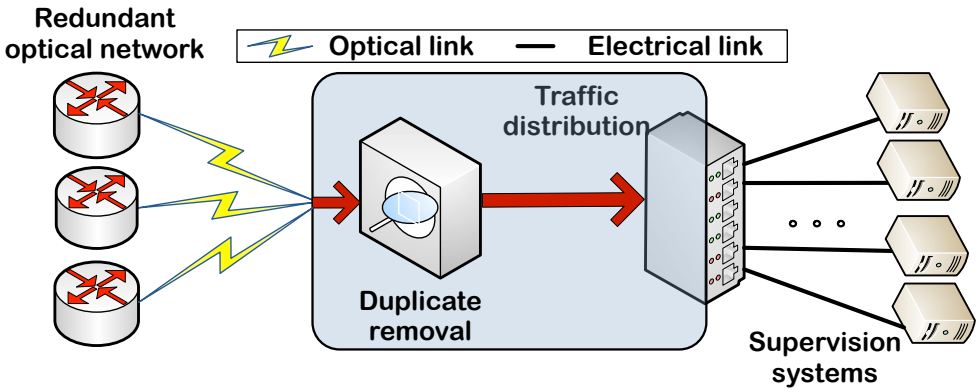


Figure 2.4: Twin^{-1}'s architecture

A deeper analysis over the network showed that the duplicated packets were near in time to the original packet, and a statistical analysis provided an estimator for the window size required for inspecting for possible duplicates, which was 600. Given this particular characteristic, it seemed rational to consider several hardware configurations for solving the problem. Specifically, a study taking into consideration solutions based on FPGA, GPGU and a multi-core processor were developed [MGAG+11]. The FPGA in use was a Altera's Stratix III inside an in-socket accelerator [Xtr], while the GPGU was a nVidia Tesla C1060, and the multi-core processor was Intel Quad-Core L5408. Importantly, the prototyping phase was carried out using HLLs for all hardware alternatives: Impulse-C for FPGA, CUDA for GPGU and OpenMP for multi-core. Although the use of a HLL for GPGU and multi-core programming does not make a difference, it has a remarkable impact on development time for FPGA programming.

The problem size made that the use of advanced data structures such as Hash tables had a negative impact on performance for the GPGU and multi-core

versions, as it generated computation imbalance and the overall throughput was limited by the slowest computing instance. However, the inherent nature of the FPGA makes it very favourable to the use of such structures, as the amount of accesses to memory elements is limited and access latency effects are minimized. Fig. 2.5 shows the throughput obtained for the duplicate removal part of the problem by each approach when working over real traffic obtained from the operator's network. It is worth remarking that both the GPGU and multi-core approaches suffered performance oscillations depending on the network traffic's characteristics while the FPGA doesn't, which is why an objective performance comparison could be done using real traffic.

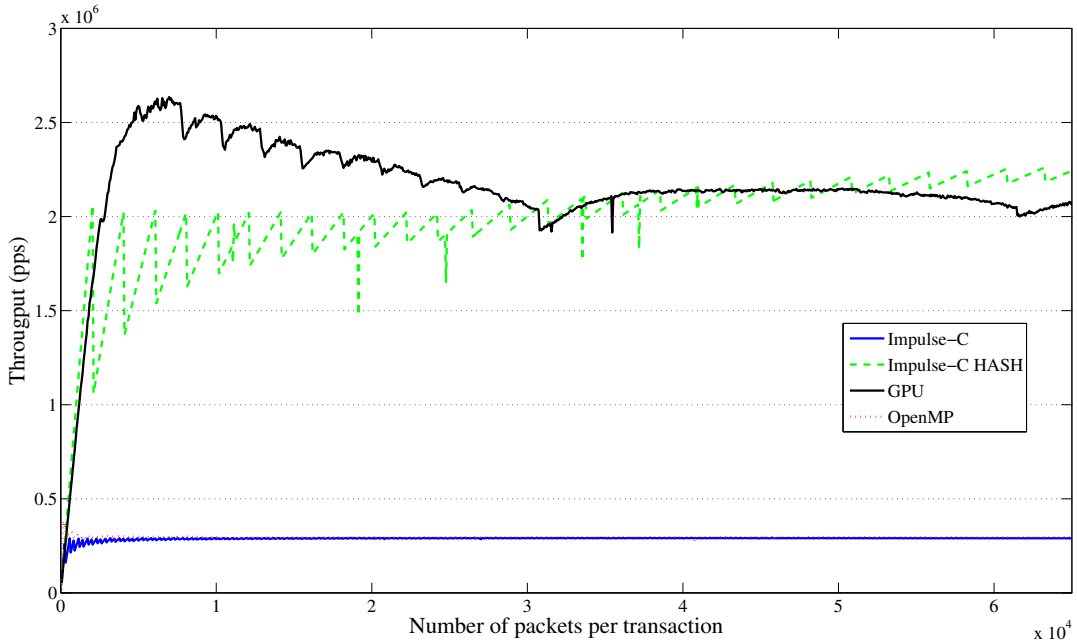


Figure 2.5: Throughput obtained by different hardware alternatives for duplicate removal over real traffic

After some practical considerations, the final version was designed choosing the GPGU hardware, as although it showed similar performance to the FPGA for the duplicate removal part of the problem, the maintainability, interconnection and packet shuffling problems were easier to address. Importantly, $\texttt{Twin}^{-1}$ was installed in the operator's core network in 2011, and has been working there since.

2.3 Conclusions

Along this chapter, several hardware alternatives for network processing tasks have ben presented, together with their majors pros and cons. Furthermore, two network processing example developments have been presented, each which have involved the use of different hardware devices for their solution. The experience acquired along those two projects have been useful in order to realize several important requirements when carrying out a general-purpose network monitoring task:

- **Timestamping accuracy:** having a as accurate as possible way of assigning each packet their arrival time is a fundamental requirement in order to carry out network performance and QoS (Quality of Service) analysis. This requirement becomes even more critical when moving towards higher speed networks, i.e., 10 Gb/s and beyond. Sources of inaccuracy are not only derived from the time source used, but also the moment in which packets are assigned their timestamp. A good-quality network processing solution should take this into account.
- **Network duplicates:** the appearance of duplicated packets in a core network is a real problem which, if not taken into consideration, may damage all analysis carried out over their traffic. Although there are hardware elements such GPGUs that have proven useful for solving this problem, the duplicated have to be pre-processed and transferred to those computing elements, with the subsequent resource consumption. An optimal network monitoring system should try to detect duplicates in a level as low as possible in order to avoid those undesired packets to prevent useful computations.
- **Network stack limitations:** the development of `Argos` implied the development of a Linux network driver that fetched the packets from the FPGA and transferred them into the operating system's network stack. Although the packet capture throughput sustained by the FPGA-to-driver connection was high, the throughput obtained when packets traversed the network stack was significantly damages. Consequently, a line-rate network fetching mechanism will have to circumvent the standard network stack.

All those lessons learnt, amongst others, motivated the development of HPCAP, which is detailed in Chapter 4. The hardware architecture chosen to carry out this development is commodity hardware. Commodity hardware offers a trade-off between the high-performance offered by other hardware approaches, while keeping a high degree of flexibility as our general-purpose goals required.

3

Packet capture using commodity hardware

The present evolution of the Internet, plus the appearance of a number of network services force network managers to operate with huge amount of data at challenging speeds. The high demands of network monitoring tasks (e.g., routing, anomaly detection, monitoring) required the use of specialized hardware for solving the problem. However, such approaches lack either flexibility or extensibility —or both. As an alternative, the research community has proposed the utilization of commodity hardware providing flexible and extensible cost-aware solutions, thus entailing lower operational and capital expenditure investments. In this scenario, we explain how the arrival of commodity packet engines has revolutionized the development of traffic processing tasks. Thanks to the optimization of both NIC drivers and standard network stacks and by exploiting concepts such as parallelism and memory affinity, impressive packet capture rates can be achieved in hardware valued at a few thousand dollars. This chapter explains the foundation of this new paradigm, i.e., the knowledge required to capture packets at multi-Gb/s rates on commodity hardware. Furthermore, we thoroughly explain and empirically compare current proposals, and importantly explain how apply such proposals with a number of code examples. Finally, we review successful use cases of applications developed over these novel engines.

In this chapter we present, in a tutorial-like fashion, all the knowledge required to build high-performance network services and applications over novel capture engines running on commodity hardware. Furthermore, we will show shortcuts to speed-up the development of such services and applications, by explaining the hardware and software keys to implement a custom packet capture engine, by detailing and illustrating with command prompt instructions how the capture engines proposed in the literature work, as well as by carrying out a performance comparison that allows users to select the capture engine best fitting

their needs. In addition to this, we present a state-of-the-art overview of current services and applications already benefiting from this new network paradigm.

The rest of this chapter is organized as follows. The next section explores the characteristics of the hardware referred to as commodity hardware as well as how current operating systems' network stacks work. Then Section 3.2 gives a strong background on the limitations of commodity hardware, necessary to understand the keys to overcome them. This knowledge would suffice for users to develop their own high-performance network drivers starting from vanilla ones. However, there is the option of using one of the high performance packet capture engines proposed in the literature. This is discussed in Section 3.3, thus enabling practitioners not interested in low-level details but in developing applications on commodity hardware, to skip much of the effort to master low-level interactions. Section 3.4 is devoted to evaluate the performance of capture engines. First, we explain how to evaluate capture engines and then we present a fair comparison between the engines in the literature in terms of packet losses and computational resource consumption to allow potential users to choose the most suitable engine for their ultimate aims. After all the theoretical knowledge has been introduced, Section A gives a cookbook on how to know the system's architecture, load a driver (customized or not), modify the driver's characteristics, optimize performance, essentially a practical tutorial on how to get started with packet capture engines. After that, in Section 3.5, we survey applications that have leveraged packet capture engines successfully since this novel paradigm emerged. This may awaken new ideas but the reader can also view this as the current state-of-the-art bounds to beat. Finally Section 3.6 provides a list of lessons learned and some conclusions.

We believe that all this background, know-how, comparisons and practical lessons are what practitioners and researchers need to develop high-performance network services and applications on commodity hardware. In addition, advanced readers may find enriching unknown details, benefit from the comparison of the different capture engines currently in the literature, which itself is of great interest, and also find the applications other researchers are developing.

3.1 Commodity Hardware

The computational power required to cope with network data is always increasing. Traditionally, when the requirements were tight an eye was turned to the use of ASIC designs, reprogrammable FPGAs or network processors. Those solution offer great computational power at the expense of high levels of specialization. Consequently, they only address the performance half of the problem

but they fail at solving the other half, which is the inexorably need to perform more diverse, sophisticated and flexible forms of analysis.

Commodity hardware systems are computers that combine hardware with a common instruction set and architecture (memory, I/O and expansion capabilities) and open-source software. Such computers contain industry-standard PCI slots that allow expansion and mechanical compatibility to provide a wide range of configurations at minimal cost. Such characteristics position commodity hardware as a strong option in terms of economies of scale with reduced manufacturing costs per unit. Moreover, the widespread use of commodity NICs and multi-core CPUs enables computers to capture and process network traffic at wire-speed reducing packet losses in 10 GbE networks [HJPM10].

Lately the number of CPU cores per processor has been increasing and nowadays quad-core processors are likely to be found in home computers and even eight-core processors in commodity servers. Along with this step-up, modern NICs have significantly evolved both in terms of hardware design and capture paradigms. An example of this evolution is the technology developed by Intel [Int12] and Microsoft [Mic] known as RSS (Receive Side Scaling). The main role of RSS is to distribute network traffic load among the different cores of a multi-core system and optimize cache utilization. Such a distribution overcomes the processing bottleneck produced by single-core approaches. RSS traffic distribution among different receive queues is achieved by using an indirection table and a hash value calculated over a configurable set of fields of received packets. Each receive queue must be bound to a different core in order to balance the load efficiently across the system resources.

As can be observed in Fig. 3.1, once the hash value has been calculated, its Least Significant Bits (LSB) are used as index for the indirection table. Based on the values contained in the indirection table the received data can be assigned to a specific processing core. The default hash function used by RSS is a Toeplitz hash. Algorithm 3.1 shows the pseudocode of such a function. The algorithm takes an array with the data to hash and a 40-byte bitmask called secret key (K) as input. This hash uses the IPv4/IPv6 source and destination addresses and protocol field; TCP (Transmission Control Protocol)/UDP source and destination ports; and, optionally, IPv6 extension headers. Applying the default secret key, the resulting hash assigns traffic to queues maintaining unidirectional flow-level coherency, that is, packets containing the same 5-tuple will be delivered to the same processing core. Changing the secret key enables different traffic distributions focusing on other features. For instance an approach for maintaining bidirectional flow-level (session-level) coherency is presented in [WP12].

In addition to RSS technology, extra features are present on modern NICs. For example, Intel 10 GbE cards allow the programming of advanced hardware filters to distribute traffic to different cores based on simple rules. This feature

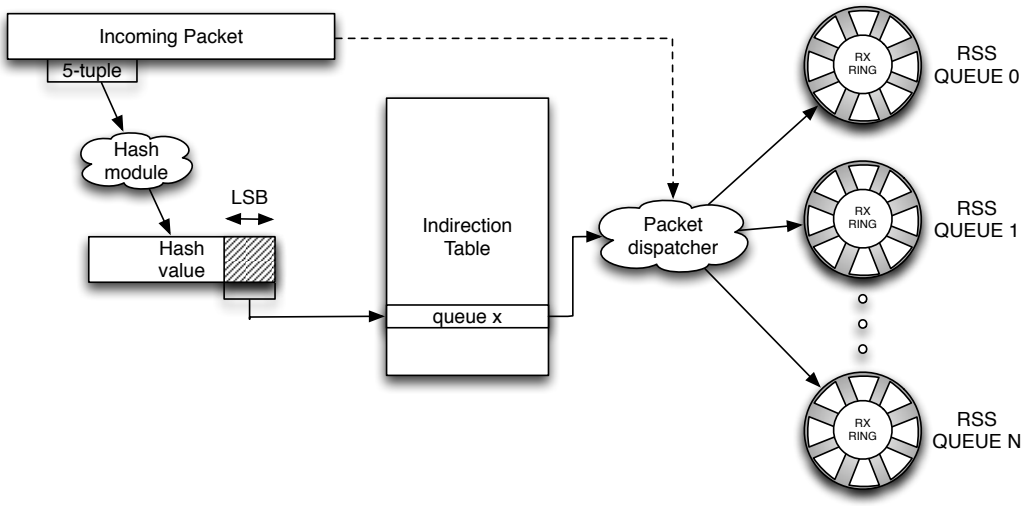


Figure 3.1: RSS architecture

```
input  : A 32-bit state key K and a bit stream input[]
output: A 32-bit result

result ← 0
for each bit b in input[] do
    /* bits are processed from left to right          */
    if b == 1 then
        result ← result xor Lowest32bits(K)
    end
    K ← K << 1
end
return result
```

Algorithm 3.1: Toeplitz hashing algorithm

is known as Flow Director and provides filters based on: source/destination addresses and ports; Type Of Service value from IP header; Level 3 and 4 protocols; VLAN ID and Ethertype.

3.1.1 NUMA architectures

Besides new hardware improvements, the interaction between the software and the hardware is an aspect of paramount importance in commodity hardware systems. For instance, NUMA (Non Uniform Memory Access) has become the *de facto* standard for multiprocessor architectures and has been exploited for high-speed traffic capture and processing. Briefly, NUMA architecture divides all available system memory into chunks and assigns each chunk to a different SMP (Symmetric Multi Processor). The combination of a processor and a memory chunk is known as a NUMA node. Some topology examples of NUMA architectures are shown in Fig. 3.2.

In NUMA architectures, each processor may access its own chunk of memory in parallel, boosting system performance and reducing the CPU data starvation problem. Notwithstanding that NUMA architecture increases performance in terms of both memory accesses and cache misses [DAR12], a careful process placement must be performed in order to avoid accessing memory located on another NUMA node.

Basically, access between a processing core and its corresponding memory chunk reduces the data fetching latency while accessing memory chunks located on another NUMA node increases it. To get the most out of NUMA architectures the distribution topology of NUMA nodes must be known in advance since it may vary depending on the hardware platform. To obtain the NUMA node distance matrix the `numactl`[1] command can be used. This matrix describes the distance from each NUMA node memory chunk to the others. As expected, the lower the distance the lower the access latency to other NUMA nodes.

Another aspect of paramount importance is the location and interconnection of devices. Typically in commodity hardware network capture systems, NICs make use of PCIe (Peripheral Component Interconnect Express) buses to connect to the processors. Connection may vary depending on the motherboard used in the commodity hardware capture system. Fig. 3.2 shows the most common interconnection patterns on current motherboards. In more detail, Fig. 3.2(a) shows an asymmetric architecture with all PCIe lines directly connected to a processor in contrast to Fig. 3.2(b) showing a symmetric scheme with PCIe lines distributed between two processors.

[1] `linux.die.net/man/8/numactl`

Figs. 3.3(c) and 3.3(d) show two topologies on which IO-hubs are used. The main difference between them is the existence of PCIe lines connected to one or more IO-hubs. Such IO-hubs interconnect PCIe buses as well as USB, standard PCI buses and other devices. Due to this architecture the bus between the IO-hub and the processor is shared with the subsequent performance problems. All these architectural issues must be considered when building a capture system. For instance, if a NIC is attached to a PCIe slot assigned to a NUMA node, all capturing threads should be executed on the corresponding cores of the NUMA node. If this is not done, data transmission between processors using the Processor Interconnection Bus may occur, degrading system performance.

To obtain the processor to which a PCIe device is assigned the following command may be executed on Linux systems: `cat /sys/bus/pci/devices/PCI_ID/local_cpulist`. Note that PCI_ID is the device identifier obtained by executing the `lspci`[2] command.

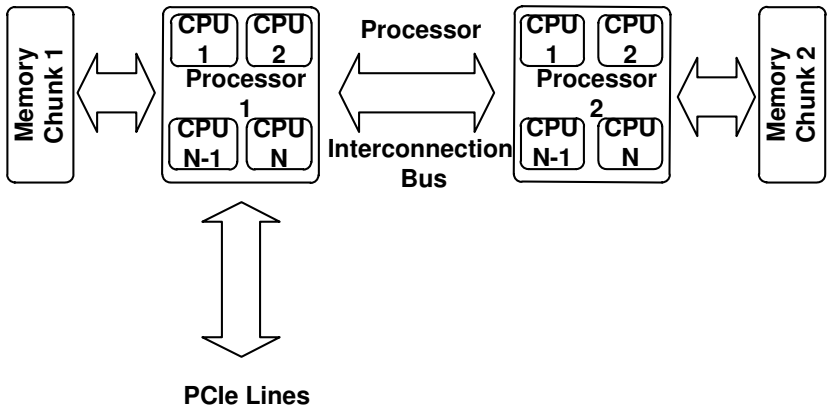
For all the above reasons, modern commodity systems are highly attractive to accomplish the demanding task of network traffic monitoring at high speeds as their performance is on par with today's specialized hardware, such as network processors [Luc14,LSI14,Int14c], FPGAs [Com14], Endace DAG cards [End14a] or commercial solutions provided by router vendors [Sys14] while keeping down the expense. Furthermore, due to the development of monitoring functionality at user level, commodity hardware-based appliances are flexible as well as scalable and extensible due to their inherent mechanical compatibility.

3.1.2 Current and past operating system network stacks

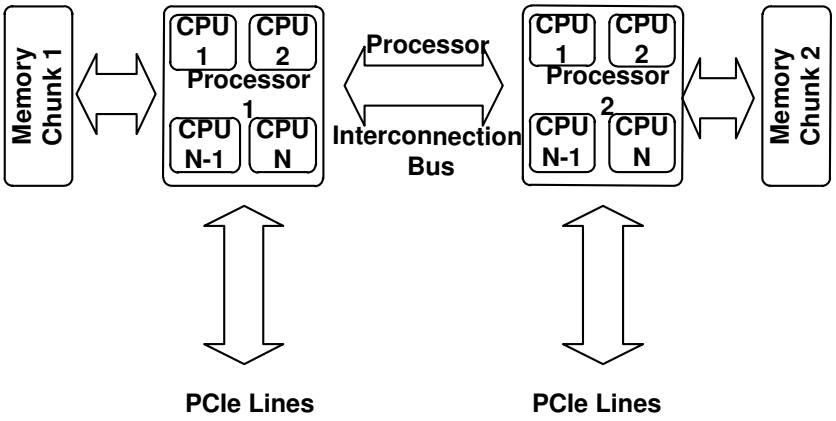
While network hardware has been rapidly evolving focusing on high-speed packet capture, software has not followed the same trend. In the case of software, modern operating systems are nowadays designed to provide compatibility rather than performance. Such operating systems present a general-purpose network stack that provides a simple socket user-level interface for data exchange and handles different hardware and network protocols. Nevertheless, such an interface is not optimal in terms of high-speed traffic capture.

Particularly, Linux kernels prior to 2.6 presented an interrupt-driven approach in the network stack. Focusing on behavior: whenever a packet arrives at the corresponding NIC, a descriptor in a NIC's receive (RX) queue is allocated and assigned to that packet. These queues are also known as rings due to their circular topology. Each packet descriptor contains a pointer to the memory re-

[2] `linux.die.net/man/8/lspci`

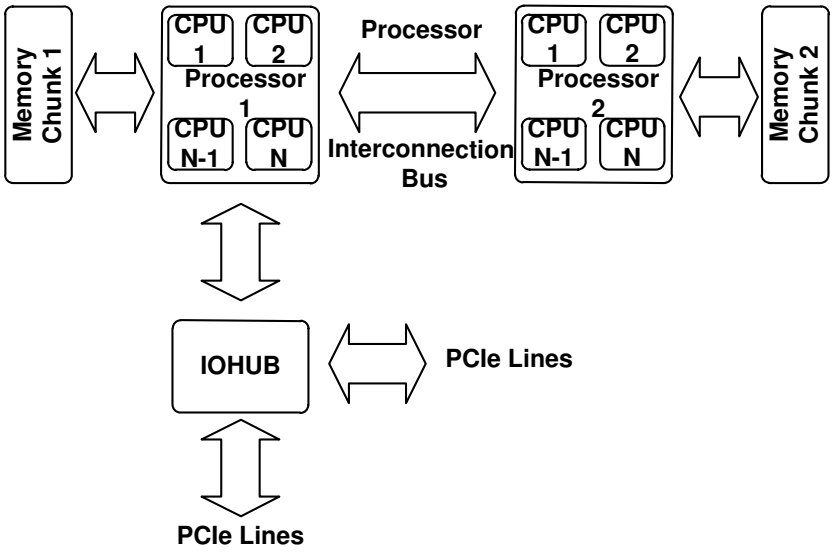


(a) PCIe lines connected to one processor

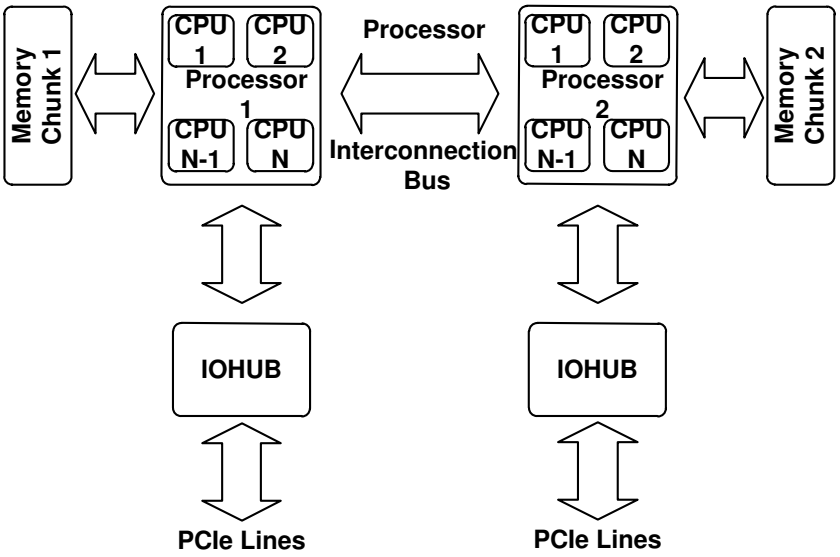


(b) PCIe lines connected to two processors

Figure 3.2: NUMA architectures topology examples



(c) PCIe lines connected to one IOHUB



(d) PCIe lines connected to two IOHUBs

Figure 3.2: NUMA architectures topology examples

gion needed to perform the incoming packet transfer via DMA (Direct Memory Access). In the case of packet transmission, DMA transfers are performed in the opposite direction and an interrupt line is raised upon completion to allow the transmission of new packets. This mechanism is common to all the different packet I/O commodity hardware solutions.

Fig. 3.3 shows the traditional Linux network stack behavior. When an incoming packet DMA transfer from the NIC to the host's memory (DMA-able memory region) is finished, an interrupt is signaled. Then, the software interrupt routine copies the packet's data into a local kernel `sk_buff` structure. This structure is typically called a kernel packet buffer. Once the packet has been copied, the packet descriptor is released so the NIC can reuse it to receive new packets. The `sk_buff` structure with the received packet data traverses the system's network stack until delivered to a user application. Following this I/O scheme, an interrupt must be raised each time a packet is received or transferred. This mechanism overcrowds the host system when the network load is high [ZFP12].

To avoid such behavior, current network drivers implement the NAPI (New API) approach [Lin14] to increase performance. NAPI was included in Linux kernel 2.6 to boost packet processing in high-speed scenarios. To speed up the packet capture, NAPI is based on two ideas:

1. **Interrupt mitigation:** following the traditional receive scheme many interrupts per second are generated when high-speed traffic is present. Handling such interrupts requires processor time and in the presence of a high interrupt rate, processors may be overloaded and performance degradation will ensue. To solve this issue, the NAPI-aware driver interrupt routine is launched when RX/TX interrupts arrive. Unlike the traditional approach, the interrupt routine schedules the execution of a `poll()` function and disables interrupts instead of copying and queuing the packet. The `poll()` function checks if new packets are received, and copies and queues them into the network stack when available in an interrupt-less way. The function reschedules itself to be executed in the near future (without waiting for interruptions). If no packets are available in this time period, packet interrupts are activated again. Note that polling mode demands more CPU time than interrupt-driven mode when network load is low but it becomes worthwhile as speed grows. Depending on the network load, NAPI compliant drivers adapt themselves to increase the performance as shown in Fig. 3.4.

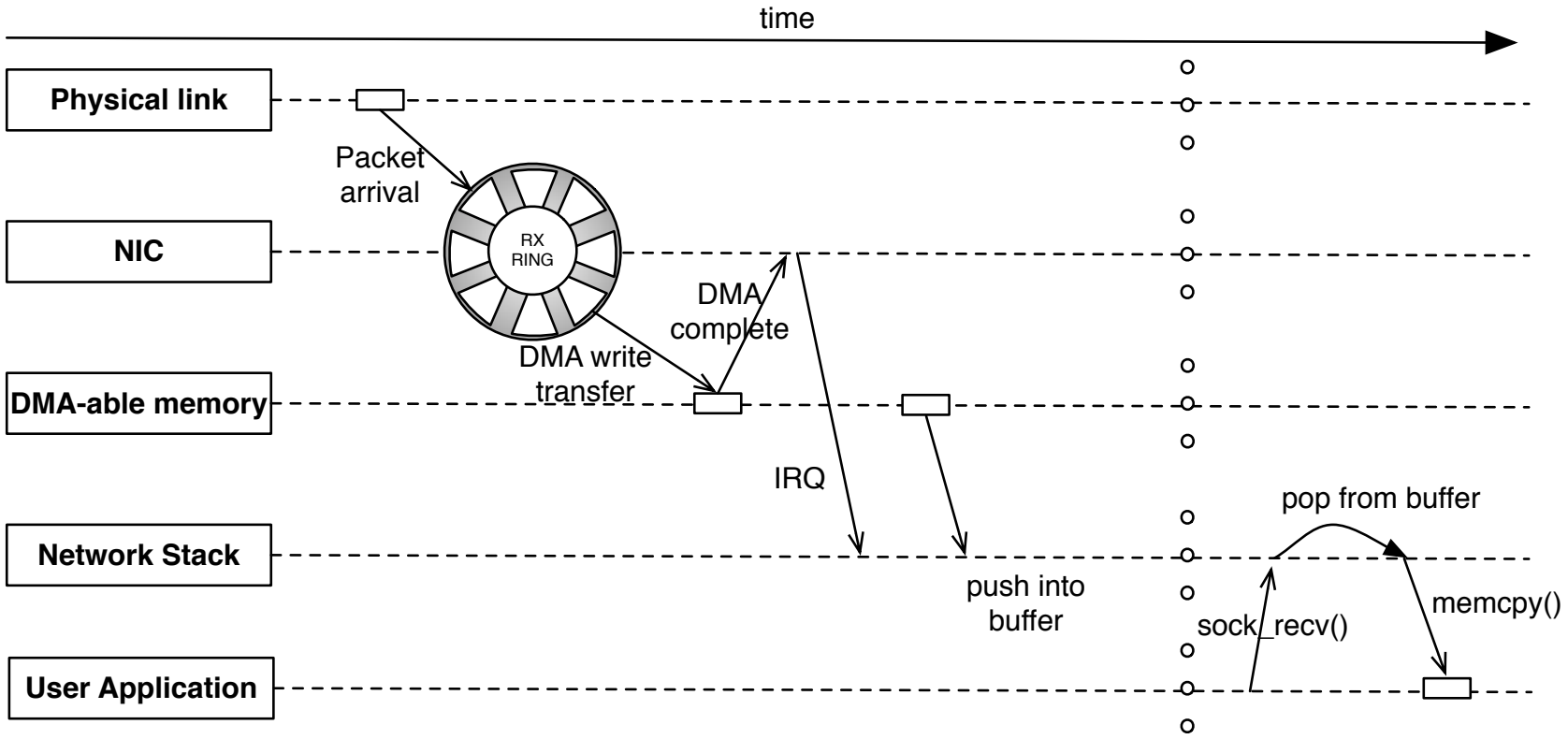


Figure 3.3: Linux Network Stack RX scheme in kernels previous to 2.6

2. **Packet throttling:** Traditionally, when high-speed traffic surpassed system capacity, packets were dropped at kernel-level rendering the previous communication and copying between drivers and kernel useless. NAPI compliant drivers drop traffic at network adapter level using flow control mechanisms thus preventing unnecessary work.

In what follows, the GNU Linux NAPI mechanism will be used to illustrate performance problems and limitations. This choice has been made as Linux is a widely used open-source operating system that allows full code modification for instrumentation and performance analysis purposes. Although the vast majority of the proposals in the literature have been developed for different flavors of the GNU Linux distribution, some of them are also available for other operating systems such as FreeBSD [Riz12a].

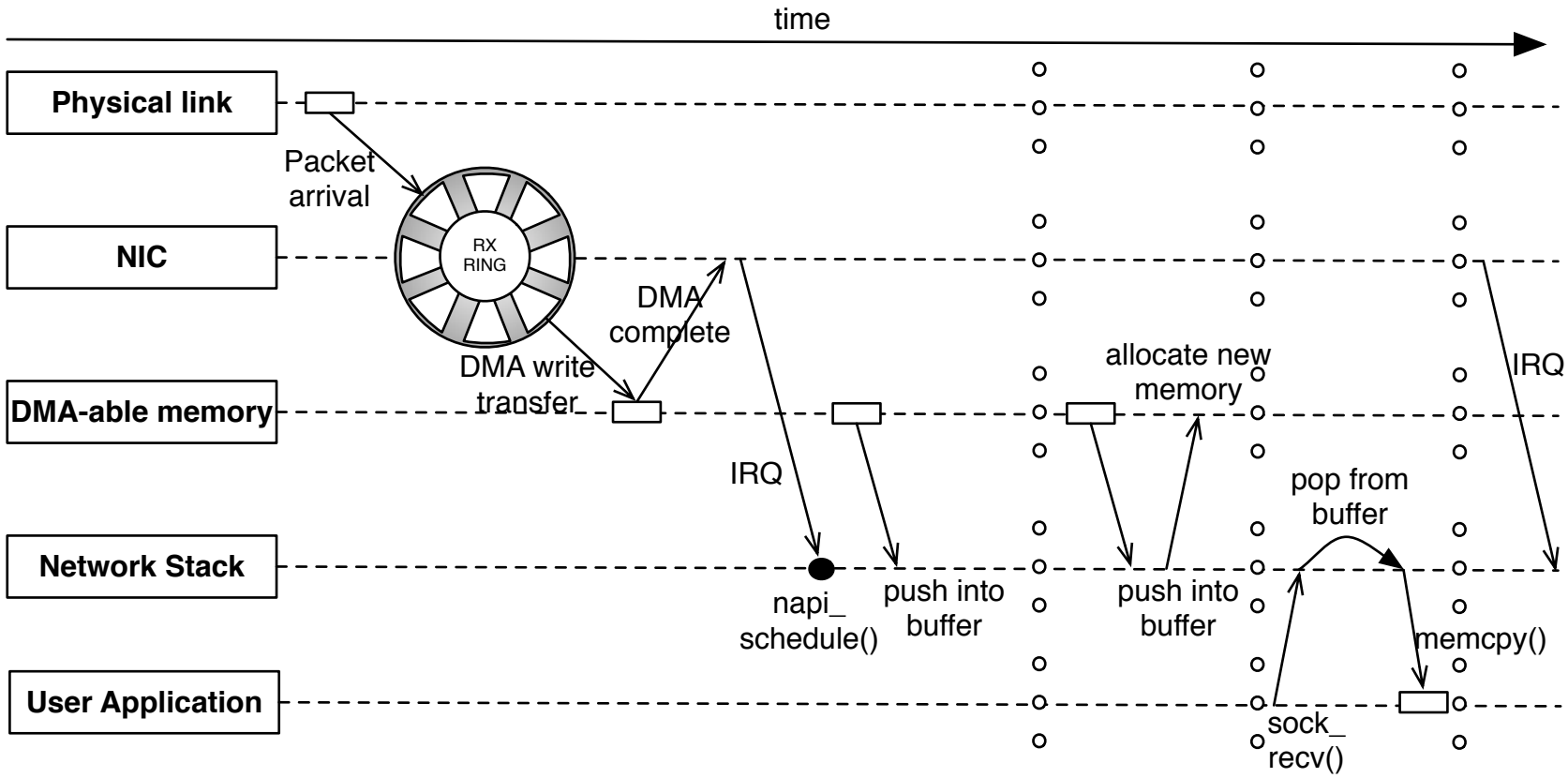


Figure 3.4: Linux NAPI RX scheme

3.2 Packet capturing

3.2.1 Limitations: wasting the potential performance

Although the way operating systems' network stacks has evolved, their robustness and flexibility remains a burden in terms of packet processing rates. The NAPI technique by itself is not enough to overcome the challenging task of very high-speed traffic capturing since other inherent architectural problems degrade the performance. After extensive code analysis and performance tests, several problems have been identified [HJPM10, Riz12a, LXB11, PVA+12]:

1. **Per-packet allocation and deallocation of resources:** Whenever a new packet arrives at a NIC, an `sk_buff` data structure is allocated by the kernel to store the packet's information. Once the packet has been delivered to user-level, its descriptor is released. This resource allocation and deallocation process generates a significant overhead, especially when receiving at high packet rates —as high as 14.88 million packets per second (Mp/s) in 10 GbE. Additionally, this `sk_buff` data structure is large because it may comprise information from many protocols on multiple layers, but most of this information is not necessary for numerous networking tasks. Modern drivers tend to group this structure allocation requests as a workaround in order to reduce the impact on performance of this process. As shown in [LXB11], `sk_buff` conversion and allocation consume nearly 1200 CPU cycles per packet, while buffer release needs 1100 cycles. Indeed, `sk_buff`-related operations consume 63% of the CPU usage in the reception process of a single 64-byte sized packet [HJPM10].

2. **Serialized access to traffic:** Modern NICs include multiple hardware RSS queues which are intended to distribute the incoming traffic using a hash function whose value is calculated at hardware-level based on the incoming packet's 5-tuple (Section 3.1). By exploiting this feature the capture process can be parallelized, since different NAPI threads could be bound to different CPU cores so each thread gathers the packets from a specific RSS queue. However, once packets are fetched, the GNU Linux network stack merges all packets at a single point on network and transport layers for analysis. Fig. 3.5 shows the architecture of the standard GNU Linux network stack. As a result, two problems arise from the use of this philosophy: first, all traffic is merged in a single point, which creates a processing bottleneck thus limiting overall throughput; second, user processes are capable of receiving the traffic from a certain RSS queue. Consequently, we cannot make the most of parallel capabilities of modern NICs delivered to a

specific queue associated with a socket descriptor. This serialization process degrades the system's performance regardless of any optimizations a particular network driver might implement. Furthermore, merging traffic from different receive queues may entail packet disordering [WDC11] and affect upper-layer packet processing policies.

3. **Multiple data copies from driver to user-level:** Packets are transferred to system memory through a DMA transaction. Until those packets are received from a user-level application they are copied several times, at least twice: from the DMA-able memory region in the driver to a packet buffer `sk_buff` structure at kernel-level, and from the kernel packet buffer to the user level. Each additional copy will obviously damage overall performance: a single data copy consumes between 500 and 2000 cycles depending on the packet length [LXB11]. Another important idea related to data copying is the fact that copying data packet-by-packet is not efficient, and deteriorates when packets are small. This is caused by the constant overhead inserted in each copy operation, which favors large data copies.

   Modern drivers reduce in one the amount of data copies required by reusing the same memory area containing the packet along the multiple layers traversed. However, this policy has collateral effects: buffers can not be released until all the upper layers are finished with them, and thus the driver must allocate new buffers or wait until some become available, which may turn into performance losses. Note that, by applying this policy, modern drivers do not need copying the data from the DMA buffer to kernel memory, but one more copy is still needed when transferring the packet's data (o a subset of its original data) to user-space applications.

4. **Kernel-to-userspace context switching:** Every time a user-level network application is to receive one packet, a system call must be performed. Each of these system calls will entail a user-level to kernel-level context switch and vice versa, with the consequent CPU time consumption. Such system calls and context switches may consume up to 1000 CPU cycles per packet [LXB11].

5. **No exploitation of memory locality:** The first access to a recently written DMA-able memory region entails cache misses, as DMA transactions invalidate cache lines. Such cache misses represent 13.8% out of the total CPU cycles consumed in the reception of a single 64 byte packet [HJPM10]. Additionally, in a NUMA-based system the latency of memory access depends on the memory node accessed. Thus, inefficient memory placement may entail performance degradation due to greater memory access latencies each time a cache miss is triggered.

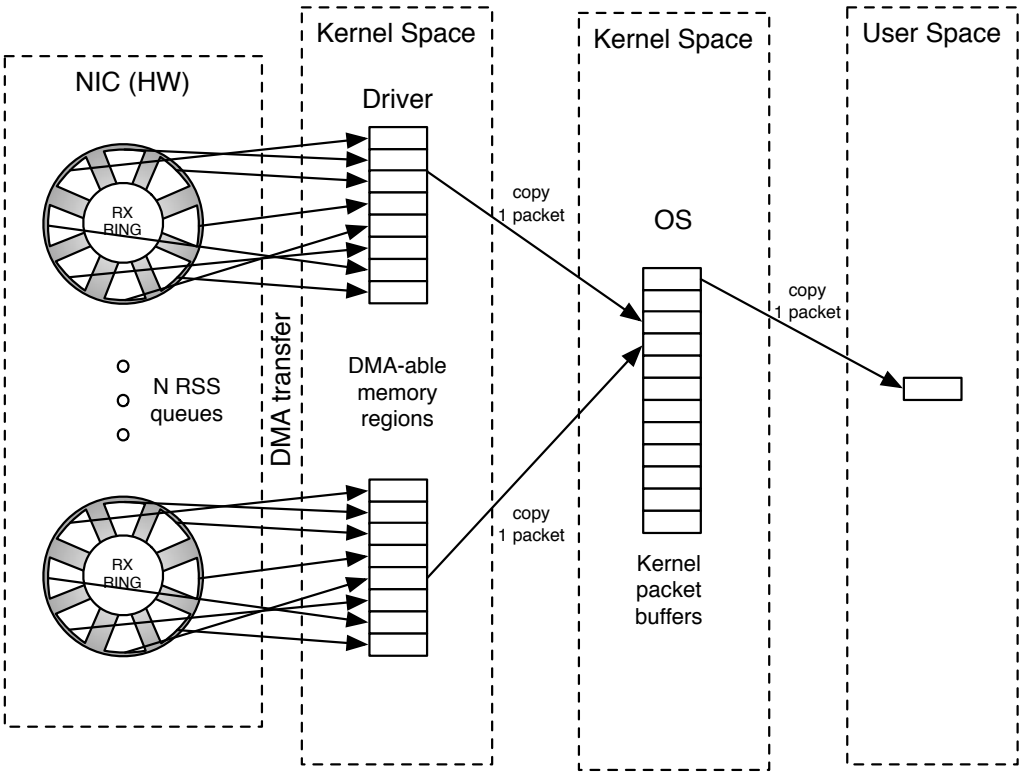


Figure 3.5: Legacy Linux Network Stack (serialized paths)

3.2.2 How to overcome limitations

In the previous sections, we have shown that modern NICs are a great alternative to specialized hardware for network traffic processing tasks at high speed. However, both the networking stack of current operating systems and applications at user-level do not properly exploit their new features. Here we present several proposed techniques to overcome limitations described above in default operating system network stacks.

Such techniques may be applied either at driver level, kernel level or between kernel level and user level, and are specifically applied to the data they exchange, as will be explained below.

1. **Pre-allocation and re-use of memory resources:** This technique consists in allocating all memory resources required to store incoming packets, i.e., data and metadata (packet descriptors), before starting the packet reception process. Specifically, N rings of descriptors (one for each rdware RSS queue on each device) are allocated when the network driver is loaded. Note that this means that the driver loading process will take some extra time, but once the reception process begins the per-packet allocation overhead is suppressed. Likewise, once each packet has been processed and transferred to userspace, its corresponding data structure will not be released, but it will be marked as available so it can be re-used to store a new incoming packet. This policy eradicates the bottleneck produced by per-packet allocation/deallocation. Additionally, the `sk_buff` data structures in use may be simplified to reduce memory requirements. These techniques must be applied at driver level.

2. **Exploiting queue parallelism:** This technique pretends to solve serialization in the access to traffic, by creating direct parallel paths between the RSS queues and the network applications as shown in Fig. 3.6. In order to achieve the best performance, specific and independent cores must be assigned for taking packets from each RSS queue and forwarding them to the user level. This technique supports the creation of new new parallel paths as the number of cores and RSS queues grow, which is an advantage in terms of scalability. In order to obtain such parallel direct paths, we have to modify the data exchange mechanism between kernel and user levels.

   On the downside, the use of this technique has two main limitations: First, each parallel path will make use of a CPU core, which reduces the number of cores available for different tasks. Second, RSS distributes traffic to each receive queue by means of a hash function. If our process does not analyze packet interaction, we can maximize parallelism by creating

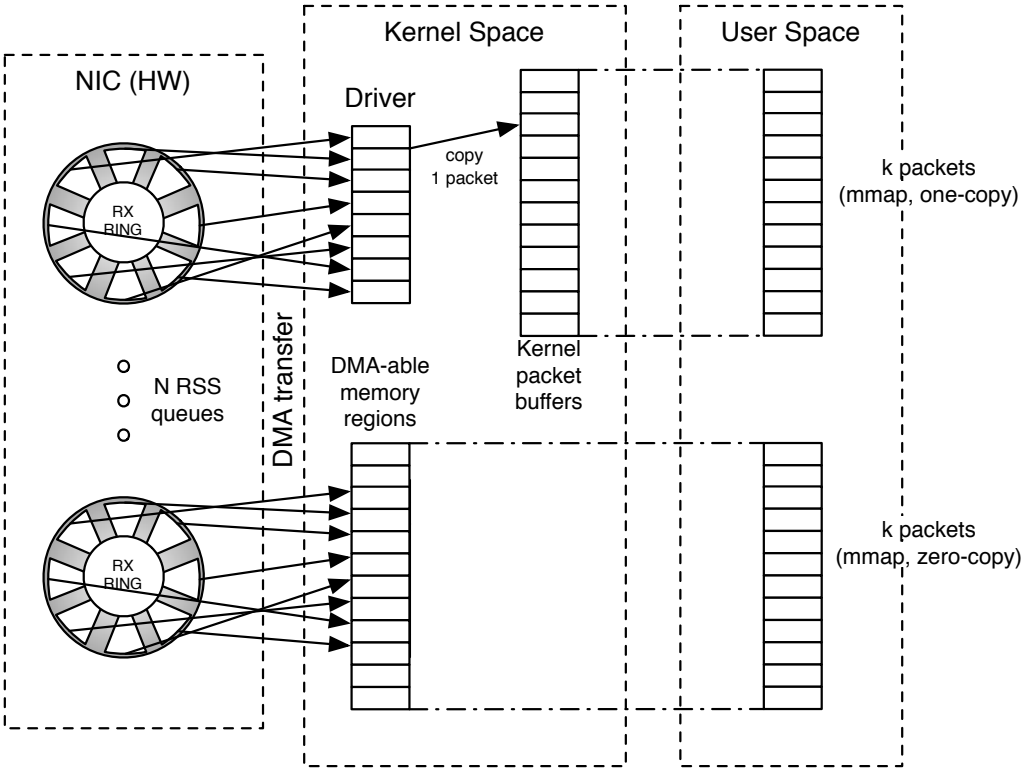


Figure 3.6: Optimized Linux Network Stack (independant parallel paths)

and linking one or more instances of this process to each capture core. However, if our networking tasks require the analysis of related packets, flows or sessions, it will need to fetch packets from different queues. For example, a VoIP monitoring system, assuming that such a system is based on the SIP (Session Initiation Protocol) protocol, needs to monitor not only the signaling traffic (i.e., SIP packets) but also calls themselves —typically, RTP (Real-time Transport Protocol) traffic. Obviously, SIP and RTP flows may not share either level 3 or 4 header fields that the hash function uses to distribute packets to each queue, hence they might be assigned to different queues and cores. The approach to circumvent this latter problem is that the capture system performs itself some aggregation tasks. The idea is that before packets are forwarded to userspace (for example to a socket queue), a block of the capture system aggregates the traffic according to a given metric. However, this is of course at the expense of performance.

3. **Memory mapping:** This feature supported by Linux's memory management model allows user-level applications to map kernel memory regions. Thus, applications are capable of directly reading and writing those memory areas without intermediate copies. This technique can be used to map from user-space those memory areas containing the data from the incoming packets and thus saving one kernel-to-user copy operation. Note that, if the memory areas mapped at user level are not those DMA-able regions where the NIC copies the packet data into, a copy is sill required so will refer to this configuration as *one-copy*. This approach is implemented on current GNU Linux as a standard raw socket when opened with the `RX_RING/TX_RING` socket options. Conversely, if the memory areas mapped by user applications are the DMA-able regions, no data copies are needed to access the packets' data, and thus the term *zero-copy* is used. As an inconvenient, a *zero-copy* driver can not re-use the DMA-able buffers until user applications are done with them. Additionally, exposing NIC rings and register memory areas to user-level access may entail risks for the system's stability [Riz12a], which must be properly handled. However, this is considered a minor issue as the APIs provided typically protect the critical regions from incorrect access. In fact, graphic cards make use of memory mapping techniques without major concerns.

   Fig. 3.6 illustrates two different approaches in which memory mapping techniques is used to achieve packet reception with *one-copy* or *zero-copy*. Applying these methods requires either driver-level or kernel-level modifications as well as in the data exchange mechanism between kernel and user levels. The use of memory mapping techniques to share data between kernel and user spaces allows reducing the amount of context switches in the packet capture process and thus improve overall performance. That is the case of modern versions of the libpcap library.

4. **Batch processing:** This technique is based on processing several packets at the same time, in order to reduce the overhead of per-packet operations. Packets are grouped into a buffer and copied to the target memory region in groups called batches. This technique reduces the number of system calls made by network applications, with their related context switches. This minimizes the overhead of processing and copying packets individually. In the NAPI architecture, there are two points where batches can be intuitively used. First, if packets are fetched via polling requests, more than one packet can be processed per poll request. Alternatively, if the packet fetcher works on an interrupt-driven basis, an intermediate buffer can be used to collect traffic until upper layers ask for it. However, the use of batching techniques may entail issues such as an increase in latency and jitter, and timestamp inaccuracy on received packets because packets have to wait until a batch is full or a timer expires [MSdRR+12]. In order to implement batch processing, we must modify the data exchange between kernel and user levels.

5. **Byte-stream oriented**: One step further than packet capture lies packet storage in non-volatile devices. In order to accomplish such task, a packet-by-packet policy issues many write operations and may not perform optimally. To mitigate this effect, some packet capture engines offer access to a byte-stream for user-level applications so they can work in terms of big blocks of bytes.

6. **Affinity issues:** In NUMA architectures, in order to increase performance and exploit memory locality, processes must allocate their memory in such a way that it is assigned to the processor (or NUMA node) in which it is being executed. This is known as memory affinity, but CPU and interrupt affinities must also be considered by software designers. CPU affinity allows control of the processors and cores where a given process (process affinity) or thread (thread affinity) is to be executed. Process affinity may be performed using Linux `taskset`[3] utility, and the thread affinity can be managed by means of the `pthread_setaffinity_np`[4] function inside the POSIX (Portable Operating System Interface) pthread library. On the other hand, software and hardware interrupts can also be bound for handling by specific cores or processors using a similar approach, whis is referred as interrupt affinity. This can be done by writing a binary mask to the file `/proc/irq/IRQ#/smp_ affinity`. The importance of setting capture threads and interrupts to the same core lies in the exploitation of cached data and load distribution. Whenever a thread accesses the incoming packets, finding them in the local cache will be more likely if they have been received by an interrupt handler assigned to the same core.

[3] `linux.die.net/man/1/taskset`
[4] `linux.die.net/man/3/pthread_setaffinity_np`

Another affinity issue that must be taken into account is to map the capture threads to the NUMA node attached to the PCIe slot the NIC has been plugged into. This PCI affinity allows maximum throughput to be obtained in DMA transfer operations. To accomplish this, the system information provided by the `sysctl` interface (shown in Section 3.1) may be useful.

7. **Prefetching:** Additionally, in order to eliminate inherent cache misses, the driver may prefetch the next packet (both packet data and packet descriptor) while the current packet is being processed. The idea behind prefetching is to load the memory locations that will be potentially used in the near future in the processor cache in order to access them faster when required. Some drivers, such as Intel's `ixgbe`, apply several prefetching strategies to improve performance. Thus, any capture engine making use of such a vanilla driver, will see its performance benefit from the use of prefetching. Further studies such as [HJPM10,SZTG12] have shown that more aggressive prefetching and caching strategies may boost network throughput performance.

8. **Capture and process isolation**: although we restrict this tutorial to the packet capture process, it is important to remark that any network processing task carried out on top of any of the explained packet capture engines will likely require additional per-packet computation, e.g., packet filtering, protocol classification, flow record extraction, ... Importantly, depending on the packet capture engine in use, this computation may have to be added into the packet capture process and thus increment per-packet processing latency and potentially damage capture performance. However, if the packet capture and processing processes are isolated one from the other and properly pipelined, this performance loss effect can be mitigated.

3.3 Capture Engine implementations

In this section, we present five proposed capture engines, namely: PF_RING DNA [Der04, Der05, RDC12], PacketShader [HJPM10], netmap [Riz12a, Riz12b, Riz14], PFQ [BDPGP12], Intel DPDK [Int14b], all of which have achieved significant performance levels. For each engine, we describe the system architecture (noting differences from the other proposals), which of the optimization techniques mentioned above have been applied, the API provided for client applications to develop network applications, and what additional functionality it may offer, while the following section will evaluate their performance. Table 3.1 shows a summarized qualitative comparison between the proposals under study. Note that this table also takes HPCAP [Mor12, MSdRR+14b] into account for comparative reasons, but, although briefly mentioned in this section, this capture engine will be deeply described in Chapter 4. We have not included some capture engines, previously proposed in the literature, because they are obsolete or unable to be installed in current kernel versions (Routebricks [DEA+09], UIO-IXGBE [Kra12]) or where a newer version of these proposals has been released (PF_RING TNAPI [FD10]). In Section 3.4 we discuss how to evaluate the existing packet capture engines. Finally, Subsection 3.4.2 shows the results from those tests and highlights the advantages and drawbacks of each capture engine, giving guidelines to the research community in order to choose the most suitable capture system.

3.3.1 PF_RING DNA

PF_RING DNA (Direct NIC Access) [RDC12] is a framework and engine to capture packets based on Intel 1/10 Gb/s cards. This engine implements pre-allocation and re-use of memory in all its processes. PF_RING DNA (Direct NIC Access) also allows building parallel paths from hardware receive queues to user processes, i.e., it allows a CPU core to be assigned to each receive queue whose memory can be allocated observing NUMA nodes, thus permitting the exploitation of memory affinity techniques.

PF_RING implements full *zero-copy*, i.e., PF_RING maps userspace memory into the DMA-able memory region of the driver allowing users' applications to access to card registers and data directly in a DNA fashion. This avoids the intermediation of the kernel packet buffer and reduces the number of copies. As previously noted, however, this is at the expense of a slight weakness to errors from user applications not following the PF_RING DNA API (which explicitly does not allow incorrect memory accesses) and this may potentially cause system crashes. In the rest of the proposals, direct accesses to the NIC are pro-

Characteristics/ Techniques	PF_RING DNA	PacketShader	netmap	PFQ	Intel DPDK	HPCAP
Memory pre-allocation and re-use	✓	✓	✓	✓	✓	✓
Parallel direct paths	✓	✓	✓	✓	✓	✓
Memory mapping	✓	✓	✓	✓	✓	✓
Zero-copy	✓	×	✓	×	✓	×
One-copy	×	✓	×	✓	×	✓
Batch processing	×	✓	✓	✓	✓	×
Byte-stream processing	×	×	×	×	×	✓
Capture & process isolation	×	×	×	✓	×	✓
CPU and interrupt affinity	✓	✓	✓	✓	✓	✓
Memory affinity	✓	✓	×	✓	✓	✓
Aggressive prefetching	×	✓	×	×	✓	✓
Multiple listeners	×	×	×	✓	✓	✓
Accurate timestamping	×	×	×	×	×	✓
Level modifications	D,K, K-U	D, K-U	D,K, K-U	D (minimal), K,K-U	D, K-U	D, K-U
API	libpcap	custom	standard libc	socket-like/C, C++, Haskell, pcap	custom	libpcap-like
Supported 10Gb NICs	Intel	Intel	Intel, Mellanox	Any	Intel, Emulex Cisco, Mellanox	Intel
Supported 1Gb NICs	Intel	Intel	Intel, Realtek, nVidia	Any	Intel	×

Table 3.1: Comparison of the diverse proposals (D=Driver, K=Kernel, K-U=Kernel-User interaction)

tected. PF_RING DNA behavior is shown in Fig. 3.7, where it can be observed that some of the steps that the NAPI approach follows disappear due to the use of the *zero-copy* technique.

PF_RING's API provides a set of functions for managing network devices and capturing incoming traffic. It works as follows: first, the network application must be registered with `pfring_set_application_name()`. Before starting the capture process, the socket descriptor can be configured via several functions, such as `pfring_set_{direction|mode|duration}()`. Once the socket is properly configured, traffic reception is enabled using the `pfring_enable_ring()` call. After this initialization process, user applications can receive new packets by calling the `pfring_recv()` function. Finally, when the user finishes capturing traffic `pfring_shutdown()` and `pfring_close()` functions are called. This process has to be replicated for each receive queue, as each user application will only receive the traffic corresponding to the RX queue the socket was configured for.

As one of the major advantages of this solution, PF_RING's API comes with a set of wrappers for the above-mentioned functions providing extensive flexibility and ease of use, essentially following the *de facto* standard of the libpcap library. Additionally, the API provides a set of functions for applying filtering rules (for example, BPF filters), network bridging, and IP reassembly. Both PF_RING DNA and a user-level packet processing library are freely available for the research community [nto14].

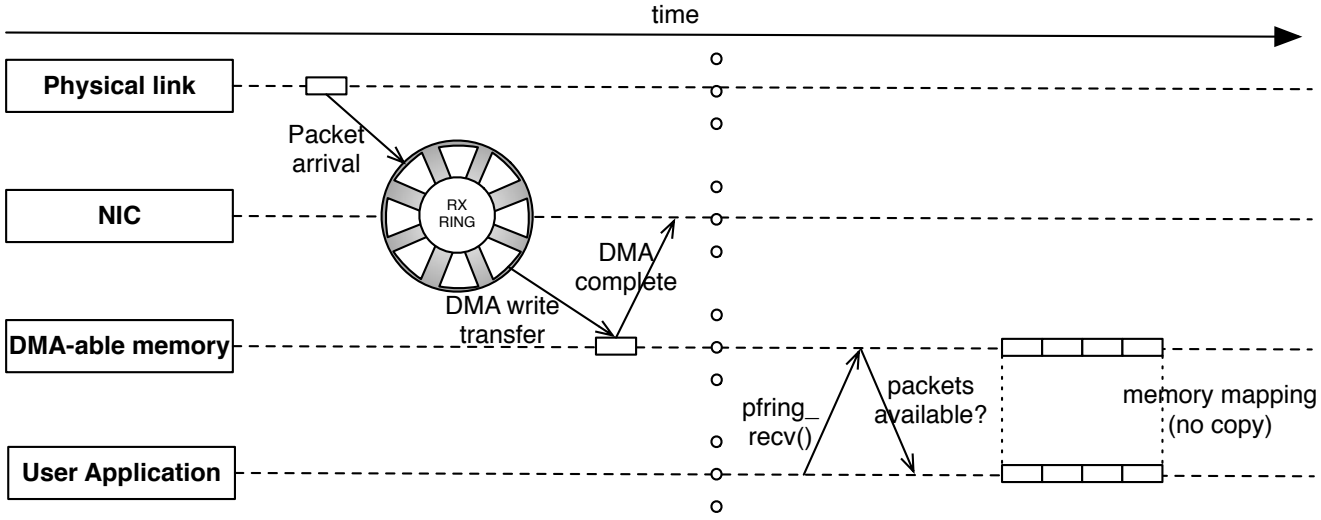


Figure 3.7: PF_RING DNA's RX scheme

3.3.2 PacketShader

The authors of PacketShader (PS) [HJPM10] developed their own packet capture engine to highly optimize the traffic capture module as a first step in the process of developing a software router based on GPGU able to work at multi-10 Gb/s rates. However, their efforts are applicable to any generic task that involves capturing and processing packets. They apply memory pre-allocation and re-use: specifically, two consecutive large memory regions are allocated: one for the packet data, and another for its metadata. Each buffer has fixed-size cells corresponding to the data and metadata for one packet. The size for each cell of packet data is set to 2048 bytes, which corresponds to the next highest power of two for the standard Ethernet MTU. Metadata structures are compacted from 208 bytes (as used by Linux's kernel) to only 8 bytes (96%) unnecessary or redundant fields.

Additionally, PS implements memory mapping to those data and metadata buffers, thus allowing users to avoid additional copies when accessing the information. In this regard, the authors highlight the importance of NUMA-aware data placement in the performance of its engine. Similarly, it provides parallelism between different RX queues, as their data may be independently processed at user level.

To reduce the per-packet processing overhead, batching techniques are used when a user-level application asks for new packets. For each batch requested, the driver copies data from the hardware descriptors to the above-mentioned packet data region and completes the corresponding metadata information. Once those copies are finished, the driver returns control to the user-level application which can now process the new packets without additional copies. In order to eliminate inherent cache misses, the modified device driver tries to prefetch the next packet's associated memory while still processing the previous one.

PS's API works as follows: (i) a user application opens a character device to communicate with the driver using the `ps_init_handle()` function, (ii) the application is attached to a given reception device (queue) using an `ioctl()` call, namely `ps_attach_rx_device()`, and (iii) kernel memory is allocated and mapped to userspace, in order to exchange data with the driver, using `ps_alloc_chunk()`. Then, when the user application requests new packets by means of an `ioctl()`, `ps_recv_chunk()`, PS driver copies a batch of them, if available, to the kernel packet buffer. PS kernel-user interaction during the reception process is summarized in Fig. 3.8.

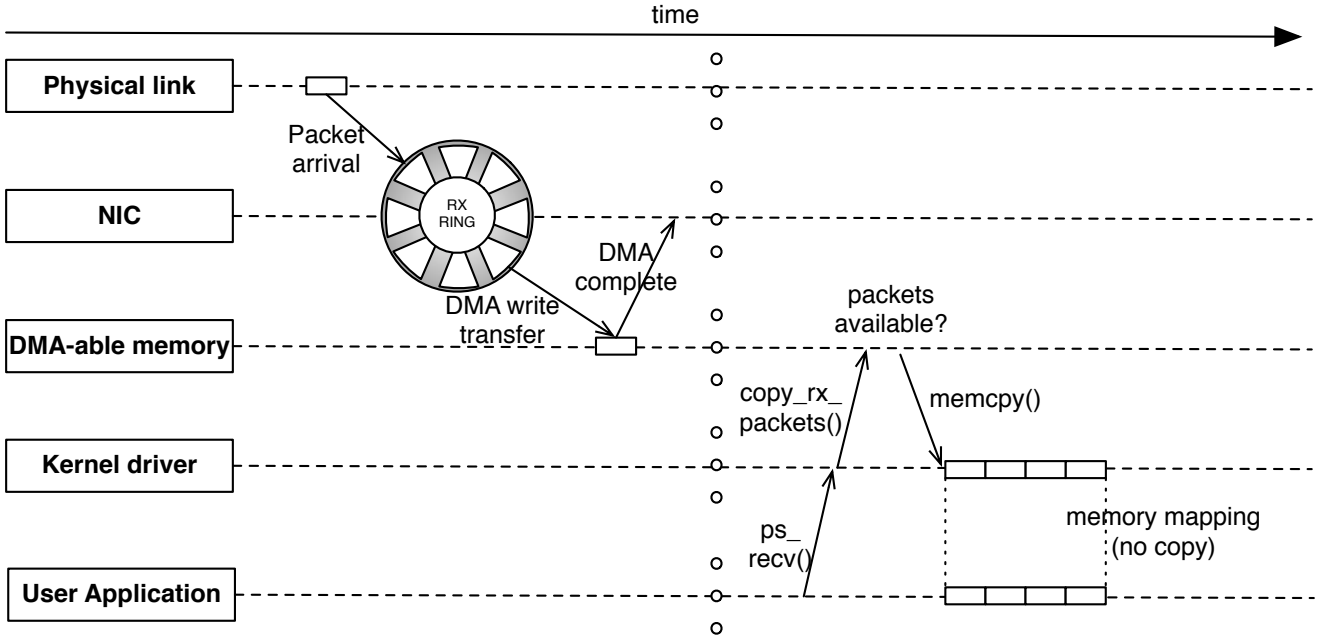


Figure 3.8: PacketShader's RX scheme

PS capture engine is available for the community [Pac12] under an open-source license. Along with the modified Linux driver for Intel 82598/82599-based network interface cards, a user library has been released to facilitate the usage of the driver. The release also includes several sample applications, namely a simplified version of `tcpdump`[6], an `echo` application which sends back all traffic received by one interface, and a packet generator which is able to generate UDP packets with different 5-tuple combinations at maximum speed.

3.3.3 netmap

The netmap [Riz12a] proposal shares most of the characteristics of PacketShader's architecture, i.e. it applies memory pre-allocation during the initialization phase, buffers of fixed sizes (also 2048 bytes), batch processing and parallel direct paths. It also implements memory mapping techniques to allow users' applications to access kernel packet buffers (direct access to NIC is protected) with a simple and optimized data structure. Its similarities with PacketShader also apply to the user-kernel interaction policy (see Fig. 3.8), except that netmap implements *zero-copy* from the NIC to buffers that will later be mapped from user level. Differently from other *zero-copy* solutions, netmap makes an emphasis on system's security and scalability by making sure no critical kernel structure is mapped by user applications and thus entail a potential threat.

This simple data structure is referred to as a *netmap memory ring* and contains information such as the ring size, a pointer to the current position of the buffer (*cur*), the number of received packets in the buffer or the number of empty slots for each reception or transmission buffer (*avail*), a set of flags related to the status, the memory offset of the packet buffer, and the array with the metadata information; it has also one slot per packet that includes the length of the packet, the index in the packet buffer and some flags. Note that there is a *netmap ring* for each RSS queue, for both reception and transmission directions, to allow exploiting parallel direct paths.

Netmap's API usage is intuitive: first, a user process opens a netmap device and maps kernel buffers with an `ioctl()` call. To receive packets, the process polls the driver about the number of available packets with another `ioctl()` and, when the system call is over, the lengths and payloads of the packets are available for reading in the slots of the *netmap ring* data structure. Note that this operation mode makes a batch of packets accessible for reading in each operation. Additionally, netmap supports blocking mode through standard system calls, such as `poll()` or `select()`, using the corresponding netmap file descriptors as arguments for those standard system calls. In addition, netmap

[6] `www.tcpdump.org`

comes with a library that maps libpcap functions to their netmap equivalents, thus allowing user applications to exploit netmap features without needing to be recompiled. A distinctive feature of netmap is that it works in an extensive set of hardware solutions: Intel 10 Gb/s adapters and several 1 Gb/s adapters (Intel, RealTek and nVidia), and even Mellanox's infiniband adapters. Netmap presents other additional functionalities as, for example, packet forwarding.

Netmap framework is available for FreeBSD (HEAD, stable/9 and stable/8) and for Linux [net14a]. The current netmap version consists of 2000 lines for driver modifications and system calls, as well as a C header file of 200 lines to help developers use netmap's framework from user applications.

3.3.4 PFQ

PFQ [BDPGP12] is a novel packet capture engine that enables packet sniffing in user applications with a tunable degree of parallelism. The approach of PFQ is different from the previous ones studied. Instead of carrying out major modifications to the driver in order to skip the interrupt scheme of NAPI or mapping DMA-able memory and kernel packet buffers to user space, PFQ implements a general architecture supporting any NIC driver.

PFQ has been designed so that it benefits from the vanilla network driver managing the NIC's hardware details. Those drivers connect with PFQ by redefining those functions that previously connected them with the operating system's network stack. In the latest version, those redefinitions are automatically made by a set of scripts, so users are isolated from those low-level details.

PFQ's kernel module implements a new layer, named *Functional Engine*, where packets are delivered by the NIC's driver. This layer distributes the traffic across different active receive sockets, without limits on the number of queues than can receive a given packet. The distribution tasks are carried out by independent packet fetcher threads. Importantly, those fetcher threads are executed in parallel and push the incoming packets' data in the *Functional Engine* with minimal overhead due to lockless access control policy. PFQ's architecture allows several fetcher threads to push the same packet to different sockets, which may imply more that one packet copy. However, those additional packet copies have low impact due to the use of caching mechanisms. Note that, as each receive socket has an independent lock-free queue, the packet capture performance is not limited by the slowest application fetching traffic from a common source. This functionality circumvents one of the drawbacks of using the parallel paths technique, namely scenarios where packets of different flows or sessions must be analyzed by different applications as explained in Subsection 3.2.2. Fig. 3.9 shows a temporal scheme of the process of requesting a packet in this

engine.

PFQ is an open-source package consisting of a Linux kernel module and a user-level library written in C++,available under GPL license in [PFQ15]. PFQ's API defines a `pfq` class which contains methods for device initialization and packet reception. Whenever a user wants to capture traffic: (i) a `pfq` object must be created using the provided C++ constructor, (ii) devices must be added to the object by calling its `add_device()` method, (iii) timestamping can be enabled using the `toggle_time_stamp()` method, and (iv) packet capture must be enabled using the `enable()` method. After initialization, each time a user wants to read a batch of packets, the `read()` method must be invoked. Using a custom C++ iterator provided by PFQ, users can read each packet in the received batch. When a user-level application has finished working with the `pfq` object, it is destroyed by means of its defined C++ destructor. A `stats()` method is also provided in order to obtain statistics about the received network traffic.

Moreover, PFQ supports high-level programming via functional programming languages, PFQ-Lang [BGPA14]. By using PFQ-Lang, developers can rapidly develop network processing applications in a flexible way while coping with high-speed rates.

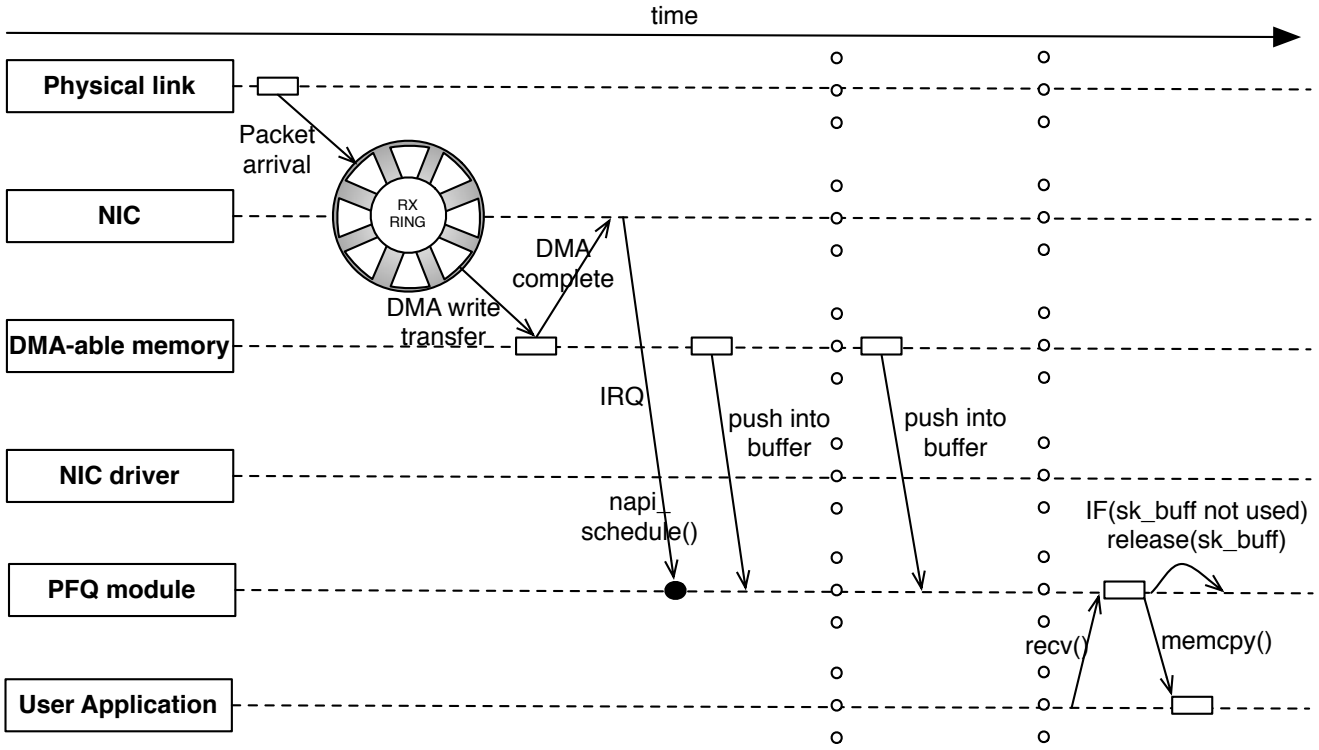


Figure 3.9: PFQ's RX scheme

3.3.5 Intel DPDK

Intel's Data Plane Development Kit (DPDK) [Int14b, DPD15] was created with the goal of providing a simple and complete framework for fast packet processing network application operating on the data plane. DPDK implements a new model for packet processing, following a modular approach. This way, users instantiate a set of worker threads, or listeners as previously called them, which will be able to receive and send packet from/to a certain distributor, as shown in Fig. 3.10. Users will place their logic inside the worker modules, and will be able to configure the distributor thread connected to each worker for RX and TX purposes. Those connections are made by means of packet rings, and they are managed automatically by the library provided.

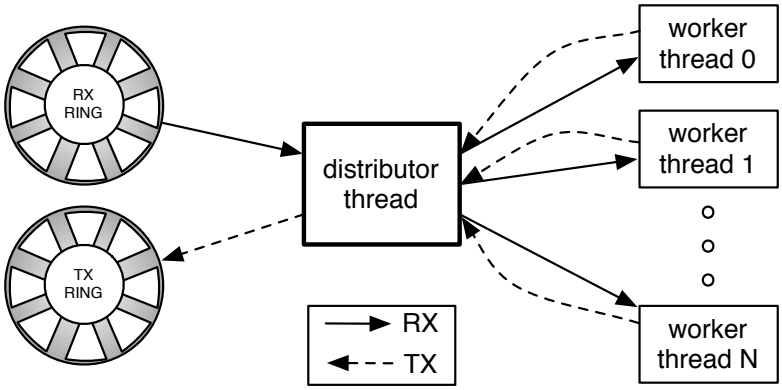


Figure 3.10: Intel DPDK's architecture

Intel's DPDK optimizes communications between the NIC and the distributor cores by pre-allocating and reusing data structures. These data structures are mapped from the user-level distributor threads and both memory and CPU affinity is carefully planned. Moreover, the multi-producer multi-consumer packet rings used to communicate the different threads in a DPDK-based application are generated using hugepages, which ensures these rings are always available on main memory and reduces the page fault overhead when accessing such regions. Note that each distributor thread will be in charge of dispatching the packets corresponding to a certain RSS queue from a certain NIC. Intel's DPDK architecture allows several worker threads to fetch packets from the same distributor, but each thread will only receive the set of packets previously requested. Additionally, a single worker thread may receive packets from different distributors.

An application making use of Intel DPDK has to initialize its resources by calling the `rte_eal_init()` function. Distributor threads will call the `rte_dis-`

`tributor_process()` function to begin capturing packets from the NIC. Meanwhile, worker threads make use of the `rte_distributor_get_pkt()()` call to request a new packet once it has finished processing the previous one, as decribed in [Int14a]. These library calls isolate users from operating with the intermediate packet rings used to communicate with the different modules. In order to optimize packet-transfer operations, local cache lines are shared between the distributor and worker threads. This feature makes it impossible for two worker threads to process the same packet simultaneously. Thus, every time a worker thread is done with a packet, it must notify the distributor via the final parameter of the `rte_distributor_get_pkt()()` function, so the distributor knows this packet can be sent to another worker if requested. Note that the packets are distributed to the diverse worker threads using meta-structures pointing to the corresponding NIC's packet descriptor, which are the ones cloned when several workers ask for the same packet. This allows sharing packet data without additional copies at the expense of potentially locking the descriptors for a longer period.

In order to keep packets ordered they are identified via a tag (the 5-tuple hash calculated by the NIC), thus packets with the same tag will be orderly processed by all worker threads processing them. Note that this policy may include latency in the processing of some packets and packet order is only guaranteed between packets with the same tag. Additionally, workers can temporally stop processing packets by using the `rte_distributor_return_pkt()` function and resume their execution afterwards, which may be interesting in order to save CPU power depending on network load.

Finally, Intel DPDK comes with a set of libraries to ease the user's work with packets at the different network layers. They help managing issues such as IP fragmentation, TCP re-assembly, etc. Nevertheless, there is no reference regarding those libraries' performance.

3.3.6 HPCAP

HPCAP is another packet capture engine that has been designed along the work of this thesis with the aim of addressing a set of features that has been ignored by the rest of packet capture engines: accurate timestamping, high-performance packet storage, duplicate packet removal and capture and packet processing overlapping. Due to the relevance of HPCAP in the context of this thesis, Chapter 4 has been devoted to give a detailed description of HPCAP and its features.

3.4 Testing your traffic capture performance

A quantitative comparison between capture engines based on the literature is not possible for two reasons: first, the hardware used by the different studies is not equivalent —in terms of type and clock speed of the CPU, amount and clock speed of main memory, server architecture and number of network cards. Second, the performance metrics used in the different studies are not the same —with differences in the type of traffic and in the measurement of the burden on CPU or memory.

Consequently, let us first elaborate a fair basis for comparison of capture engines at 10 Gb/s rates. And, then, let us carry out our own quantitative comparison based on such a common basis using the same hardware.

3.4.1 General concerns

The first metric to be considered is the amount of traffic that an engine may process. We assume a fully-saturated link of 10 Gb/s, and both constant-sized packets and real trace with variable packet sizes are injected. The constant-size oriented tests aim at evaluate worst case scenarios. As the effort to capture a packet is almost constant but small packets have smaller time gap between consecutive packets and consequently less time to carry out any subsequent task, those scenarios with small-size packets are the most challenging. Unfortunately, small-sized packet traffic profiles are not uncommon on the Internet as for example VoIP traffic, distributed databases or even anomalous traffic [KWH06].

According to 10GbE standard, 60-byte packets in a 10 Gb/s fully-saturated link gives a throughput in Mp/s of 14.88: 10^{10} / ((60 + 4 (CRC) + 8 (Preamble) + 12 (Inter-Frame Gap)) · 8), and an effective throughput of 7.14 Gb/s (due to the preamble and inter-frame gap overheads). Equivalently, if packet sizes grow to 64 bytes, the throughput in Mp/s decreases to 14.22 and the effective throughput rises to 7.27 Gb/s. Table 4.3 shows how those values evolve for the packet sizes used in our test experiments. In order to avoid dealing with these values that depend on packet sizes, we find it more intuitive to evaluate this metric in terms of the percentage of packets received for each scenario.

Similarly, it is important to agree if the four Ethernet CRC bytes are considered in the packet size or not. In the following analysis, when referring to X-byte packets, those X bytes will not take Ethernet's CRC into account. In order to avoid Ethernet management mechanisms to contaminate network measurements, pause frame negotiation and hardware offload capabilities must be

disabled as shown in Section A. If pause frame negotiation is enabled the receiver side could send a pause frame when it is congested and prevent the sender from transmitting new frames. On the other hand, offload mechanisms allow NICs to merge small packets belonging to the same flow into bigger ones and may affect network diagnosis algorithms.

Max. throughput	Packet size (bytes, CRC excluded)								
	60	**64**	**128**	**256**	**512**	**750**	**1024**	**1250**	**1514**
Gb/s	7.14	7.27	8.42	9.14	9.55	9.69	9.77	9.81	9.84
Mp/s	14.88	14.21	8.22	4.46	2.33	1.62	1.19	0.98	0.82

Table 3.2: Maximum throughput in terms of packets and bits for different packet sizes in a full-saturated 10GbE link

It is worth remarking that extremely positive results may arise in short duration experiments due to caching effects. Thus experiments regarding packet capture performance must be carried out for a period of time such that the amount of traffic processed does not fit in system memory. Thus, all the experiments carried out in this section have been obtained by replaying the corresponding traffic over a period of 30 minutes.

In order to provide a comparison as fair as possible, capture performance results refer only to a simple packet receive and update counters application developed for each engine, without any additional characteristic except from timestamping when it was possible (see Table 3.1). In all cases, we have paid attention to NUMA affinity by executing capture threads in the processor the NIC is connected to, which is only possible when there are less concurrent threads than cores available in the target NUMA node. In fact, ignoring NUMA affinity entails extremely significant performance losses, especially in the case of the smallest packet size where performance may be halved.

Another metric that must be considered is the number of CPUs and usage made by a solution. More computational power is available for additional tasks if the number of CPUs and usage are low. However using more than a couple of CPU and RSS queue may cause collateral effects such as packets belonging to the same session or flow to be processed by different CPUs. This may be a vital drawback for certain monitoring applications. Instant CPU usage measurements are obtained using the `pidstat`[7] command, instead of `ps`[8] command that provides the amount of time that the process has been using the processor since it started.

The amount of system memory used is the third main metric to take into account. High memory requirements may increment the cost of the monitoring

[7] http://linux.die.net/man/1/pidstat
[8] http://linux.die.net/man/1/ps

system, and limit the number of additional processes that can be simultaneously executed. Memory usage for a certain process can be obtained by means of the Linux `ps` command.

Additional non-performance-related features should also be taken into account. For example, the accuracy of packet timestamping is a vital factor for QoS monitoring systems. As not all the capture engines evaluated support packet timestamping, this factor has not been included in the evaluation and readers are encouraged to take a look at [MSdRR+12]. Importantly, the ease of use of each capture system, together with their related APIs, is an important factor which won't be evaluated here due to its subjectivity. For a quick qualitative comparison between the packet capture engines see Table 3.1.

3.4.2 Captures engines performance evaluation

After detailing the main characteristics of the most prominent capture engines in the literature, we turn our focus to their performance in terms of percentage of packets correctly received. It should be noted that a quantitative comparison between them based on the literature is not an easy task for two reasons: first, the hardware used by the different studies is not equivalent (in terms of type and clock speed of the CPU, amount and clock speed of main memory, server architecture and number of network cards); second, the performance metrics used in the different studies are not the same, with differences in the type of traffic and in the measurement of the burden on CPU or memory.

For this reason, we have stress-tested the engines described in the previous section using the same architecture. Specifically, our testbed setup consists of two machines (one for traffic generation purposes and another for receiving traffic and evaluation) directly connected through a 10 Gb/s fiber optic link. The receiver side is based on Intel Xeon with two 6-core processors each running at 2.30 GHz, with 128 GB of DDR3 RAM at 1,333 MHz and fitted with a 10 GbE Intel NIC based on the 82599 chip. The server motherboard model is Supermicro X9DR3-F with two processor sockets and three PCIe 3.0 slots per processor, directly connected to each processor, following a scheme similar to that depicted in Fig. 3.2(b). The NIC is connected to a slot corresponding to the first processor or NUMA node. The system runs an Ubuntu server 12.10 with a 3.8.0.29-generic kernel.

The sender uses a HitechGlobal HTG-V5TXT-PCIe card with a Xilinx Virtex-5 FPGA (XC5VTX240) and four 10 GbE SFP+ ports. Using such a hardware-based sender guarantees accurate packet interarrivals and 10 Gb/s throughput regardless of packet size. The sender server also has an Intel 82599 NIC and a software traffic generator, which has been developed as a tool on top of Packet-

Shader's [HJPM10] API capable of replaying PCAP (Packet Capture API) traces at variable rates.

For our experiments, we have used both synthetic and real traffic. Synthetic traffic is sent using the FPGA generator and consists of TCP segments encapsulated into fixed-sized Ethernet frames, forged with incremental IP addresses and TCP ports. Note that synthetic traffic allows us to test worst-case scenarios in terms of byte and packet throughput, but they are not useful for testing the flow-related modules. Real traffic is generated using a software generator replaying a trace consisting of a packet-level trace sniffed on an OC192 backbone link of a Tier-1 ISP located between San Jose and Los Angeles (both directions), available from CAIDA [WAcAa]. Replaying the backbone trace at line-rate leads to a throughput of 9.59 Gb/s[9], and 1.65 Mp/s. Our packet capture experiments have taken into account two factors: the number of available queues/cores and packet sizes, and their influence on the percentage of correctly received packets.

In addition to the capture engines presented in this chapter, we have considered it interesting to evaluate the packet capture performance offered by the traditional solution, i.e. the `ixgbe` driver following a NAPI approach plus the use of the PCAP library. It is worth pointing out that the behavior of the Linux version of netmap is to set the number of RSS queues to match the number of cores. Thus, we have had to modify the number of queues used in netmap by writing a 1 or a 0 in the `/sys/devices/system/cpu/cpuX/online` file to respectively enable or disable CPUs. Regarding netmap and PFQ, we evaluated their performance by respectively installing the netmap-aware `ixgbe` driver and the `ixgbe` vanilla driver compiled with a script shipped with PFQ. We wanted to evaluate each capture engine using a number of queues ranging from 1 to 12 (as our system has 12 CPUs). It is worth noting that in the case of Intel DPDK 11 is the maximum amount of queues reached, because the capture system needs one core to be reserved for management purposes, reducing by 1 the number of cores eligible for packet capture. Note that performance comparison also takes HPCAP [Mor12, MSdRR+14b] into account, but this capture engine's results will be further discussed in Chapter 4.

First, Fig. 3.11 aims to show both the worst-case scenario of a fully-saturated 10 GbE link (packets with a constant size of 60 bytes) and an average scenario. Note that the worst case represents an extremely demanding scenario, 14.88 Mp/s, but probably not very realistic given that the average Internet packet size is clearly larger [CAI].

[9]This is the maximum achievable speed due to the preamble and inter-frame gaps that the Ethernet protocol requires.

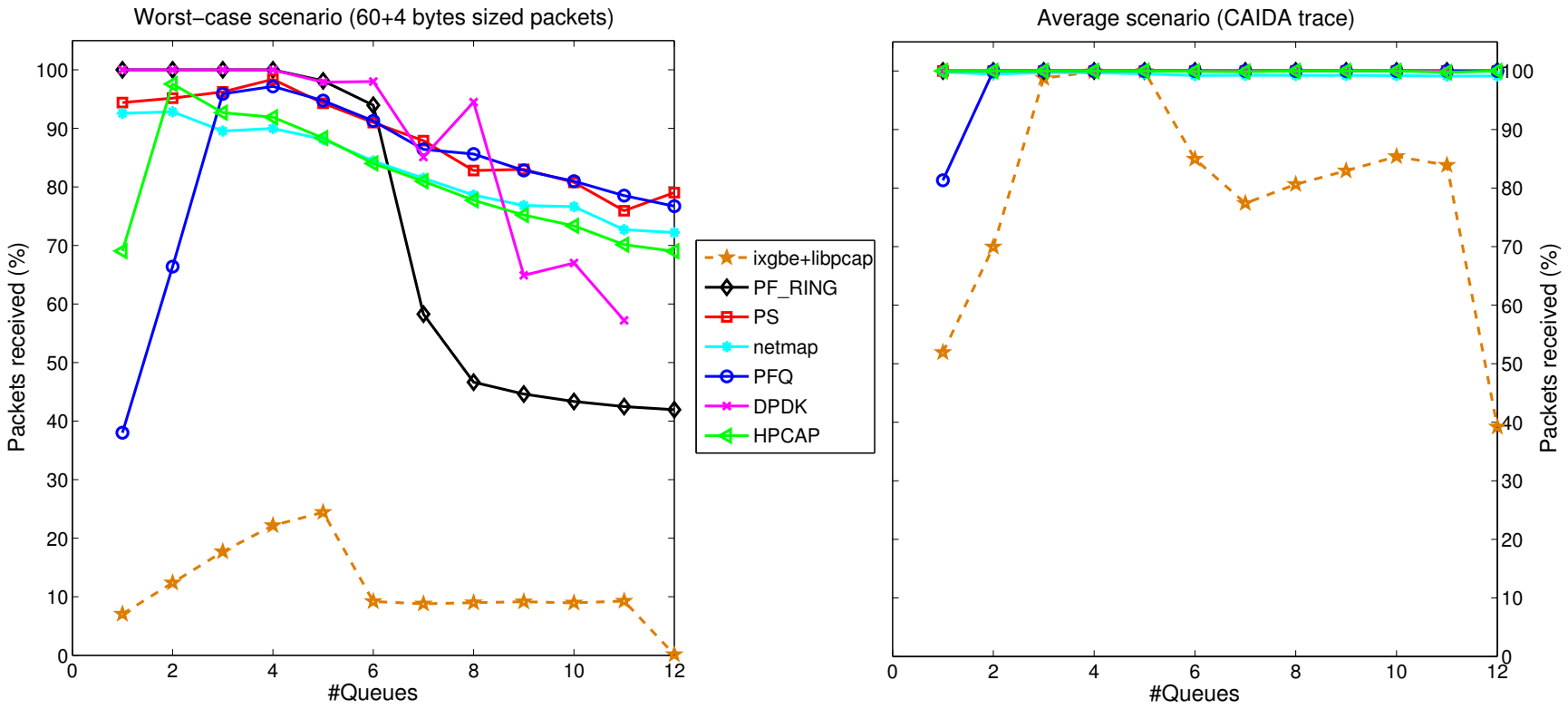


Figure 3.11: Engines' performance for worst and average scenarios

In the worst-case scenario (left-hand side of Fig. 3.11) the traditional solution (`ixgbe` + PCAP library) reaches peak performance with 5 queues, capturing over 24.4% of the incoming packets. PF_RING DNA and Intel DPDK are the only capture engines that achieve line rate if fewer than 4 receive queues are used. PacketShader is also able to handle nearly the total throughput when the number of queues ranges between 1 and 4, after which point the performance declines. Netmap has a performance level similar to PacketShader with one queue, but capture performance worsens as more receive queues are used. Conversely, PFQ increases its performance while the number of queues rises to a maximum with four queues, when improvement stalls. Finally, HPCAP shows peak performance, capturing 97.6% of the traffic, when using two queues but this figure reduces as the number of queues increases.

In the average scenario shown on the left-hand side of Fig.3.11, capture with `ixgbe` shows the best performance using four or five queues, with 0.1% of incoming packets lost. In that same scenario, PF_RING, PacketShader, PFQ (with two or more queues), Intel DPDK and HPCAP are capable of working with 0% packet loss. Netmap suffers a slight packet loss ranging from 1% to 0.1% (reached with four queues).

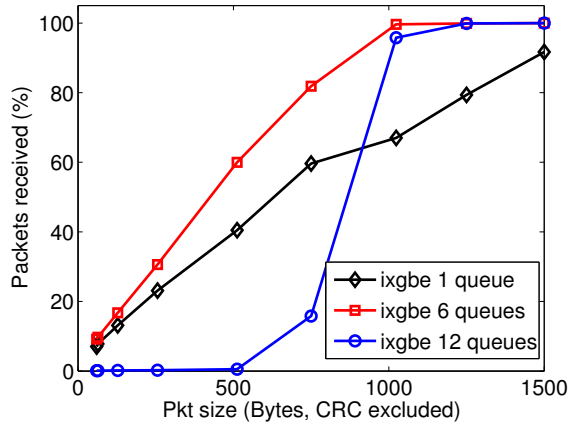
In conclusion, our tests have confirmed that all capture engines suffer from scalability issues in the worst-case scenario. This effect becomes more relevant when the number of cores in use needs more than one NUMA node. To further investigate this phenomenon, Fig. 3.12 depicts the results for the packet sizes shown in Table 4.3 using one, six and twelve queues respectively.

PF_RING DNA shows the best results with one and six queues. It does not show packet losses for any scenarios except for those with packet sizes of 64 bytes and, even in this case, the figure is very low (about 4% with six queues and lower than 0.5% with one). Surprisingly, increasing packet sizes from 60 to 64 bytes entails a degradation in the PF_RING DNA performance, although the performance recovers 0% loss rates beyond these packet sizes. Note that, as stated before, larger packet sizes imply lower throughputs in terms of Mp/s. According to [Riz12a], investigation in this regard has shown that this behavior is due to the design of NICs and I/O bridges that make certain packet sizes fit better with their architectures.

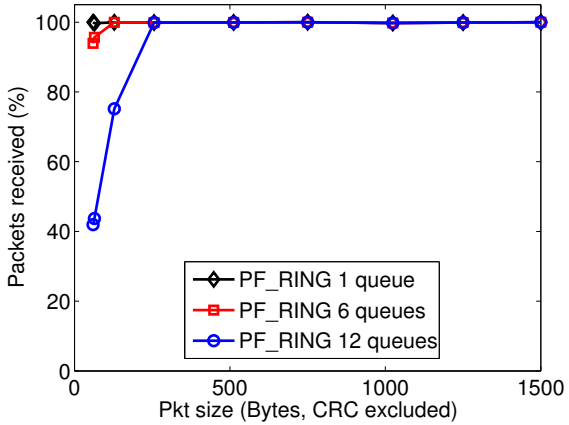
In a scenario where one single user-level application is unable to handle all received traffic, it may be of interest to use more than one receive queue (with one user-level application per queue). With the maximum number of queues, PacketShader and HPCAP have shown comparatively the best result, although, like PF_RING DNA, they perform better with a smaller number of queues. Specifically, for packet sizes larger or equal to 128 bytes, they achieve full packet received rates regardless of the number of queues. Conversely, Intel DPDK shows the worst results for the maximum number of queues, showing packet losses for

packets below 750 bytes.

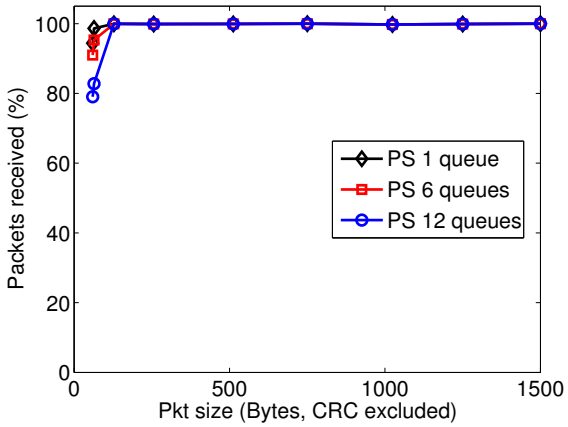
Analyzing PFQ's results, we note that this engine also achieves 100% received packet rates but, conversely to the other approaches, works better with several queues. It requires at least six to achieve no losses with packets of 128 bytes or more, whereas with one queue, packets must be larger or equal to 256 bytes to achieve full rates. This behavior was not unexpected due to the importance of parallelism in the implementation of PFQ.



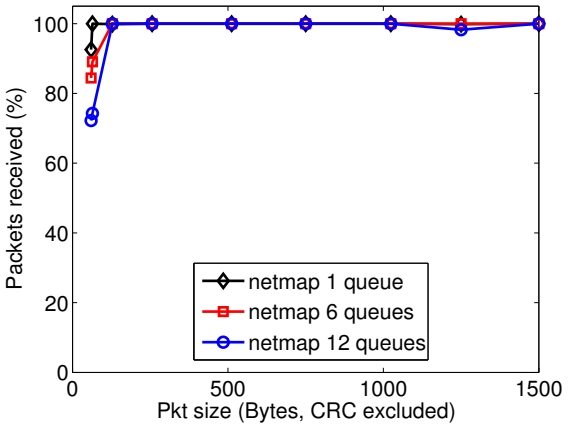
(a) NAPI scheme



(b) PF_RING DNA

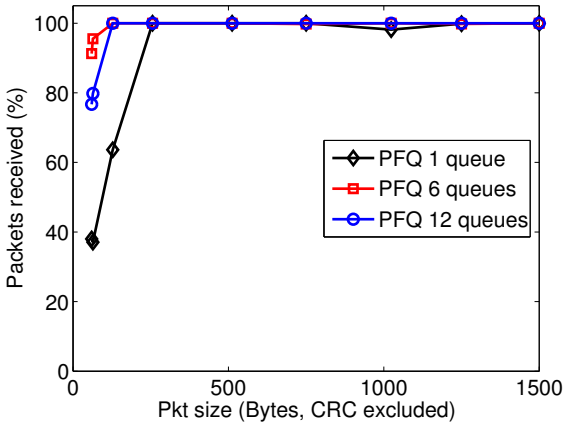


(c) PacketShader

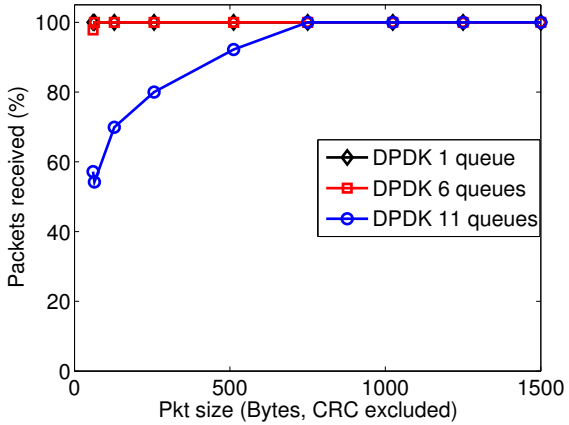


(d) Netmap

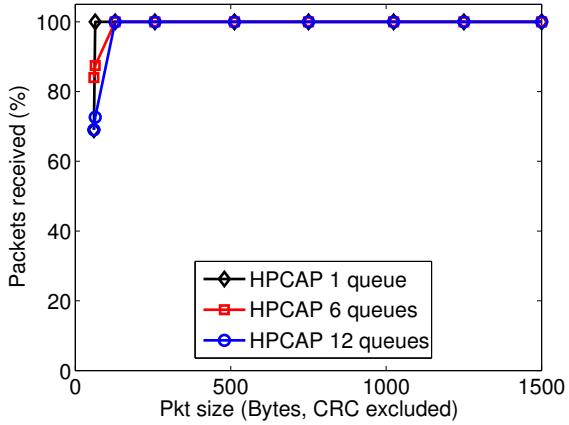
Figure 3.12: $(a), (b), (c), (d)$ Packets received for different numbers of queues and constant packet sizes in a full-saturated 10 Gb/s link



(e) PFQ



(f) Intel DPDK



(g) HPCAP

Figure 3.12: $(e), (f), (g)$ Packets received for different numbers of queues and constant packet sizes in a full-saturated 10 Gb/s link

We have found that these engines may cover different scenarios, even the most demanding ones, distinguishing them on the basis of two criteria: whether or not we may assume the availability of multiple cores, and whether or not the traffic intensity (in Mp/s) is extremely high (for example, packet size averages smaller than 128 bytes, which is not very common). In other words, if the number of queues is not relevant, given that the capture machine has many cores available, or no other process is executing except for the capture process itself and the traffic is not maximal, PFQ seems to be a suitable option. On the other hand, if traffic intensity is near to maximum, PF_RING, PacketShader, netmap, Intel DPDK and HPCAP present a good compromise between the number of queues used and the performance offered.

Nonetheless, multi-queue scenarios are often not adequate. For example, accurate timestamps may be necessary [MSdRR+12], packet disorder may be a significant drawback (depending on the application running on top of the engine) [WDC11], or it may simply be of interest to save cores for other tasks. In such a scenario, PF_RING DNA and Intel DPDK are great options, as they show (almost) full rates regardless of packet size even with only one queue (thus, avoiding any objections due to parallel paths).

Capture engine	Number of RSS queues	Memory in use (MB)	Number of cores used	Average CPU usage (per active core)
`ixgbe`	5	9.2	6	5×3.4% (kernel) + 82.7% (user)
PF_RING	1	110.3	1	75.8% (user)
PS	1	12.6	1	77.4% (user)
netmap	1	351.8	1	66.2 (user)
PFQ	1	425.2	1	99.9% (user)
DPDK	1	2192.4	2	99.2 (distributor) + 99.3 (worker)
HPCAP	1	1054.9	2	99.7 (kernel) + 99.8 (user)

Table 3.3: Memory and CPU usage in a 10 Gb/s average scenario

In addition to the packet capture performance figures that each capture engine offers, users may also need to decide which solution they can use in terms of resource consumption. Table 3.3 shows the results for each capture engine in terms of CPU and memory usage. The table shows the resource consumption of each capture engine with the least resource-consuming configuration capable of capturing all packets (99.9% of the packets for `ixgbe`) mentioned previously: a CAIDA trace replayed at line-rate. Note that PF_RING, PacketShader, netmap and PFQ use as many cores as there are RSS queues. On the other hand, the default usage of `ixgbe` and Intel DPDK use one more core than the number of CPUs used. In the case of `ixgbe` the reason is that the kernel fetches packets from each queue into a different core, and a user-level application in a

different core aggregates the traffic received by each queue. Intel DPDK uses one core to receive traffic from each RSS queue, where the distributor cores will be executed, and one additional core for each worker thread instantiated. HPCAP, on the other hand, uses two cores per RSS queue, as it instantiates one kernel-level thread to fetch the packets from the network and copy them to the intermediate buffer, and a user-level thread to process the packets already stored in that buffer.

3.5 Use cases of novel capture engines

Thus far, we have reviewed the first three layers of commodity high-performance network systems, NIC, driver and framework, paying special attention to the combination of the last two in a capture engine. In this section, we turn our attention to the upper layer: the final services or applications that are built on top of such engines. We first provide some advice on how to translate the new paradigm ushered in by the packet engines studied into the development of these services. We now turn our attention to both performance and cost of an extensive set of different systems, which may serve as the state-of-the-art bounds that any novel application should overcome to be of interest. Moreover, we believe the service examples reviewed may awaken new ideas and utilities in both the research community and practitioners.

3.5.1 Creating a high-performance network application

In these paragraphs, we highlight the main principles to apply when designing and developing high-performance network applications in contrast with traditional approaches. Then, we illustrate the implementing of such principles with success cases of real implementations of high-performance network applications.

Although the optimization of network drivers and capture engines is required to reach high performance, it has been shown to be insufficient in a final system [KMC+00]. In other words, if the high-level application is not capable of processing all traffic captured and provided by the capture engine, then we will only have shifted the bottleneck to the upper layer and we will not have solved the problem. Thus, regardless of the capture engine chosen to capture packets, application developers must follow ideas equivalent to those presented in Section 3.2.2:

- **Avoiding (as far as possible) per-packet operations**: Due to the challenging packet rates to which we have to face, we must reduce per-packet operations. In order to do this, we may implement different approaches (the decision will depend on our specific application): (i) byte-stream oriented: to access packet buffers as a stream of consecutive bytes instead of packet-by-packet, which is useful when storing packets to hard drive (we do not need to parse packet headers), e.g., `hpcapdd` [MSdRR+14b] implements such a technique; (ii) batch processing: to access packet buffers when there are several packets to process (or a timeout has expired), e.g., the

traffic classifier proposed in [SdRRG+12] implements such an approach.

- **Using smarter memory structures**: In the case of packet capture engines too, memory management (allocations, transfers, deallocations...) is a paramount task in terms of performance. We should pre-allocate and reuse memory resources instead of allocating and freeing memory for each packet (e.g., the router software implemented in [RCC12] is accelerated by up to 3x-4x thanks to recycling memory). Most network applications aggregate packets at other levels (such as flows, connections, sub-networks) and then deal with such aggregations. Generally, this aggregation process is done by using hash tables. To optimize this aggregation process and efficiently distribute packets in bins, we should tune hash functions. For instance, authors in [SGV+10] propose to use the Zobrist hashing algorithm, thus reducing memory requirements. In general, we must pay attention to memory management operations to achieve high performance, `VoIPCallMon` [GDSdRR+14], for example, uses tailored data structures to optimize call insertion, export and deletion.
- **Taking advantage of parallelism at user-level**: The parallelism achieved in the low-level capture process is useless unless further exploited in the application layer. This parallelism must be implemented to make the most of multi-core architectures by both processing traffic for each receive queue in a different core and carrying out each system task in a different core, e.g., [MSdRR+14b, SdRRG+12]. Furthermore, parallelism can be increased by making use of GPGUs as in [ND10b], multiplying the capacity with the number of concurrent threads in each GPGU (normally, hundreds of them). For instance, the software router proposed in PacketShader [HJPM10], MIDeA [VPI11] and the classification system proposed in [SGV+10], take advantage of parallelism provided by GPGUs.
- **Designing efficient inter-module communication and to properly set affinity**: Although parallelism may potentially increase the processing capacity of our application, we have to be careful with communication between the system's different modules so they do not cause bottlenecks due to locking or other issues. Thus, we have to use shared memory structures that avoid, or at least reduce, locking issues (as used in [MSdRR+14a, SdRRG+12]) and to optimize the way applications transfer data from main memory to GPGU and vice versa, e.g. by adjusting transaction size to a given optimal fixed size [VPI11]. Additionally, to make the most of parallelism and memory locality it is necessary to set CPU affinity properly, i.e. to tie each thread/process to an appropriate NUMA node.
- **Achieve flexible solutions**: Beyond performance, thanks to the use of commodity hardware and software-only solutions, we can cut capital expenditure. However, if these solutions are not flexible enough to adapt to new requirements and demands, then we will need to redesign the applications,

thus pushing up their cost, even to the point of exceeding commercial solutions' costs. Thus, we must design our network applications bearing in mind flexibility and re-usability, without losing sight of high performance. Blockmon [DdH+12, DPHB+13a] is a good example of this where each block represents a different subtask of a given application. For instance, we have a block for packet capturing and, as long as the system is well designed and implemented, we can replace one packet capture engine with another depending on system requirements.

3.5.2 Application examples

We have found in the literature a number of final systems whose high performance deserve attention, but these results are even more remarkable because of their limited costs. Through being based on commodity hardware, their costs are less than a few thousand dollars, apart from the cost of the common 10 Gb/s NICs.

The implementation of software routers, network intrusion detection systems, traffic classifiers , in particular and other specific monitoring tasks have been the leading examples. Table 3.4 summarizes the performance and characteristics of some of these applications. Importantly, we note that some approaches are based on systems whose first step lies in the distribution of the load between several subsystems or clusters [SWF07]. These subsystems may also work in isolation but at a lower rate. Thus, in order to show a fair performance comparison, we include the results for isolated systems instead of sets of distributed ones. Note that, in all cases leveraging an external traffic splitter (at higher cost) or with *ad-hoc* traffic balancing schemes, the load could be distributed over different machines to increase overall performance nearly linearly.

System Name	Category	Capture Engine	Application	Throughput	Comments
PacketShader [HJPM10]	Software Routers	PacketShader	IPv4 forwarding IPv6 forwarding OpenFlow Switch IPSec gateway	39 Gb/s 38 Gb/s 32 Gb/s 10.2 Gb/s	Packets of 64B
Ad-hoc version Click router [RCC12]	Software Routers	netmap	IPv4 forwarding	6-8 Gb/s	Packets of 64B
MIDeA [VPI11]	NID	PF_RING	Snort NID	7.2 Gb/s Below 2 Gb/s 5.7 Gb/s	Packets of 1500B Packets of 200B Real traces
Szabó et al. [SGV+10]	Traffic classification	PF_RING	DPI Connection pattern Port based	6.7 Gb/s	Real traces
Santiago et al. [SdRRG+12]	Traffic classification	PacketShader	Statistical classification	10 Gb/s	Packets of 64B
hpcapdd [MSdRR+14b]	Monitoring	HPCAP	Packet storage	10 Gb/s	Packets of 64B
ffProbe [DDDS11]	Monitoring	PF_RING DNA	Netflow construction	10 Gb/s 7 Gb/s	Packets of 500B Packets of 60B
VoIPCallMon [GDSdRR+14]	Monitoring	HPCAP	VoIP tracker	10 Gb/s	Codec G.711
Blockmon [DPHB+13a]	Monitoring	PFQ	Heavy hitters SYN flooding VoIP anomaly	3.8 Gb/s 5.5 Gb/s 10 Gb/s	Codec G.711

Table 3.4: Summary of the performance and characteristics of a set of typical high-performance network applications using commodity hardware

Software Routers

The use of commodity hardware to perform high-speed tasks started with the significant increase in popularity achieved by software routers in recent years. Software routers present some interesting advantages with respect to hardware-designed ones, essentially cost and flexibility. This increase has been strengthened by multiple examples of successful implementations and by the appearance of GPGUs [ND10b] which multiply the parallelism between processes while the cost remains low.

The authors of PacketShader [HJPM10], as stated in Section 3.3, developed their own network application with both novel and optimized packet capture characteristics, but, in fact, their final target was to develop a software router able to work at multi-10 Gb/s rates. To this end, they proposed to move the routing process from the CPU to GPGUs, where hundreds of threads can be executed in parallel. As most software routers operate on packet headers, the use of GPGUs and parallel threads fits perfectly. Therefore, it is intuitive to bind each received packet to a thread in a GPGU, multiplying the capacity of the router by the number of concurrent threads in each GPGU. The results are astonishing given the use of commodity hardware and software solutions. Specifically the cost of the testbed server is roughly $ 7000. IPv4 forwarding service achieves a throughput of 39 Gb/s with 64 byte packets, and even better results for larger packet sizes in a unique machine. The results for IPv6 forwarding are only slightly lower: 38 Gb/s. In addition to IP routing, the authors also evaluated the performance of their approach working as an OpenFlow Switch and an IPSec (Internet Protocol security) gateway. The results show that they are able to switch at 32 Gb/s, and they obtained a throughput of 10.2 Gb/s for IPSec overcoming commercial solutions.

Similarly to PacketShader's approach, the authors that proposed netmap illustrated their engine's functionality with a router software application [RCC12], specifically, a Click Modular Router developed about fifteen years ago [KMC+00]. Conversely to PacketShader system, they neither use GPGUs to parallelize tasks nor any further code optimization, thus the performance at application-level was lower, about 2 Gb/s witht 64 byte packets although the capture engine worked at much higher rates. However thanks to this road block in the study of capture engines, they found that the capacities these novel devices achieved was far over the capacities of many of the application developed decades ago. The previous section explained what advice a network-software developer must follow in this task. In the specific case of Click, the authors pinpointed that the process of allocating memory in the C++ code was not ideal. In the original version, two blocks of memory were reserved per packet: one for the payload and another for its descriptor. However, this was not necessary as the memory can be recycled inside the code to avoid the allocation of new blocks and us-

ing fixed-size objects. The improvement ranges between 3x and 4x depending on the size of the batches, which represents a significant gain. The cost of the system is estimated to be about $ 2000.

Network Intrusion Detection Systems

Network Intrusion Detection (NID) has become one of the most active research topics in the field of monitoring given its importance in network security. There are essentially two approaches to implement NID systems: those based on identifying (anomalous) characteristics of the traffic (for example, the distribution of port numbers' popularity) and those related to Deep Packet Inspection (DPI), which basically consists in searching for a given signature in the traffic payload. While the former typically results in faster speeds, the use of DPI approaches tends to be more accurate.

The authors in [VPI11] evaluated this latter option proposing a full software implementation (called MIDeA) based on the PF_RING as capture engine. As application, they present a prototype implementation of a NID system based on Snort, the *de facto* standard software for this purpose, which includes more than 8,192 rules and 193,000 strings for string matching purposes. Similar to the previously explained PacketShader application, the key to its implementation is the use of GPGUs. Especially, the optimized the way the application loaded data from/to the GPGU by adjusting data transfers to multiples of the minimum size for memory transaction on the GPGU used. The results show that their system, currently valued at roughly $ 2,500, is able to achieve 7.22 Gb/s for synthetic traces in the ideal scenario of 1500 byte packets. This represented an improvement of more than 250% over traditional multi-core implementations. However, the performance remains below 2 Gb/s in the case of packet sizes of 200 bytes. While this presents a significant reduction, it is worth noting that the average Internet packet size is clearly larger than such 200 bytes. In fact, when the system is evaluated with real traces, it achieves rates of 5.7 Gb/s.

Traffic Classification Technology

Traffic classification technology has gained in importance in recent years, as it has proved useful in tasks such as accounting, security, service differentiation policies, network design and research [CKS+09]. Since its inception to date, the research community has paid special attention to improving the accuracy of this technology, but it has not been until recently that the evaluation of their performance has gained relevance. Thus, some of the most accurate mechanisms have seen that their execution on high-speed networks is hardly likely. This has increased the interest in mechanisms to reduce the application burden required

by classification. These mechanisms are essentially DPI and Machine Learning (ML) tools [NA08], once port-based classification has been ruled out because of the widespread use of random port numbers by P2P (Peer-to-peer) and VoIP applications.

In this regard, the authors in [SGV+10] show a system to classify traffic by leveraging both DPI and connection patterns (i.e., analyzing the interaction in terms of number of connections or ports involved in inter-host communications). The capture engine is implemented as a part of the system, but its foundations are equivalent to PF_RING. To deal with PF_RING's packet rates, the authors also exploit the parallelism provided by GPGUs. In this case, the authors pay attention to the fact that GPGUs' fast cache memory tends to be too small to allocate the state machines that their traffic classification system requires. Thus, the authors propose to implement such state machines using the Zobrist hashing algorithm. Basically, this reduces memory requirements of state machines, which enables their allocation in cached memory. The throughput achieves a rate of 6.7 Gb/s with real traces (packet sizes of approximately 500 bytes) while the system may currently cost $ 2,500 in total. Again, this example shows that adapting applications to the capacities of novel hardware, in this case GPGUs, is an essential step in obtaining the best performance.

The authors in [SdRRG+12] present a software-based statistical traffic classification engine that exploits the size of the first few packets of every observed flow. The application uses PacketShader as packet engine. Unlike the previously explained application proposed in [SGV+10], this classification engine is not based on the utilization of GPGUs, but runs only on commodity multi-core hardware. In addition to the use of PacketShader as capture engine and the lightweight statistical technique as classifier, the remarkable classification rates achieved are made possible by a careful tuning of critical parameters of both the hardware environment and the software application itself. In particular, the proposed system properly sets memory and CPU affinity of different threads composing it, processes packets in a batch-oriented fashion (replicating batch processing ideas from PacketShader at user level), reuses memory structures for flow storing, exploits multi-core parallelism overlapping the different tasks (namely, reception, flow-handling and classification) while asynchronously communicating them by means of intermediate buffers (chunk and job rings). The system, currently valued at $ 6,000, achieves wire-speed classification in the worst-case scenario of 64 byte packets (10 Gb/s and even reaches 20 Gb/s when using two 10 GbE NICs and real traces (average packet size about 750 bytes).

Other monitoring tasks

The authors in [MSdRR+14b] dig into the most intuitive service to build over a packet capture engine, packet storage in non-volatile drives. They present an application named `hpcapdd`, that running on the HPCAP engine that is able to store packets in commodity hard-drives at 10 Gb/s rates for all packet sizes. The testbed value is about $ 3,000, to which should be added 12 hard drives with a total cost of about $ 2,500 as of today. The contributions of this application are to exploit affinity by automatically executing threads in the same NUMA node, accessing hard drives to store packets as a stream of consecutive bytes instead of following a packet-by-packet fashion and using a huge intermediate buffer to handle the irregular throughput of mechanical hard drives.

ffProbe [DDDS11] is an implementation of a NetFlow probe [Sys12], i.e. a probe that constructs a flow register for any consecutive set of packets sharing header information, e.g. same IP addresses, port numbers and layer-4 protocol. These registers typically comprise the number of packets and bytes. NetFlow has become a fundamental tool for any network manager. ffProbe runs over the PF_RING DNA engine and has proven to sustain 10 Gb/s rates with packets of about 500 bytes and rates over 7 Gb/s in the worst-case scenario of minimal packet size. The server and total hardware used on the performance evaluation should value roughly $ 3,000 as of today. The keys of the implementation of ffProbe is to leverage the concurrence capacity that PF_RING DNA provides by applying a hash function to packet headers and forwarding each of them to different concurrent processes. Note that this is possible as all packets comprising a flow share header fields used to construct flows. Additionally, ffProbe divides the work of constructing flows, looking for expired ones and the work of exporting to different processes. Due to the use of prefetching flow and memory caches the total latency is reduced.

The authors in [GDSdRR+14] focus their attention on how HPCAP engine may turn out useful to monitor VoIP networks. They proposed the system, VoIP-CallMon, which is able to track IP calls at aggregate rates of 10 Gb/s assuming codec G.711 with a packet size of 200 bytes. The testbed server costs would be around $ 3,000. The heart of this proposal is dealing with the high rates served by bottom layers as well as the fast construction of flow records. To this end, the authors designed several tailored data-structures so that after any flow insertion and exportation, the active-flow list remained sorted.

Finally, we mention Blockmon [DdH+12, DPHB+13a] which is neither an application nor a system by itself but a framework to implement monitoring applications. For example, reading packets, applying a level-4 filter and exporting a NetFlow registers. The authors leveraged PFQ as capture engine, and provided several sample applications running on it. Specifically, heavy hitter statis-

tics (flows whose number of packets or bytes are over given thresholds), SYN flooding detection and VoIP anomaly detection. As of today, the server used for testing has an estimated price of below $ 2,000. The authors do not provide further details of the implementations but report rates ranging between 3.8 and 10 Gb/s. These figures represent a 15% cut in performance compared to PFQ's capture process.

Importantly, this whole section highlights that, despite these novel packets engines' set-ups, the bottleneck may have risen through the network stack. This result alerts us that some old implementations of popular networking applications have been overtaken by the novel capture engines, which call for a review and subsequent optimization of the software.

3.6 Conclusions

The use of commodity hardware on high-performance network tasks has opened an exciting scenario where even the hardest task can be carried out by a flexible, extensible, adaptable and even inexpensive system. Examples of these tasks that have been enriched by this novel paradigm are applications such as software routers, anomaly and intrusion detection, traffic classification, and VoIP monitoring.

Unfortunately, the process of developing a high-performance networking task on commodity hardware from scratch may turn out to be a non-trivial process composed of a set of thorny sub-tasks, each of which presents fine-tuned configuration details. In this light, this work has aimed at providing practitioners and researchers with a road-map to the exploration of this useful paradigm.

Such road-map can be summarized by means of the following lessons learned and pieces of advice sprinkled throughout this chapter:

- Both default NIC drivers and network stack are shown to be insufficient to provide application layer with packets at multi-Gb/s rates. Depending on the scenario, bounds can be as low as 1 Gb/s.
- We have reviewed the main driver and stack limitations, and explained their respective countermeasures a capture engine should follow:
    - Dramatic cost of performing any operation at packet level:
      → Preallocating at driver load-time and reuse of memory during execution time.
      → Increasing packet data's access time by prefetching its contents while predecessor packets are still being processed.
      → Carrying out any task over a group of packets, not one-by-one, when possible.
    - Serialized access to traffic:
      → Exploiting parallel capacities of modern NICs by assigning a core to each RSS queue.
    - Multiple data copies from the wire to the user-level:
      → Mapping memory kernel regions at user-level.
    - Random placement of threads (or processes) across the available processors, leading to higher memory access latencies and cache inefficiency:
      → Carefully planning the thread-processor pair —threads must allocate memory in a chunk assigned to the NUMA node on top of which it is being executed.
      → Leveraging CPU and Interrupt affinity by setting both capture and

interrupt threads to the same processor —thus exploiting cached data and load distribution.

- Heavy kernel-to-userspace context switches:
  → Accessing as many packets as possible in a single system call: batches, streams.

- The industry and academia have applied most of these solutions giving rise to different and prominent off-the-shelf capture engines:
  - PF_RING DNA, PacketShader, netmap, PFQ, Intel DPDK and HPCAP.

- Practitioners and researchers interested in running applications over commodity hardware may base their development on one of these engines and skip most of the low-level details. They may make a decision based on both the additional features and the performance level offered by each engine.
  In terms of features we have identified:
  - Different APIs, some of them are similar to *de facto* standard, libpcap, and socket-alike.
  - Different level of timestamping precision and concurrent application support.
  - Different levels of compatibility with 1Gb NICs or NICs different from the Intel's 82598/82599 —which has become the *de facto* reference for all approaches.

  In terms of performance:
  - We have assessed that all engines surpass default NIC drivers and networking stacks' performance by far.
  - A certain scenario's defiance depends on both the available machine's topology and the traffic intensity in terms of Mp/s —essentially, the smallest packets are the most challenging ones.

  As take-away messages:
  - We have found that PFQ spans several advantages such as flexibility and ease of use.
  - PF_RING DNA, PacketShader, and netmap achieve full rates regardless of packet size and low resource utilization even with only one queue.
  - HPCAP provides accurate timestamping and enhanced packet storage.
  - Intel DPDK stands out given its extensive compatibility with NICs from several manufacturers.

- We remark the success of the explained capture engines by means of real-world applications, and state their significant state-of-the-art bounds. To illustrate this, we highlight that contemporary software routers based on commodity achieve rates above 30 Gb/s in tasks such as IP forwarding and OpenFlow switching. Moreover, flow construction, VoIP monitoring and packet storage applications achieve rates of 10 Gb/s, and even DPI and Snort-based applications can operate at rates higher than 5 Gb/s.

To conclude, we believe these lessons learned and pieces of advice may serve as a catalyst for the arrival of new high-performance network applications based on the pair of commodity-hardware and open software. We hope this also serves to stimulate further ideas and proposals to face the near and demanding future, paving the way for 40 Gb/s or even 100 Gb/s interfaces.

4

HPCAP IMPLEMENTATION DETAILS AND FEATURES

This chapter describes the developement process of HPCAP. Specifically, it focuses on the key aspects that guided and motivated HPCAP's design as a new high-performance packet capture engine. Those aspects are manifold: accurately timestamping the incoming traffic, optimizing the packet storage proccess, and the ability of removing duplicated packets that may consume unnecessary resources.

In this chapter we describe the implementation details of HPCAP: our packet capture proposal. HPCAP is designed with the aim of meeting a set of features that, according to our experience, an ideal packet capture engine should meet, namely:

- it should achieve maximum packet capture performance using just one RX queue in order to avoid collateral effects,
- it should associate incoming packets with an as accurate as possible timestamp,
- it should be oriented to optimize packet storage into non-volatile volumes,
- it should minimize the employment of computational resources with irrelevant traffic. Specifically, our first proposal is to detect and reject duplicated traffic.

Importantly, those features HPCAP has been focused on were ignored by other proposals. The rest of this chapter begins with a description of the design of HPCAP, followed by an extensive evaluation of each of its previously mentioned distinguishing features.

4.1 HPCAP's design

Even though 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. Although the use of multiple queues may entail collateral issues such as packet reordering or timestamping drifts, if the traffic distributed amongst different queues is independent from the rest, supporting multiple queues becomes a valuable feature. Furthermore, ir order to make HPCAP as general-purpose as possible, the possibility of instantiating more than one application feeding from the same RX queue has been supported. HPCAP's architecture is depicted in Fig. 4.1. In this figure, three main blocks can be distinguished:

- 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.

Importantly, although the development of HPCAP has been tied to the Intel's 82599 NIC, the source code has been structured so that the NIC-independent code is separated in different source code files, and the NIC-dependent code has been explicitly labelled. That is, HPCAP has been developed so that it could be easily extended to support new network interface cards and vendors.

4.1.1 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. 4.1).

Other capture engines, such as PacketShader [HJPM10] or Netmap [Riz12c], copy incoming packets 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

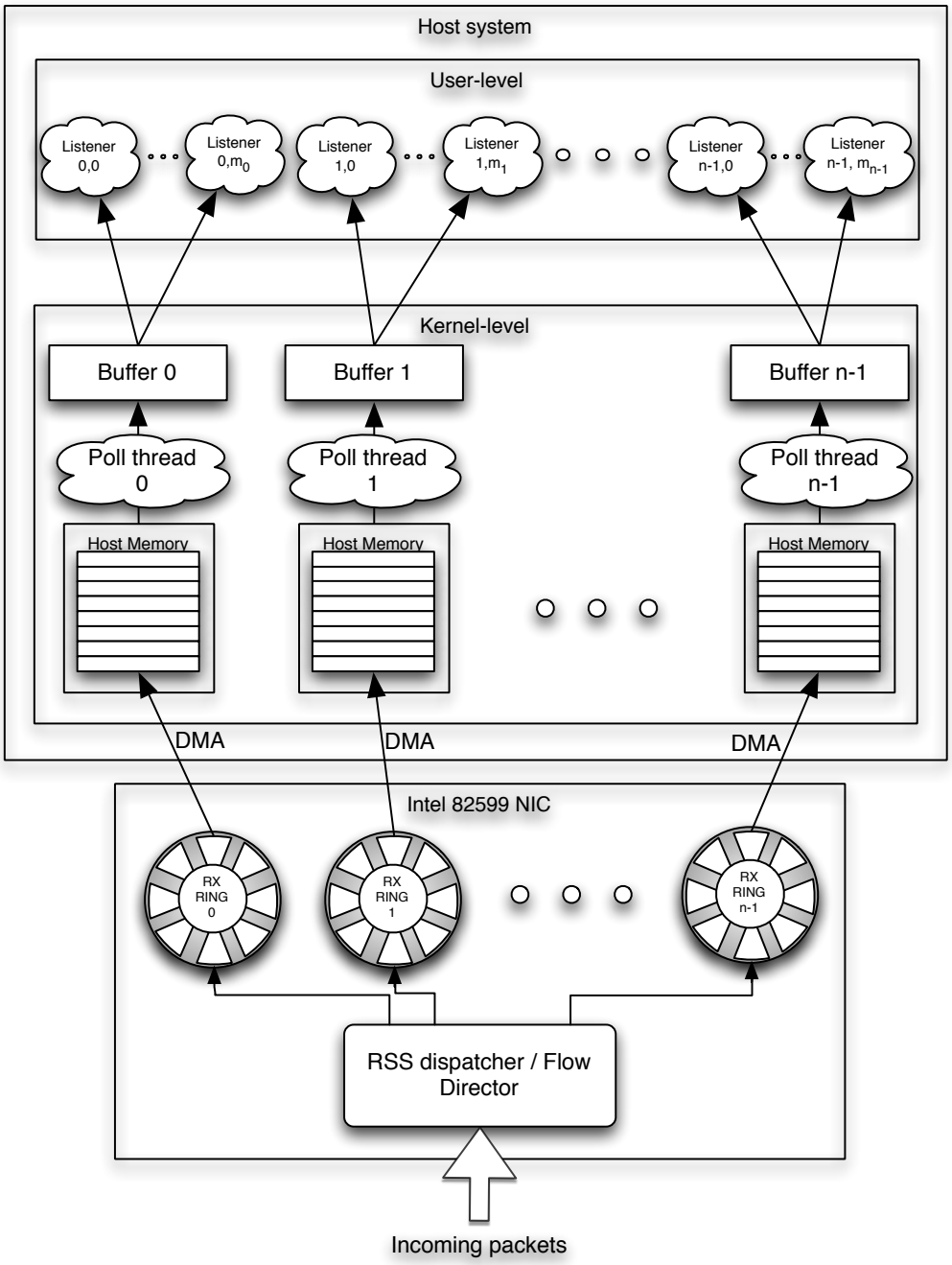


Figure 4.1: HPCAP kernel packet buffer

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. Furthermore, 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 section 4.2.

In order to overcome such problems, HPCAP instantiates a KPT (Kernel-level Polling Thread) per each RX queue. Those threads are 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 obtain the system time by means of the Linux kernel `getnstimeofday()` function.

All the incoming packets are copied into the buffer in a raw data format (see Fig. 4.2): 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 and captured length (as two 16-bit fields), and in the last place the packet data (with a variable length).

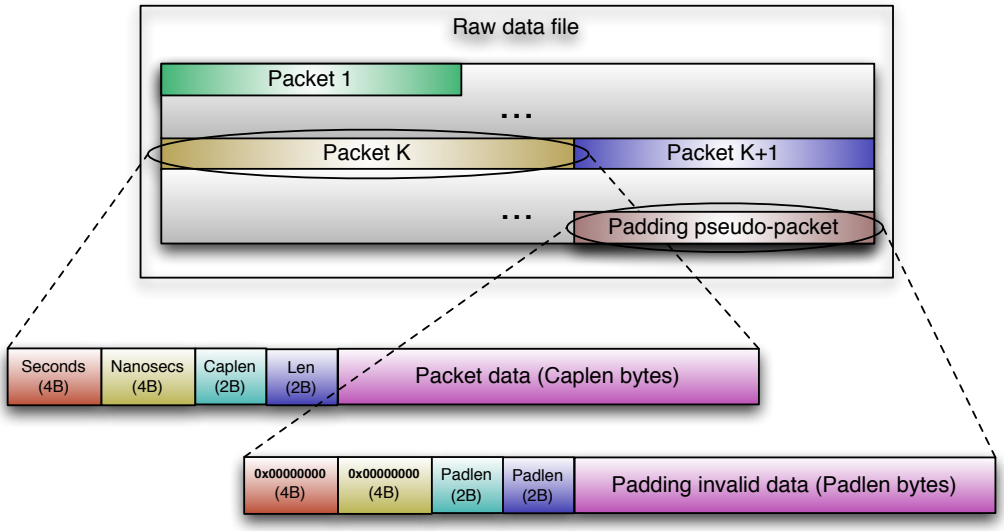


Figure 4.2: HPCAP kernel packet buffer

This raw buffer format gives a higher level of abstraction than the typical packet structure used by PCAP-lib or other approaches, so upper level applications can efficiently access the data in both a packet oriented or a byte-stream

basis. In order to make sure the data files are not cut in the middle of the packet stream, the last packet of each file (whose size was fixed at HPCAP's load time) is a false packet or padding packet, whose header contains a zero-valued timestamp and the size of the padded data. This byte-stream oriented access to the network data allows upper-level applications to efficiently store the captured data, by carrying out block-level operations over the storage device and thus maximizing performance.

4.1.2 Multiple listeners

As shown in Fig. 4.1, HPCAP supports multiple applications, referred as *listeners*, to fetch packets from the same RX queue in a SPMC (Single Producer, Multiple Consumer) model. 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 read 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.
- an **available-bytes 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 [Lov02]. 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.

4.1.3 User-level API

The *everything is a file* Linux's philosophy, provides a simple way to communicate user-level applications with the HPCAP driver. Following that philosophy, HPCAP instantiates a different character device [CRKH05] node in the `/dev/` filesystem for each of 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_0_0
/dev/hpcap_0_1
        ...
/dev/hpcap_0_<M-1>
/dev/hpcap_1_0
/dev/hpcap_1_1
        ...
/dev/hpcap_1_<M-1>
...
/dev/hpcap_<N-1>_0
/dev/hpcap_<N-1>_1
        ...
/dev/hpcap_<N-1>_<M-1>
...
```

Code 4.1: 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, two approaches can be used by those user level applications to process the traffic that is stored inside the driver's buffers:

- it can read the incoming packets by performing a standard `read()` call over the previously opened file. This way, the data will be copied from the driver's buffers to the system's internal buffers. Although this additional

copy imposes an overhead over the application's throughput, the raw data format inside the buffers allows to directly perform a `write()` system call targeted to a storage device.

- the user level application can directly map the kernel buffers' memory via a set of library functions that make use of the standard `mmap()` system call. Applications using the memory mapping techniques must use a set of library functions that provides custom commands via the `ioctl()` call to synchronize with the driver. This mechanism allows user applications to access the captured data without additional copies and consequently improve overall performance. Additionally, the use of the memory mapping mechanism allow several applications to access a single buffer's data without additional copies.

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.

4.1.4 HPCAP packet reception scheme

Fig. 4.3 shows HPCAP's packet reception scheme, in contrast with those representing the packet capture engine of a traditional NIC and other high-performance packet capture engines already shown in Chapter 3.

Similarly to other packet capture engines, 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 kernel poll thread will constantly copy (and timestamp) the available packets into its corresponding buffer. Note that, differently from other packet capture engines, the packet copies are not made when a user application asks for more packets, but always there are available packets.

Importantly, the buffer-oriented approach followed by HPCAP isolates the process made over the incoming packets at kernel level from the process carried out by upper-level applications. If properly managed, this isolation allows to create a pipelined process chain that increments the total amount of computation that can be carried out over each incoming packet. This increments is because the time constraints existing for each packet's computation will apply to the slowest stage in the computational pipeline.

Note that, just as mentioned in previous chapters, 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 [1]. Multithreaded applications can make use of the `pthread` (POSIX Threads) library in order to map each thread to its corresponding core [2].

[1] http://linux.die.net/man/1/taskset

[2] http://man7.org/linux/man-pages/man3/pthread_setaffinity_np.3.html

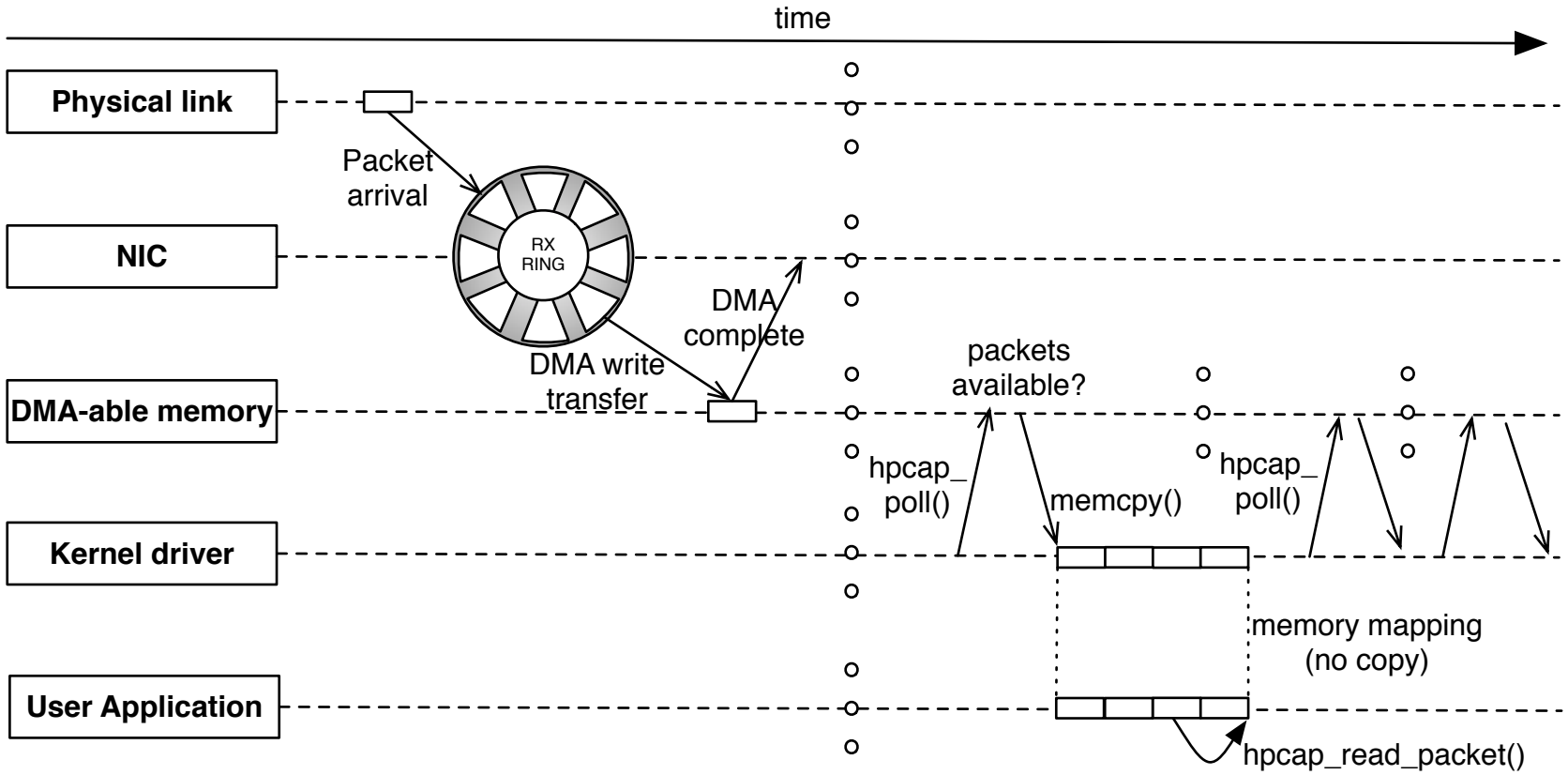


Figure 4.3: HPCAP packet reception scheme

4.2 Packet timestamping

Novel packet I/O engines allow capturing traffic at multi-10Gb/s using only-software and commodity-hardware systems. This is achieved thanks to the application of optimization techniques such as batch processing. Nevertheless, no attention has been paid on the fact of timestamping those packets captured at high-rates, and even more in the quality that a potential timestamping policy developed as an extension of those packet capture engines would experience. We want to emphasize the relevance of this fact, as packet timestamping has proven vital when carrying out analysis tasks over network traffic [MSD+08]. Importantly, some of the techniques and inherent packet capture architectures presented by most packet capture engine involve degradation in the timestamp accuracy. This section pretends to evidence, via empirical results, the timestamping accuracy problem and present a discussion on the impact of packet timestamping over the final system's packet capture performance.

4.2.1 Accuracy issues

Dealing with high-speed networks claims for advanced timing mechanisms. For instance, at 10 Gb/s a 60-byte sized packet is transferred in 67.2 ns (see Eq. 4.1 and 4.2), 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.

$$TX_{time} = \frac{(Preamble + Packet_{size} + CRC + Inter\_Frame\_Gap)}{Link_{rate}} \tag{4.1}$$

$$TX_{time} = \frac{(8 + 60 + 4 + 12)\frac{bytes}{packet} \times 8\frac{bits}{byte}}{10^{10}\frac{bits}{second}} = 67.2 \times 10^{-9}\frac{seconds}{packet} \tag{4.2}$$

In an ideal scenario, the timestamp would be done by a hardware agent which is synchronized with a time source using a sub-microsecond synchronization protocol [LCSR11]. Although a hardware timestamping policy would offer best accuracy and minimal performance interference, such as in [MGGA+11], this feature is not implemented by commodity NICs and would thus entail changing to non-standard solutions. Importantly, some NICs such as Intel's 82599 offer features that allow defining a set of rules to identify a subset of packet that

would be timestamped with a PTP (Precision Time Protocol)-synchronized internal clock (see Section 8.2.3.26 in [MGGA+11]). However, this only allows to timestamp a reduced set of the incoming packets as the timestamp for a packet matching the filter requirements is stored in an internal register until released. For this reason, hardware timestamping is not a realistic option in commodity NICs yet.

Conversely, if the incoming traffic is to be timestamped by the capture software, regardless this software is executed at driver or user level, it will suffer from timestamp inaccuracy due to time synchronization and kernel scheduling policies [BRE+15]. 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 work as it has already been treated by methods like NTP (Network Time Protocol), LinuxPPS or PTP [LCSR11]. Via a fine tune on the capture processes schedule priorities this effect could be slightly mitigated. Only a real-time operating system [Bar97, BY96] could grant a constrained schedule time. However, although very interesting, those issues are out of the scope of this work. We will focus on the impact of the packet timestamping policy on the timestamp accuracy, as we consider this fact have a heavier impact than the previously mentioned low-level operating-system-related details.

The timestamp accuracy problem becomes more dramatic when some of the reception steps taken before a packet is timestamped is carried out in a batch-oriented fashion, which is not uncommon as explained in Chapter 3.3. In that case, even though the incoming packets were 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. 4.4 this effect is exposed for a 100%-loaded 10 Gb/s link in which 60-byte packets are being received using PacketShader [HJPM10], 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.

Figure 4.5 shows the standard deviation of the observed timestamp error after receiving 1514-byte sized packets for one second at maximum rate. The timestamp accuracy is degraded with batch size. Note that when the batch size is beyond 16 packets, the error tends to stall because the effective batch size remains almost constant —although a given batch size is requested to the driver, user applications will be only provided with the minimum between the batch size and the number of available packets.

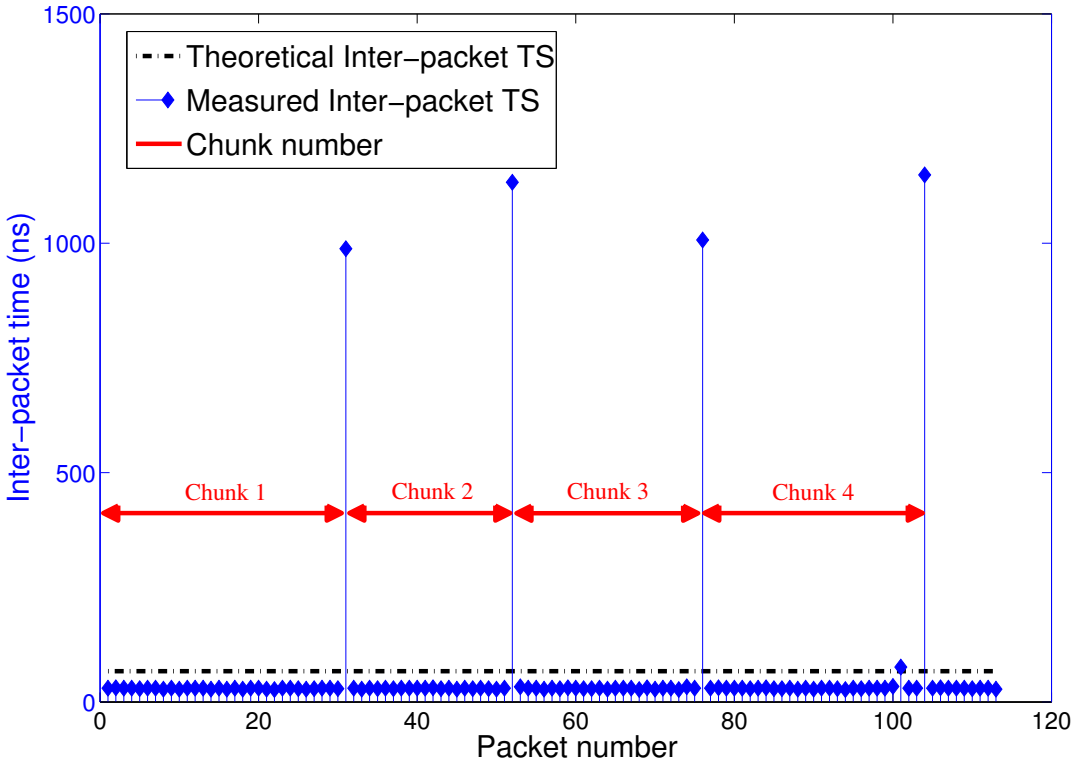


Figure 4.4: Batch timestamping effect on inter-arrival times

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 [WDC11]. 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 [BRV09]. As a consequence, a capture engine using more than one reception queue would suffer from those inaccuracy sources.

Timestamping policies and accuracy

To overcome the problem of batch timestamping, we proposed 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.

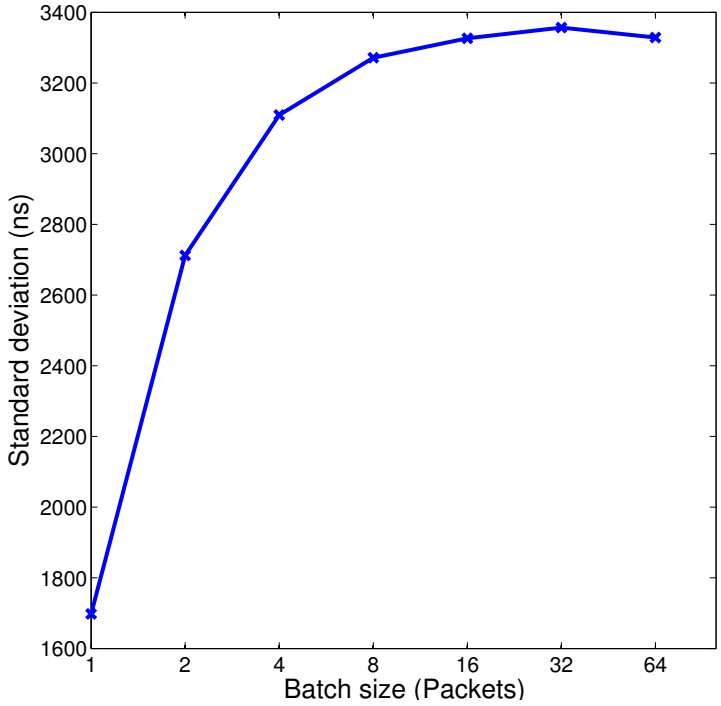


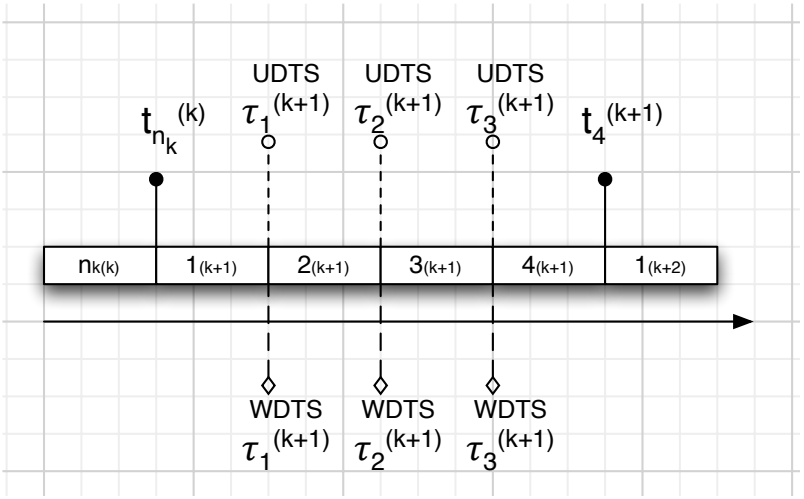
Figure 4.5: Degradation on timestamping accuracy with batch size

UDTS: Uniform Distribution of TimeStamp

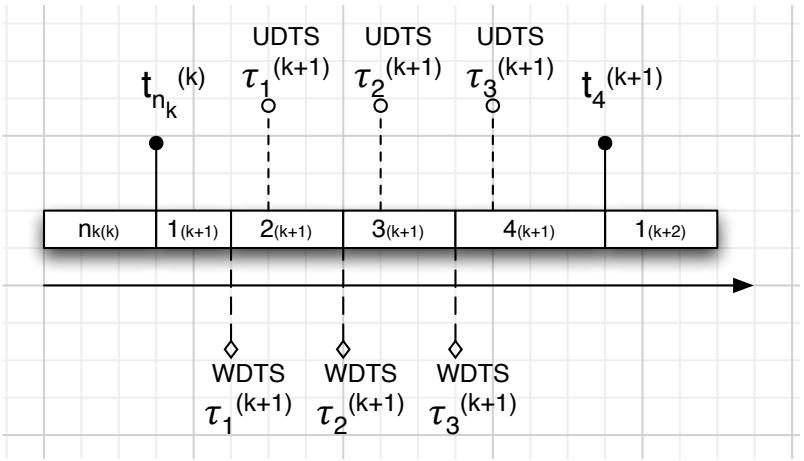
The simplest technique to reduce the huge time gap between batches is to uniformly distribute inter-batch time among the packets of a batch. Equation 4.3 shows the timestamp estimation of the i-th packet in the $(k+1)$-th batch, where $t_m^{(j)}$ is the timestamp of the m-th packet in the j-th batch and n_j is the number of packets in batch j.

$$\tau_i^{(k+1)} = t_{n_k}^{(k)} + \left(t_{n_{k+1}}^{(k+1)} - t_{n_k}^{(k)}\right) \cdot \frac{i}{n_{k+1}} \; \forall i \in \{1, \ldots, n_{k+1}\} \tag{4.3}$$

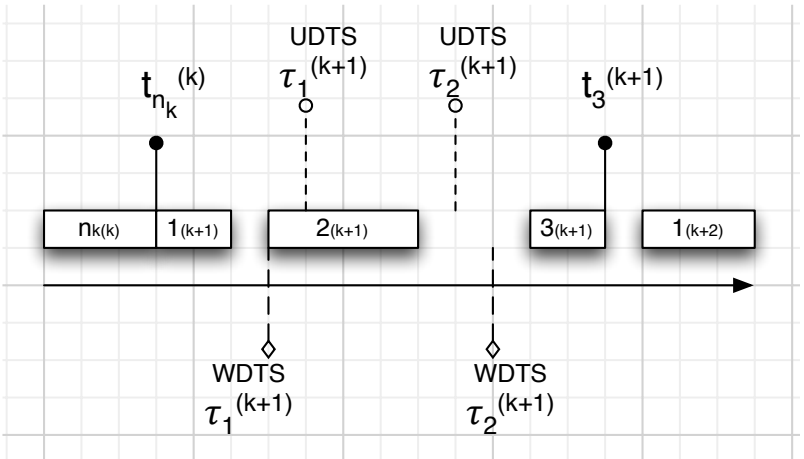
As shown in Fig. 4.6(a), this algorithm would perform correctly when all the incoming packets of a given batch have the same size. The drawback of this solution is that all packets of such batch would show have the same inter-arrival times regardless their size. This would lead to the appearance of timestamping errors in realistic scenarios, even though they are fully-saturated, as shown depicted in Fig. 4.6(b). Note that the inter-packet gap is proportional to the packet size when transmitting packets at maximum rate.



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



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



(c) Non fully-saturated link with variable packet size.

Figure 4.6: Inter-packet gap distribution

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 4.4 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.

$$\tau_i^{(k+1)} = t_{n_k}^{(k)} + \left(t_{n_{k+1}}^{(k+1)} - t_{n_k}^{(k)}\right) \cdot \frac{\sum_{j=1}^{i} s_j^{(k+1)}}{\sum_{j=1}^{n_{k+1}} s_j^{(k+1)}} \; \forall i \in \{1, \ldots, n_{k+1}\} \tag{4.4}$$

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. 4.6b. 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. 4.6c). That is, the lower the inter-packet gap is, the higher the accuracy is.

KPT: Kernel-level Polling Thread

Towards a timestamping approach that performs properly regardless the link load, we proposed a redesign of the network driver architecture. Novel packet capture engines fetch packets 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, HPCAP implements 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 using HPCAP for packet capture 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 active waiting approach. However, subsection 4.2.2 will show that it is not the active wait what may cause packet capture performance degradation, but the system call required to obtain the system's timestamp.

Our setup consists of two servers (one for traffic generation and the other for receiving traffic) directly connected through a 10 Gb/s 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 [Glo]. 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 Gb/s, i.e., 812,744 packets per second and measure the interarrival times in the receiver side. Table 4.1 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 $\sim$600 ns. Despite timestamping packet-by-packet, PFQ shows a timestamp standard error greater than 13 μs.

Solution	**Batch size**	**Error**	
		$\bar{\mu}$**[ns]**	$\bar{\sigma}$ **[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 4.1: 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 [WAcAa]). We perform two

experiments: in the first one, the trace is replayed at wire speed (that is, at 10 Gb/s), and then, we replay the trace at the original speed (i.e., at 564 Mb/s, respecting inter-packet gaps). Due to storage limitations in the FPGA sender, we are able to send only the first 5,500 packets of the trace. Table 4.2 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	**Error**			
	Wire-Speed		**Original Speed**	
	$\bar{\mu}$**[ns]**	$\bar{\sigma}$ **[ns]**	$\bar{\mu}$**[ns]**	$\bar{\sigma}$ **[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 4.2: Experimental timestamp error (mean and standard deviation). Real traffic: Wire-speed and Original speed

4.2.2 Performance evaluation

As already mentioned, HPCAP implements a KPT policy for processing the incoming packets. This way, each incoming packet is processed independently from the rest, which has positive effects on its timestamp accuracy, but may imply a performance degradation in terms of packet capture performance. Importantly, Eq. 4.2 stated the time available for processing each incoming packet in the worst-case scenario for a 10GbE link, which is 67.2 ns. This fixes a time bound for the things that can be done per incoming packet if a zero-loss rate is to be reached. Precisely, this temporal constraint is what inspired the buffer-oriented architecture of HPCAP, as it allows to pipeline the process carried out over each packet, so that part of the process is carried out at kernel level and the rest is made at user-level.

Consequently, the amount of work made at driver level must be minimized in order to sustain maximum packet processing throughput, but there are always two tasks that can not be moved to upper layers:

- **Copying the packet's data:** in order to allow the memory area assigned to the incoming packet's receive descriptor to be re-used for future incoming packets, this memory must be released as soon as possible. This implies copying the packet's data into the previously mentioned packet buffer.
- **Timestamping the packet:** as there are no guarantees about when the packet will be processed by upper layers, its timestamp must be assigned as near to its physical arrival to the NIC as possible. This implies a system call to the `getnstimeofday()` function for each incoming packet. Importantly, there are lighter functions that provide a local-CPU time and have minimal overhead, but they do not guarantee inter-CPU synchronization nor monotonically increasing time stamps so their usage has been discarded for our purposes.

At this point, we will evaluate the packet capture performance of HPCAP using the same test framework as the one presented in Section 3.4.1 when comparing the performance of the diverse packet capture engines. Furthermore, the impact over capture performance of timestamping every incoming packet will be evidenced by comparing the default behaviour—i.e., timestamping every packet— with the performance obtained when timestamping one out of two, four or eight packets, and the performance obtained when no packet is timestamped at all.

The left-hand side part of Fig. 4.7 shows the packet capture performance when processing 14.88 Mpps with a constant size of 60 bytes (CRC not included) when varying the amount of receive queues used. As it could be expected, the amount of packets captured rises from 69.37%, when all packets are timestamped, up to 82.85%, when no packet is timestamped at all, with the amount of packets captured increasing when then amount of packets timestamped decreases. When increasing the amount of queues in use, the impact on the amount of packets timestamped of performance becomes more irrelevant, and thus the performance losses experienced are due to a reduction in the amount of descriptors available for each queue or the overall CPU consumption when the amount of queues is high, rather than then in the time available for processing each packet. When the packet size is increased to an average size, the processing time is big enough so no packet losses are experienced in any case, as shown in the right half of Fig. 4.7.

On the other hand, subfigures in Fig. 4.8 show the packet capture performance for each of the timestamping policies mentioned when capturing from a fully-saturated 10GbE link with different constant-sized packets. Results show a performance improvement in the worst-scenarios when reducing the amount of packets timestamped, but when the packet size is greater or equal to 64 bytes (CRC not included) then all packets are captured if one queue is used. When the number of queues is greater, maximum performance is reached when the

packet size is greater or equal to 128 bytes.

The results shown here prove the existence of an impact over packet capture performance on the timestamping policy implemented by HPCAP. This performance results in conjunction with the accuracy results exposed in the previous subsection should give a potential user of a high-performance packet capture engine an idea about which engine would be more convenient according to his performance-accuracy constraints. In line with the future work previously mentioned, we propose the use of a dynamic policy capable of deciding the amount of packets to be timestamped depending on the network load (and thus minimize potential packet loses), and then applying a WDTS algorithm to re-construct the timestamp of those packet which where not assigned a temporal mark.

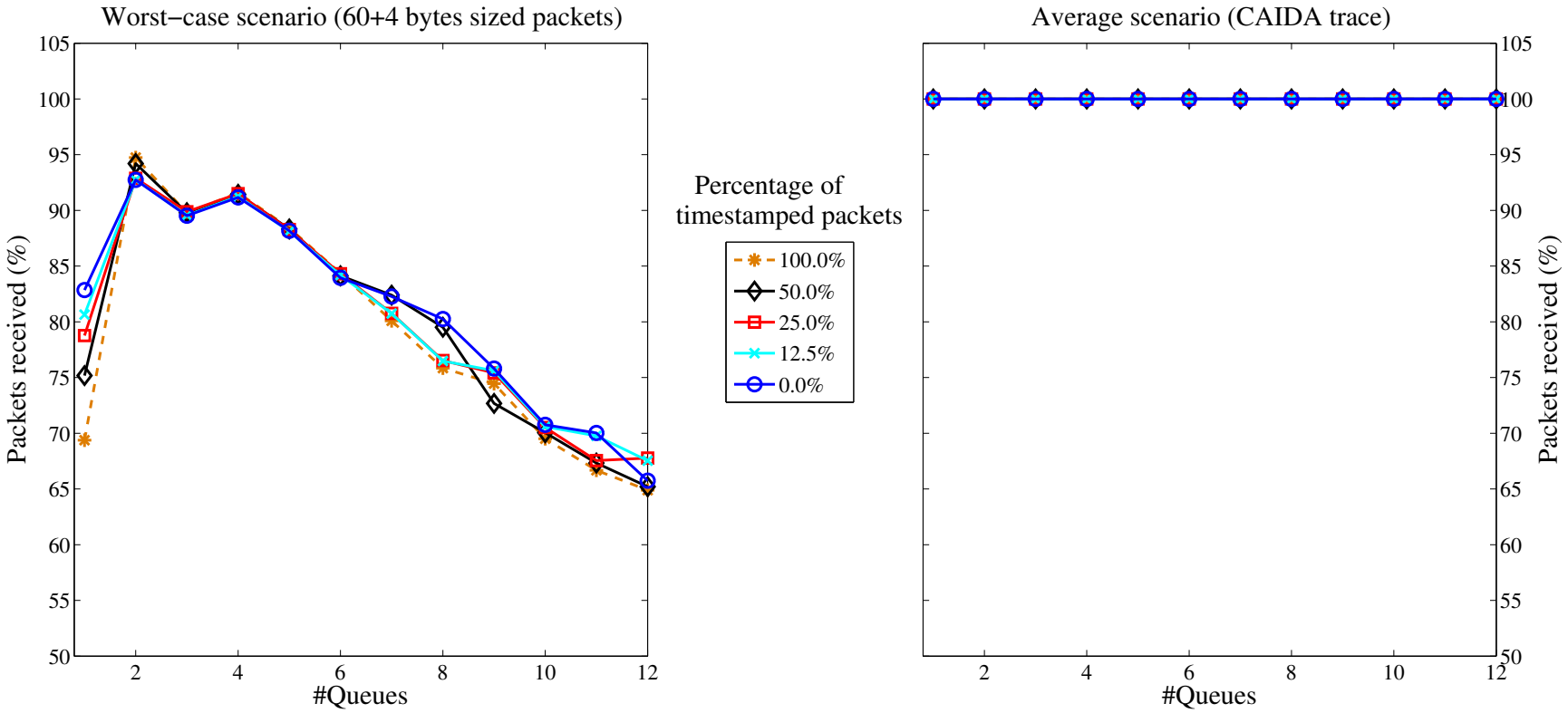
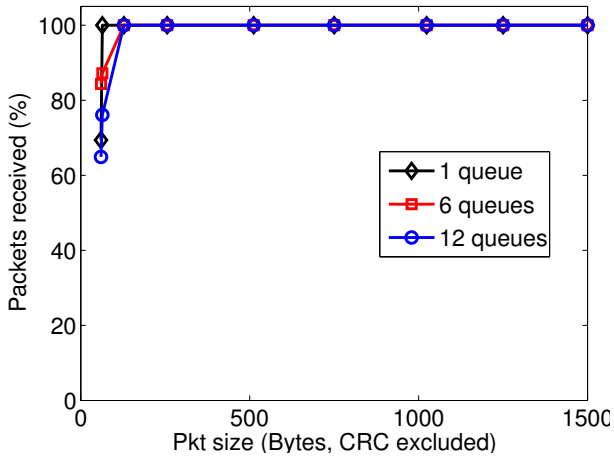
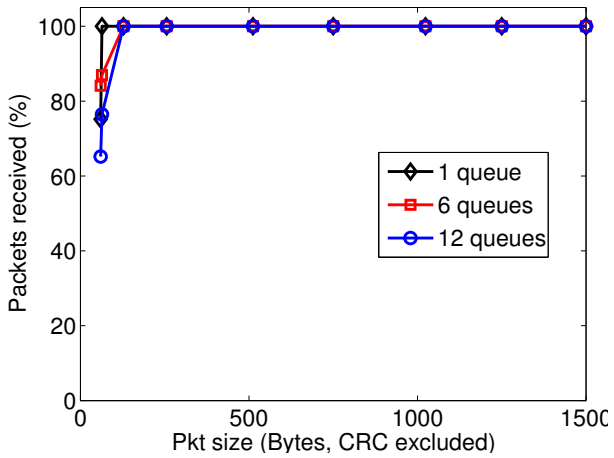


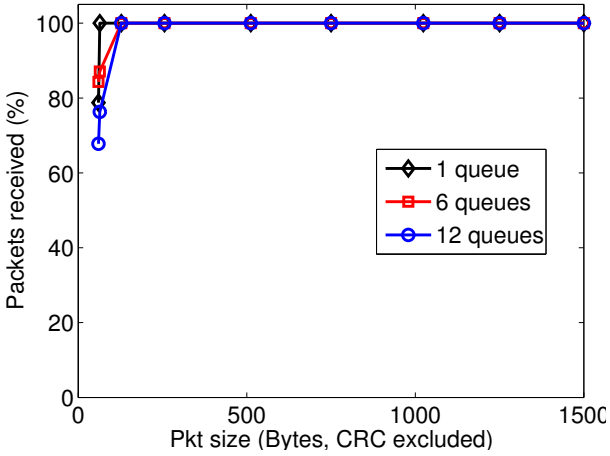
Figure 4.7: Packet capture performance for different timestamping policies in a worst-case and average scenarios when varying the number of receive queues



(a) All packets are timestamped

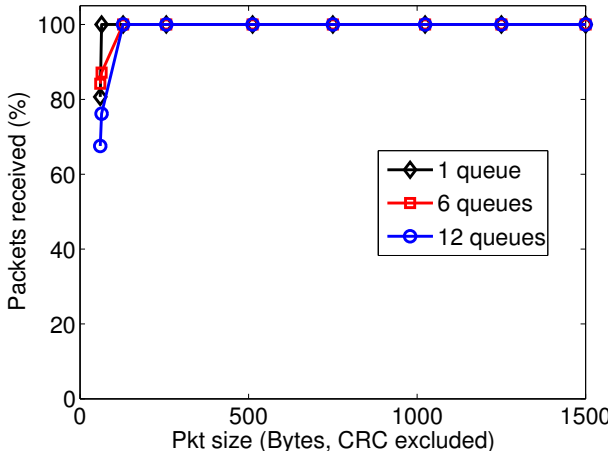


(b) Half of the packets are timestamped

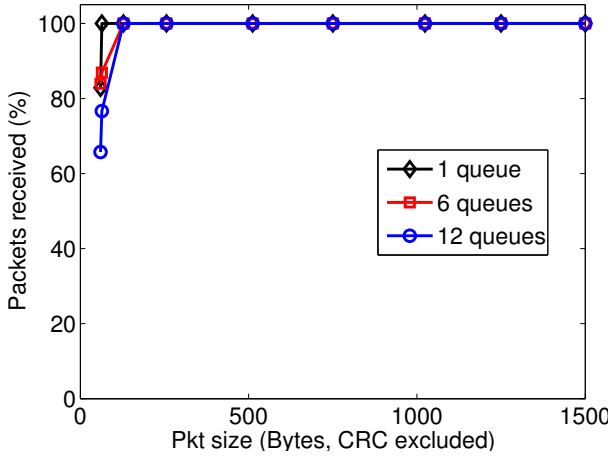


(c) One out every four packets are timestamped

Figure 4.8: $(a), (b), (c)$ Effect of packet timestamping on performance for variable constant-sized packets in a full-saturated 10 Gb/s link when varying the number of receive queues



(d) One out every eight packets are timestamped



(e) No packets are timestamped

Figure 4.8: $(d), (e)$ Effect of packet timestamping on performance for variable constant-sized packets in a full-saturated 10 Gb/s link when varying the number of receive queues

4.3 Packet storage

Nowadays, the increase in users' bandwidth demand caused by the deployment of new services and technologies has become of paramount importance for Internet Service Providers. Similarly, fashionable multimedia services such as VoIP or Video on Demand (VoD) call for the tightest SLA (Service Level Agreement) that take into account not only bandwidth but also other QoS parameters. The cherry on this cake is that the wide range of network operators available has placed Internet customers in a strong position in the telecommunication market, thus leading operators to enter into strong competition. In such a demanding scenario, operators are deploying novel network architectures and equipment with bandwidth capabilities of multi-gigabit rates and beyond. Testing the performance and correct operation of such deployments is a challenging task that operators must face in order to minimize unexpected problems when production traffic is placed on their infrastructure. This problem applies not only to operators but also to other players in the Internet arena, such as third-party enterprises and banks that deploy new services for their customers and employees.

The most simple and efficient way of testing such infrastructures and services is sniffing and storing all the traversing test traffic for its subsequent analysis [MSD+08]. Such an analysis may focus not only on searching malformed or unexpected packets (e.g., erroneous VLAN or MPLS headers, or duplicated frames) but also on the network performance and QoS parameters' value — bandwidth, packet loss, delay or jitter. Additionally, stored traffic may be used not only passively but actively when replaying the content of the stored traces to test the performance and behavior of a deployed network [LLC+12]. We will refer to systems that sniff and store traffic as NTSS (Network Traffic Storage Solution).

Even an intuitively simple task such as sniffing and storing the traversing traffic is a challenge when dealing with 10 Gb/s speeds or higher due to the great amount of resources and computational power needed. Traditionally, specialized hardware devices such as FPGA-based solutions, network processors or high-end closed commercial solutions have been applied to tackle the traffic sniffing and storage problem. Although such solutions show remarkable levels of performance, they lack flexibility and extensibility in addition to their high expenditures [Nap10].

As an alternative, the research community has recently focused on off-the-shelf solutions to accomplish high-performance tasks [BDKC10]. Off-the-shelf systems have emerged as the combination of commodity hardware and open source software. Such systems provide flexibility, availability and scalability while handling multi-gigabit rates and cutting expenditures both in terms of deployment and maintenance [GDMR+13]. With this in mind, this study explains the

key aspects for off-the-shelf systems to sniff and store packets at multi-gigabit rates. The key aspects involved comprise fine low-level tuning at NIC driver, hard drives, RAID configuration, and final application level. Subsequently, we provide an extensive performance evaluation of state-of-the-art NTSS systems that have successfully reached such a goal. The performance evaluation carried out covers an extensive set of scenarios, including both the most demanding and realistic ones.

4.3.1 Motivation

Traffic storage has become a challenging task, as a full-saturated 10 GbE link in the worst-case scenario (minimal size packets, i.e., 64 Bytes on Ethernet with CRC included) carries more than 14 million packets per second. In this demanding scenario, we first must ask ourselves how much traffic may be sniffed in an out-of-the-box configuration—i.e., standard software running on a commodity server. Specifically, our commodity server is a Supermicro X9DR3-F commodity server and two 6-core Xeon E52630 processors running at 2.30 GHz and hyper-threading disabled, with 96 GB of DDR3 RAM at 1333 MHz. The server is equipped with an Intel 82599 10GbE NIC plugged into a PCIe 3.0 slot. On the other hand, the software side is composed by an Ubuntu Server 14.04 configured with a 3.14 kernel and the default network stack, the vanilla Intel `ixgbe` NIC driver, and the de-facto standard traffic sniffer `tcpdump`.

The results are shown on the leftmost 2-column group of Fig. 4.9, which show the percentage of stored packets with this configuration (named `tcpdump` vanilla). These columns show two traffic injection cases: namely, synthetic 64-byte packets with CRC included and a real backbone trace [WACAb], both replayed at wire-speed. We can observe that this out-of-the-box scenario is only able to sniff and store less than 10% and 38% out of the total sent packets, for synthetic and real traffic respectively. As a consequence, the out-of-the-box configuration has shown to be insufficient to capture full-rate 10 GbE traffic.

The research community's answer to those results has been first to improve the NIC vanilla driver to boost up its performance, as the following section explains. After traffic is sniffed, the storage process must be carried out. Unfortunately, there is scarce knowledge about writing at multi-gigabit rates in commodity hard drives. Consequently, this work studies how to tune such drives to increase their performance, a question which is dealt with in Section 4.3.2. Finally, Section 4.3.3 further explains how to make the most of the interaction of these two issues, and provides a performance evaluation of the state-of-the-art NTSS in 10 GbE networks.

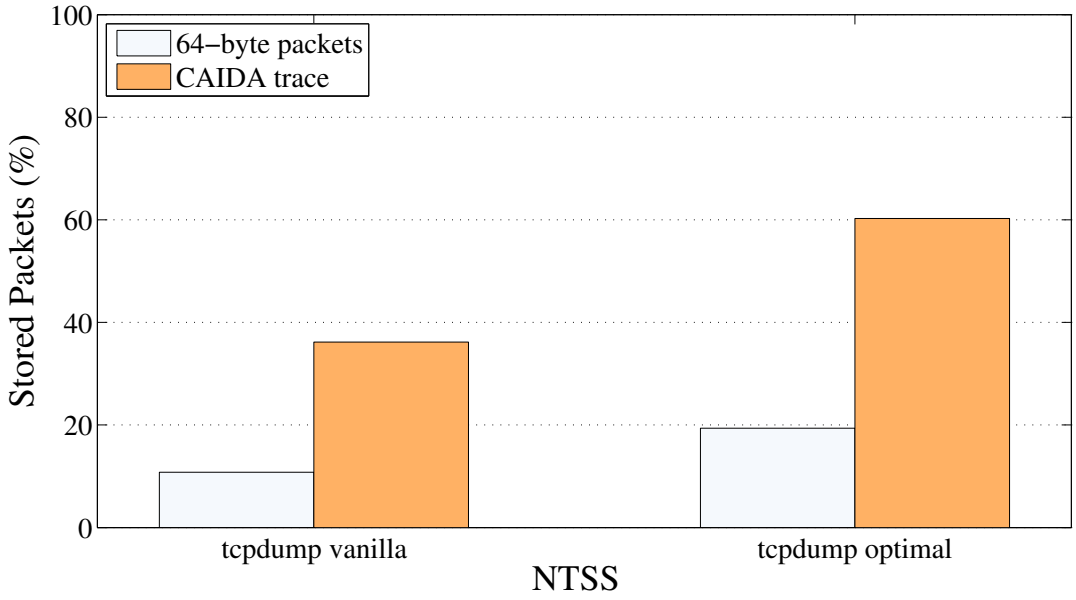


Figure 4.9: Percentage of stored packets (into a RAID-0 volume with 9 disks) for a full-saturated 10 GbE link for NTSS based on tcpdump for a 30-minute experiment

4.3.2 Storing data on hard-drives

Nowadays, a high-end SATA-3 mechanical disk allows theoretical rates up to 4.8 Gb/s for sequential reads and 1.2 Gb/s for sequential writes. A SATA-3 SSD (Solid-Sate Drive) may achieve speeds close to 3.2 Gb/s for both read and write operations, but its price per GB is 10 times greater. Consequently, no matter if the hard drive is SSD or conventional, a single disk is not enough to comply with the line rate of 10GbE networks and a RAID array is in order.

Chapter 3 took an in-depth look at the capacities of the modern sniffing engines. In terms of packet sniffing the worst case scenario is where packets are of minimal size and, therefore, the amount of packets per second to be processed is maximized. Specifically, 14.88 Mp/s is such a worst case in 10 GbE networks using 64 byte (CRC included) packets. However, we note that the worst case is the opposite for storing traffic, as shown in Table 4.3. It turns out that the disc load increases with packet size and, to complicate matters further, an additional header is aggregated to the packet with the timestamp, and both packet and capture lengths. For instance, de-facto standard PCAP, stores a 16-byte header per packet—4 bytes for caplen, 4 bytes for len and 8 bytes for timestamp. This header may be reduced using 16-bit instead of 32-bit fields for caplen and len as the maximum frame size in 10 GbE is 9216 bytes for jumbo frames. Table 4.3 shows the storage overhead using both approaches — PCAP and reduced PCAP labeled as RAW.

Paying attention to the last row of Table 4.3, we note that disk capacities have to be over 9 Gb/s with a worst case of 9.95 Gb/s assuming PCAP headers. It is worth remarking that even for the typical packet mean of Internet, i.e., ranging between 256 and 512 bytes according to CAIDA, the demand for store throughput is 9.71 and 9.85 Gb/s respectively. As a conclusion, once the sniffing engines proved capable of dealing fairly with all packet sizes, now the goal is for the RAID array to attain rates of fairly 10 Gb/s to cope with common realistic scenarios.

Max. throughput	**Packet size (bytes, CRC included)**								
	60	**64**	**128**	**256**	**512**	**750**	**1024**	**1250**	**1518**
Mp/s	14.88	14.21	8.22	4.46	2.33	1.62	1.19	0.98	0.81
Gb/s	7.14	7.27	8.38	9.13	9.55	9.69	9.77	9.81	9.84
Gb/s (PCAP header included)	9.05	9.09	9.46	9.71	9.85	9.90	9.92	9.94	9.95
Gb/s (RAW header included)	8.57	8.64	9.19	9.57	9.77	9.84	9.89	9.91	9.92

Table 4.3: Maximum throughput in terms of packets and bits for different packet and header sizes in a fully-saturated 10GbE link

Unfortunately, RAID configuration parameters are manifold and a wrong choice of values leads to severe performance degradation. Our tests have been carried out using the out-of-the-box Linux server described in Section 4.3.1. Specifically, we have conducted thorough testing with the following configuration parameters:

- *Disk technology*: We have evaluated the actual write throughput of a RAID-0 volume composed of both high-end mechanical and solid-state drives. The mechanical disks used are Hitachi HUA723030ALA640 and 3 TB of capacity, whereas the solid-state drives are Samsung SSD 840 EVO with 250 GB of capacity, both of them with SATA-3 interface.
- *Number of disks*: We have assessed how performance differs with a varying number of disks from 1 to 12 in RAID-0 (we used an Intel RS25DB080 RAID controller).
- *Strip size*: The strip size is the amount of data per basic write operation. Thus, small strip sizes will be translated into a higher number of write operations into the RAID volume and may degrade the overall write throughput due to per-operation overheads. We have tested strip sizes of 64 KB, 256 KB and 1MB.
- *RAID write cache policy*: this parameter refers to the use of the RAID controller's cache memory. The *D*irect policy disables the cache and performs poorly. The WTC (Write-Through Cache) policy writes the cache content to disk, and then, a new cache write operation proceeds. Thus, when using

WTC the data has to be stored both into the disks and their caches before a new write operation is started. Finally, the WBC (Write-Back Cache) policy is less conservative and does not require the cache to be flushed to the hard disks before a new write operation is performed.

- *Disk cache*: Some hard drives feature a cache that performs bundling of write operations to a given sector, thus saving disk head movements and minimizing the number of times disks have to stop and start spinning again. We have considered this option in our experiments as a binary feature.
- *Filesystem*: We have evaluated the `ext4` filesystem, which is the de-facto standard for Linux systems. Additionally, we have tested the `xfs` and `jfs` filesystems, as a previous analysis highlighted them as promising candidates for storing a number of large files. We have additionally tested the RAID's write throughput when no filesystem is instantiated for reference.

The experiments have been carried out by taking all possible combinations of the parameters. For each combination, one hundred 2 GB-sized files have been written using the Linux `dd` tool. This size has been chosen as a trade-off between write performance and later read accessibility for packet traces. The effects of the afore-mentioned configuration parameters in the write throughput are summarized in Tables 4.4 and 4.5 for mechanical and solid-state drives respectively. Specifically, those tables show how to tune parameters to obtain the minimum and maximum write throughput for different combination of disks, along with the mean write throughput, the confidence interval for this mean with a 0.01 significance level and $5^{th}/95^{th}$ percentiles.

Number of disks	Scenario	Strip size	RAID cache policy	Disks' cache	Filesystem	Throughput (Gb/s)			
						average	confidence interval ($\alpha = 0.01$)	Percentile	
								5^{th}	95^{th}
1	min	1 MB	WTC	off	jfs	0.69	$(0.67, 0.70)$	0.58	0.77
	max	64 kB	WBC	on	xfs	1.27	$(1.26, 1.27)$	1.26	1.27
2	min	64 kB	WTC	off	jfs	1.20	$(1.18, 1.22)$	1.11	1.30
	max	64 kB	WBC	on	xfs	2.53	$(2.52, 2.55)$	2.52	2.54
3	min	64 KB	WTC	off	jfs	1.76	$(1.74, 1.79)$	1.58	1.84
	max	1 MB	WBC	on	xfs	3.81	$(3.80, 3.82)$	3.76	3.85
4	min	64 kB	WTC	off	ext4	2.03	$(1.99, 2.07)$	1.79	2.22
	max	256 kB	WBC	on	xfs	5.08	$(5.07, 5.10)$	5.04	5.14
5	min	64 KB	WTC	off	jfs	2.68	$(2.65, 2.70)$	2.49	2.80
	max	64 KB	WBC	on	xfs	6.28	$(6.24, 6.32)$	6.15	6.34
6	min	64 KB	Direct	off	jfs	3.24	$(3.22, 3.27)$	3.13	3.29
	max	256 KB	WBC	on	xfs	7.52	$(7.49, 7.56)$	7.37	7.63
7	min	64 KB	WTC	off	jfs	3.64	$(3.61, 3.67)$	3.53	3.71
	max	1 MB	WBC	on	xfs	8.78	$(8.74, 8.83)$	8.56	8.96
8	min	64 kB	Direct	off	jfs	3.64	$(3.51, 3.76)$	2.90	4.19
	max	1 MB	WBC	on	xfs	10.06	$(10.01, 10.12)$	9.77	10.35
9	min	1 MB	Direct	off	jfs	4.14	$(3.98, 4.30)$	3.27	4.81
	max	1 MB	WBC	on	xfs	11.31	$(11.25, 11.37)$	10.96	11.47
10	min	64 kB	Direct	off	ext4	4.19	$(3.92, 4.45)$	2.00	5.02
	max	1 MB	WBC	on	xfs	12.60	$(12.53, 12.68)$	12.09	13.03
11	min	64 kB	Direct	off	ext4	5.15	$(4.82, 5.48)$	2.51	6.16
	max	1 MB	WBC	on	xfs	13.8	$(13.70, 13.91)$	12.96	14.29
12	min	1 MB	WTC	off	jfs	5.90	$(5.83, 5.97)$	5.63	6.08
	max	1 MB	WBC	on	xfs	14.86	$(14.56, 15.17)$	13.75	15.87

Table 4.4: Write throughput summary results for **mechanical drives**

In practical terms, Table 4.4 shows that a single mechanical disk has an average write throughput of 1.26 Gb/s for its best configuration, with both a narrow confidence interval and a percentile range of roughly tenths of Mb/s. Interestingly, average throughputs scale linearly with the number of disks when they are optimally configured —note that this linearity is not observed if the configuration is not the optimal one. The mean (more precisely, the lower bound of its confidence interval) is over the target rate all packets can be served without losses in the long-term. This suggests that 8 disk will suffice for all scenarios and packet sizes under study if a properly sized buffer is used to overwhelm peaks in the write throughput. In fact, the 5^{th} percentile for the throughput obtained with 8 disk, which is 9.77 Gb/s, is below the target for some of the most typical scenarios on the Internet assuming PCAP header.

Alternatively to the use of a buffer, a lighter packet header such as the one in the RAW format makes 8 disk to suffice for packets with a size up to 512 bytes. Additionally, we can note that a 9-disk RAID presents measurements for its best configuration over the target rate both in the estimation for the mean and 5^{th} percentile. This configuration presents a good trade-off to deal with the oscillations that commodity hard-drives have experimented and to handle all scenarios even those with the largest packet size.

On the other hand, Table 4.5 shows the effect of the diverse configuration parameters over a solid-state RAID volume. Results show that a single SSD drive is capable of consuming nearly the double amount of data than a mechanical one, that is, an average of 2.21 Gb/s in contrast with 1.26 Gb/s. However, our experiments show that the write throughput obtained by a SSD RAID does not scale a well as a mechanical one does. Specifically, this effect is exposed in Fig. 4.10. This figure shows as dashed-dotted line the write throughput that would be obtained if a linear scale applied. It can be easily seen that a mechanical RAID scales very close to linearity for any amount of drives available, while the solid-state RAID looses linearity when the amount of disks is above two, even experiencing performance losses when adding more drives. Fig. 4.10 also shows another effect which is the influence of the existence of a filesystem in the write throughput obtained: while mechanical drives favour from having a filesystem to manage their contents, solid-state drives experience a serious decrease in their performance (reaching a maximum of 49.9 loss with three disks). We also note that, although this relevant effect of the filesystem in performance may be solved by using a flash-aware high-performance filesystem [PP11], the problems when managing SSD RAID volumes are not only on the software side but also on the hardware, i.e., RAID controllers to be aware of the peculiarities that a SSD volume has.

Number of disks	Scenario	Strip size	RAID cache policy	Disks' cache	Filesystem	Throughput (Gb/s)			
						average	confidence interval ($\alpha = 0.01$)	Percentile	
								5^{th}	95^{th}
1	min	64 KB	WTC	off	xfs	0.58	$(0.58, 0.59)$	0.55	0.62
	max	1 MB	WBC	on	jfs	2.21	$(2.18, 2.23)$	2.19	2.21
2	min	64 KB	Direct	off	ext4	1.07	$(1.06, 1.08)$	1.03	1.10
	max	64 KB	WBC	on	jfs	4.44	$(4.32, 4.56)$	4.35	4.43
3	min	64 KB	Direct	off	jfs	1.22	$(1.21, 1.24)$	1.14	1.28
	max	256 KB	WBC	on	xfs	3.96	$(3.68, 4.24)$	2.53	5.82
4	min	64 KB	Direct	off	jfs	1.46	$(1.41, 1.52)$	1.26	1.68
	max	256 KB	Direct	on	xfs	6.43	$(6.03, 6.83)$	4.27	8.57
5	min	64 KB	Direct	off	jfs	1.37	$(1.34, 1.40)$	1.23	1.55
	max	1 MB	Direct	on	xfs	5.10	$(4.73, 5.48)$	4.03	7.65
6	min	64 KB	Direct	off	jfs	1.46	$(1.43, 1.48)$	1.32	1.64
	max	256 KB	WTC	on	xfs	8.18	$(7.80, 8.55)$	5.84	10.62
7	min	64 KB	Direct	offf	jfs	1.69	$(1.64, 1.73)$	1.50	1.96
	max	64 KB	WBC	on	xfs	10.31	$(9.60, 11.03)$	6.07	15.15
8	min	64 KB	WTC	off	jfs	2.15	$(1.99, 2.31)$	1.68	3.82
	max	1 MB	WBC	on	xfs	7.52	$(6.54, 8.51)$	3.17	15.97

Table 4.5: Write throughput summary results for **solid-state drives**

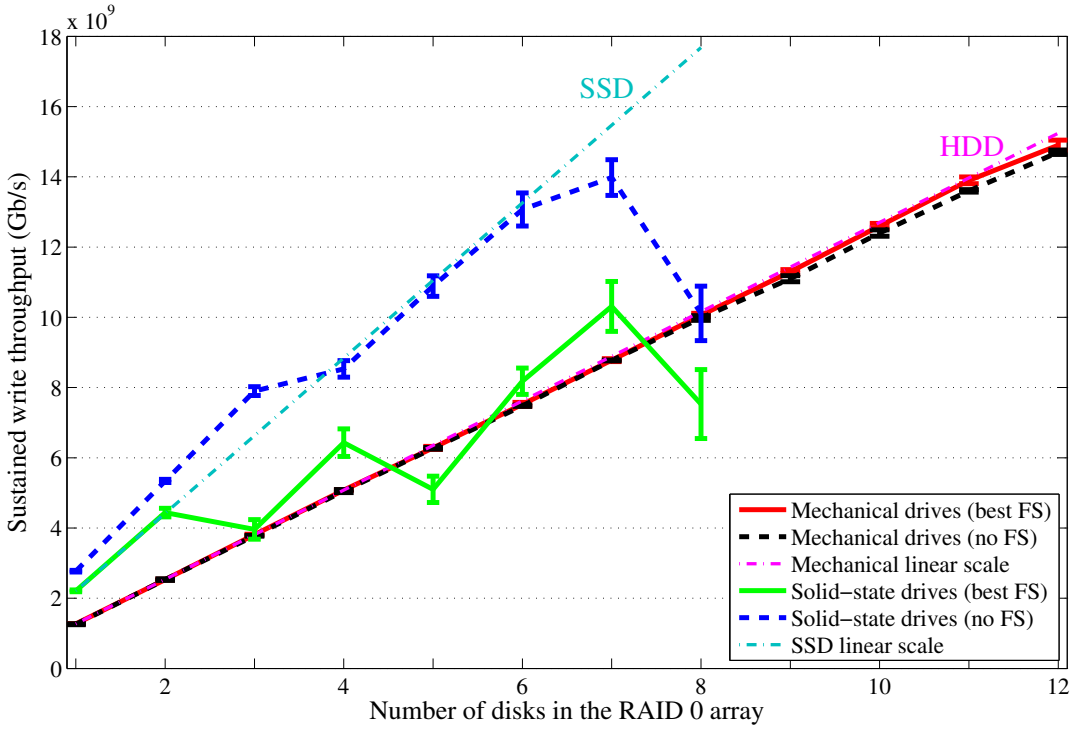


Figure 4.10: Write throughput scalability for mechanical and solid-state drives

Once we have studied how to gauge a disk RAID, we turn our attention to explain how the set of configuration parameters impacted on its performance. We have posed a balanced full-factorial analysis of the data for a RAID-0 array with 9 mechanical hard drives. All factors and their interactions turn out to be statistically significant. Thus, each sample is fully-characterized by the addition of 192 terms, the main-effect factors and an additional term per each of the possible interactions between factors. As a main conclusion, the factor analysis highlighted the importance of hardware caches. Starting from an overall mean of roughly 4 Gb/s, the use of disk caches represent an average addition of 3.1 Gb/s and similarly, the use of WBC policy gives an average gain of 3.3 Gb/s. The volume's strip size has a relatively marginal significance which translates into a few hundred of Mb/s for the best configuration—a strip size of 1 MB. On the other hand, the choice of file system is also significant: `xfs` has shown the best results with an increase of 0.7 Gb/s in mean compared to `jfs` which shows the worst results. This analysis has been restricted to the mechanical case because no solid-state configuration has proven to provide a sustained throughput capable of reaching our 10 Gb/s goal.

Additionally, we found that the choice of the filesystem does not only affect the average write rate, but also exerts a critical effect on its variance. Fig. 4.11 shows the throughput obtained when writing the same files as in the previous

experiment (one hundred 2 GB-sized files) on a 9 mechanical disk RAID-0 volume with the optimal configuration for each filesystem. The figure shows that writing data on the volume if there is no filesystem present has a low-variance behavior. This is a non-practical scenario for later data access, although it is of interest for reference purposes. When a filesystem is set up, throughput oscillations appear. In some filesystems such oscillations are severe. Interestingly, both Fig. 4.11 and Table 4.4 show that the `xfs` filesystem presents the smallest oscillation, making such a filesystem again the most suitable in our setup. In the case of `jfs` and `ext4`, the throughput oscillation may entail adding more disks to the RAID volume to make sure that even the lowest throughputs are over the target rate as already discussed in this section.

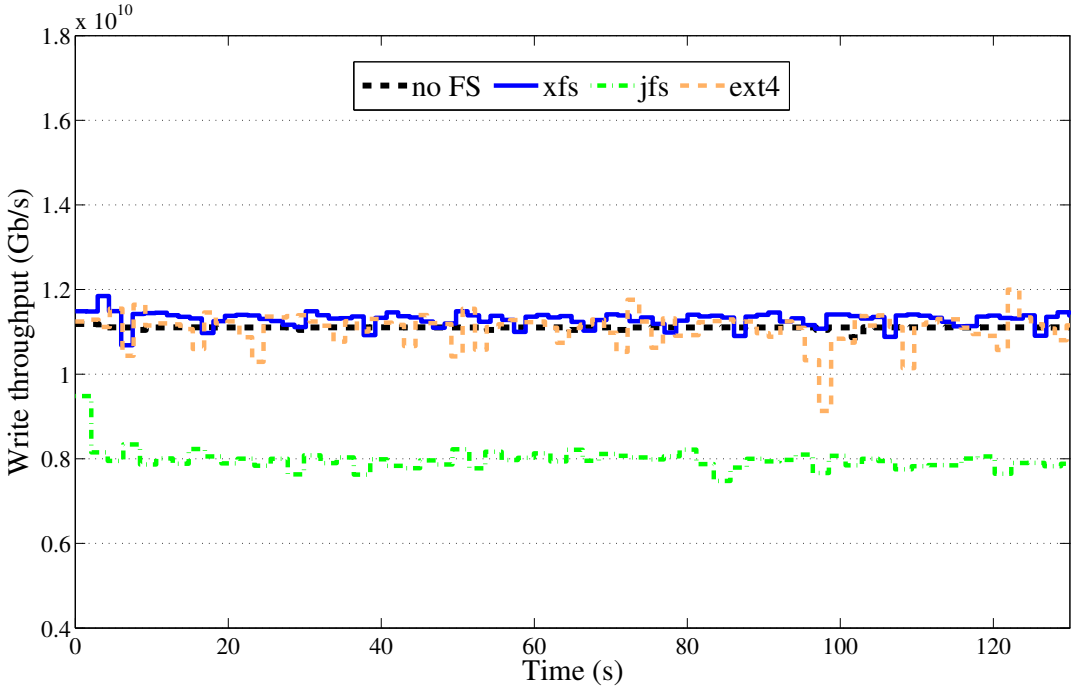


Figure 4.11: Effect of the FS on a 9-mechanical-disks RAID-0 volume with the optimal configuration for each filesystem

Conversely, Fig. 4.12 shows the effect of the filesystem choice over the variability of the write throughput in a solid-sate RAID. In this case, the oscillations amplitude are much wider, leading to poorer performance results in a sustained throughput scenario. Again, we assume this effect is due to the need of SSD-aware software and hardware to deal with the peculiarities implied by this kind of drives.

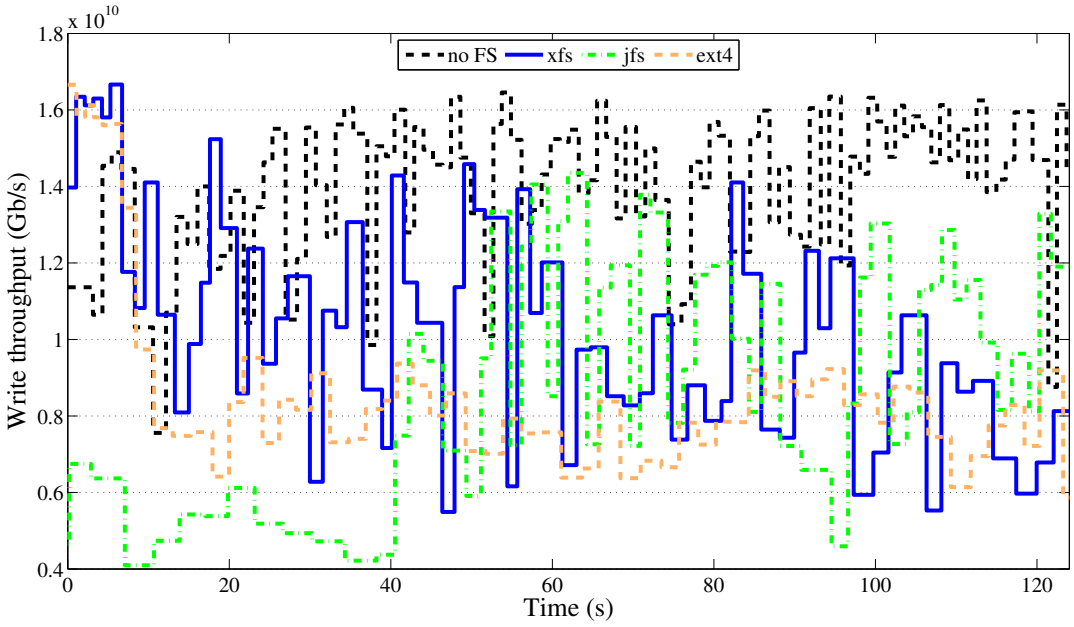


Figure 4.12: Effect of the FS on a 7-SSD RAID-0 volume with the optimal configuration for each filesystem

4.3.3 Network traffic storage solutions

Once we have discussed how to optimize both network traffic sniffing and data storage processes separately, we pay attention to how to make the most of their combination: network traffic storage. In order to carry out such a task, we outline the fundamental techniques NTSS may adopt:

- *System call minimization*: The sniffing and storage processes imply data transfer and synchronization between user and kernel level contexts. Minimizing the amount of system calls used along this interaction is key in order to reduce context switches and improve overall performance. Some ways of minimizing the amount of system calls include buffer mapping, or accessing data in a byte-stream or batch fashion rather than in a per-packet basis.
- *Huge intermediate receiving buffers*: As mentioned in Section 4.3.2, the target storage device may experience write throughput drops. In order to prevent the NTSS from packet losses due to this spurious effects, the final application may use a big enough intermediate buffer.
- *Memory-alignment*: Maximum write performance is achieved when the transfers between system memory and the storage device are done via DMA operations. This way, no CPU nor cache management nor memory bandwidth is spent in the data transfers. This behavior is forced when cre-

ating the destination file with the `O_DIRECT` flag. However, this efficient configuration required the transfer operations to be page-aligned, making memory alignment become a critical feature.

- *Sniffing and storage overlapping*: If the sniffing and storage processes were isolated, their execution could be parallelized and thus overall performance increased.

Note that only two of the high-performance packet capture approaches available on the state-of-the-art, PF_RING and HPCAP, have given rise to a final NTSS, `n2disk` and `hpcapdd` respectively. We focus our attention on them hereafter.

On the one hand, `hpcapdd` was developed on top of the `HPCAP` driver. Both the driver and the application were designed with the goal of optimizing network traffic storage [MSdRR+14b]. Regarding the aforementioned techniques, the `HPCAP+hpcapdd` system instantiates a 1 GB kernel-level buffer, limited by the kernel configuration. The driver is in charge of timestamping and copying the incoming packets into this buffer, so `hpcapdd` can access them in a stream-oriented basis. This buffer is efficiently accessed as it is properly aligned and mapped at user-level. Furthermore, this buffer isolates the sniffing and storage processes, so the overall process is parallelized.

On the other hand, `n2disk` has been recently developed by the authors of PF_RING [DCF13]. Specifically, `n2disk` instantiates one or more packet storage threads, leading to a single-thread (ST) and a multi-thread (MT) version, which are executed in parallel. Each of those threads has an independent memory buffer, and traffic is distributed among them using a hash function. Importantly, `n2disk` not only stores the incoming packets, but also creates additional index files for optimizing later access to the stored data.

We are measuring the percentage of incoming traffic stored versus the number of cores used in our testbed, i.e., an optimized 9-disk RAID-0 volume. Specifically, `n2disk` (with two different flavors) and `hpcapdd` results which are shown in Fig. 4.13. This figure also depicts the amount of fully occupied CPU cores used by each of the NTSS under test.

Remarkably, `hpcapdd` is capable of storing 99.2% of the incoming traffic for synthetic 64-byte packets (CRC included) and 100.0% real-traffic and synthetic maximum-sized packets experiments. Those figures are achieved by fully-occupying two CPU cores.

The results show that `n2disk`'s single-thread version uses one thread for packet sniffing and one more thread for storage whereas the multi-thread version uses one thread for sniffing and four threads for processing and storing. The ST version stores 92.4% of the packets for the 64-byte experiment, and 98.2% for

both maximum-sized packets and real traffic. The MT counterpart stores 99.1% of the packets for the 64-byte experiment, and 98.0% for both maximum-sized packets and real traffic. These last results show that instantiating several threads helps in dealing with the worst-case sniffing scenario, i.e., 64-byte packets, but does not solve demanding storage throughput scenarios.

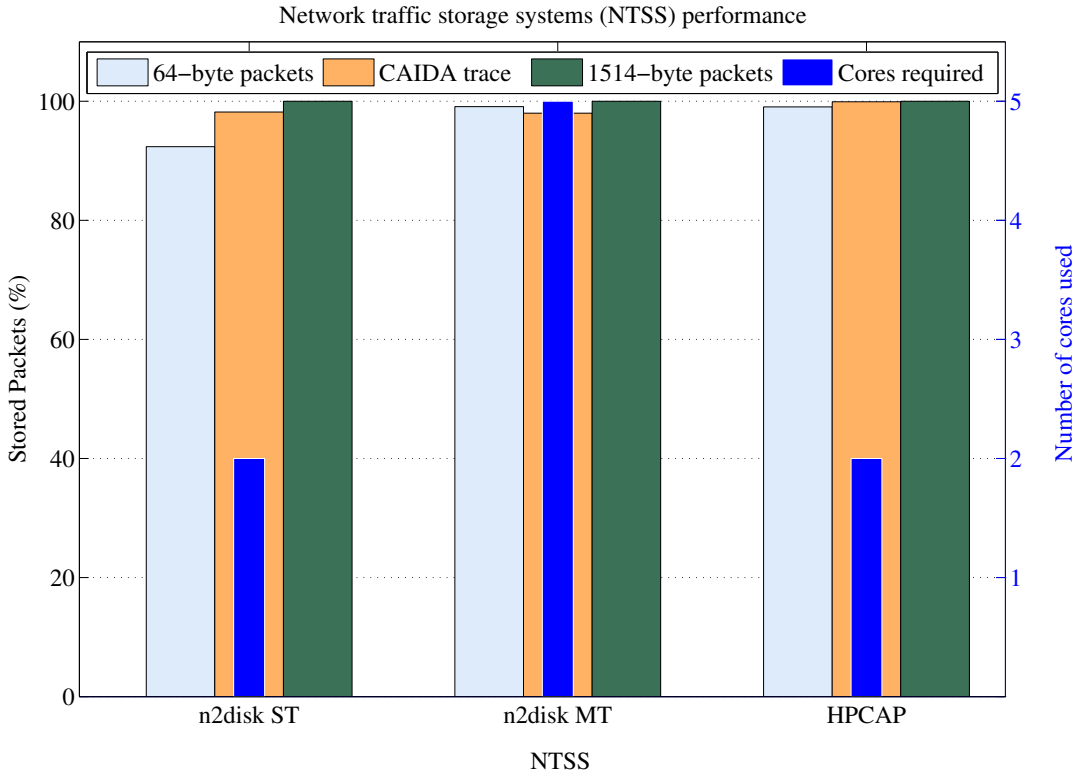


Figure 4.13: Percentage of stored packets (into a RAID-0 volume) for a full-saturated 10 GbE link versus the number of occupied CPU cores for a 30-minutes experiment

4.4 Duplicates detection and removal

The presence of duplicate data is a general problem for systems with the aim of extracting information from this data which has been studied in the last decades [EIV07,CCMO11]. Those studies focus mainly on the field of databases, knowledge engineering and statistical analysis. In our field, the data duplication emerges with the appearance of duplicated packets in a network. The appearance of such duplicated packets can generate problem at two different levels: first, generating a malfunction in the network segment those duplicates appeared at, and second, obstructing network monitoring tasks being carried out over the network. The first is dependant on the diverse protocols and configurations present in the network under study and is difficult to measure in an objective way. This section focuses on, by means of the HPCAP driver, solving the second problem in order to propel an effective network monitoring policy.

There are different kinds of packet duplication problems that may arise that may arise in a network, which are deeply explained in [UMMI13]. As the packet processing time is tight in high-speed network processing, we restrict our efforts to detecting the lowest-level type of duplicates: switching duplicates. This kind of duplicates are typically generated due to port mirroring policies carried out at layer 2 interconnection elements. For example, when a mirror port is configured to redirect the traffic traversing a subset of the existing ports in a switch, both ingress and egress packets must be mirrored or traffic coming to/from non-mirrored interchanged with the mirrored ones would be lost. However, this generate that packets that ingress and egress the switch through mirrored ports will be redirected twice. The appearance of those switching duplicates distort MAC-to-MAC analysis tasks and consequently affects upper layers.

In order to address the duplicate removal problem, works such as [UMMI13, MGAG+11] follow a packet window approach, so that each packet is compared with the X previous packets. However, if the duplicate removal process is to be carried out at kernel in the HPCAP driver, it may benefit from some calculations that are done at hardware level by the NIC and placed inside each incoming packet's receive descriptor. That is precisely the case of the Toeplitz hash value which is calculated for RSS packet distribution (see Section 3.1). Further information regarding the information contained in each packet's receive descriptor structure can be found in section 7.1.6.2 of [Int12]. Specifically, if one packet is a duplicated copy of another they will present the same hash value, so it can be applied in order to reduce the amount of comparisons carried out and thus reduce the amount of processing time used by the duplicate removal process.

The duplicate removal approach implemented inside the HPCAP driver uses a multi-level hash table, which is indexed by the hash value of each incoming packet modulo the size of the table. For each record in the table, the data of K

packets is stored, being K the amount of levels or pools in the table, as shown in 4.14. Having multiple pools pretends to mitigate the effects that hash value collisions may have over the duplicate detection accuracy. For each packet, the table stores:

- the packet's length,
- arrival timestamp,
- and the first M bytes of data.

The arrival timestamp is used to avoid pointing an incoming packet as a duplicate if the time elapsed between the arrival of the first packet and the new one exceeds a certain threshold. This threshold depends on each network's specific configuration, and in the our cases under study, $200\ ms$ has proven to be a reasonable value. The amount of data stored per packet M may be configured at compile-time. Previous studies [UMMI13] show that an amount of 60 bytes is enough to identify switching duplicates. However, in order to keep the table's records memory-aligned, 70 bytes of data of each packet are stored as default value. Importantly, the total size of the table N is limited by kernel memory allocation policies to a table with 32768 records. Note also that, if the table has a total amount of N records, each of the K pools will have thus N/K packets.

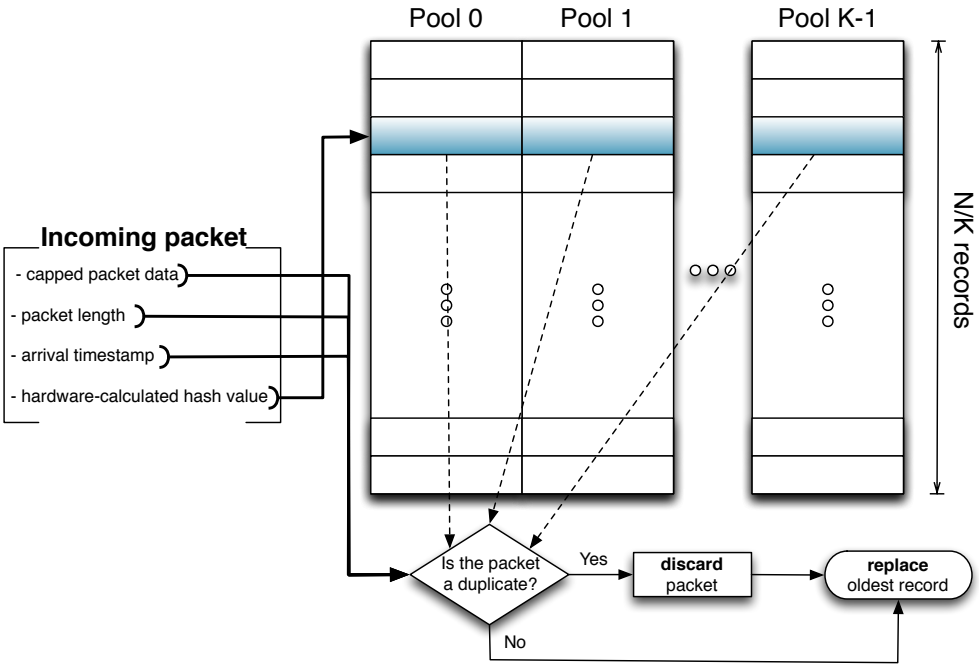


Figure 4.14: Duplicate packet removal hash table structure

The scheme used to decide whether a packet is a duplicate or not is the following: Every time an incoming packet is fetched by the HPCAP driver, the

packet is compared against all the records stored which has the same hash value that had the same length and is inside the time-sliding window. If the comparison concludes that the new packet is a duplicate, it is not copied to the kernel intermediate buffer so no more resources are dedicated to it. Regardless the packet is classified as a duplicate or not, the record corresponding to the oldest packet which had the same hash value is replaced with the new packet's data for future comparisons.

Importantly, the additional processing required for duplicate packet removal may damage capture performance because the timing constraints are high and the time available for processing each packet is small, as already shown in section 4.2.2. In the following, both the classification accuracy and the impact over capture performance of this duplicate removal approach is analysed.

4.4.1 Accuracy

The accuracy of our packet removal approach has been tested with two different traffic sources, each one belonging to an industrial network that has been proven to experience the packet duplication problem. The characteristics of each of the traces is summarized in Table 4.6. The first trace has a very relevant ratio of duplicates, 32.1%, whereas Trace B has an amount of duplicate packets below 1%.

	Trace A	**Trace B**
Number of packets	32,912,969	149,505,380
Average rate	2.03 Kp/s 3.97 Mb/s	9.56 Kp/s 48 Mb/s
Average packet size	245.0 bytes	634.1 bytes
Number of duplicates	10,569,746	20,933

Table 4.6: Characteristics of the traces used for duplicate removal testing

In order to obtain accuracy results of this approach, the tool `infodups` [inf] is used as a ground-truth reference. Importantly, the time-sliding window has been configured for all traces to $200\ ms$ for both the ground-truth reference tool and the HPCAP driver-level approach. The duplicate classification accuracy has been tested for different window sizes and for different amount of pools, depending on the table's total size. We have restricted the experimental space so that the smallest pools used contain 1024 packet records, as a smaller number

would increment the probability of collisions and may have a negative impact over accuracy.

Table 4.7 represents the True Positive Rate (TPR) against the False Positive Rate (FPR) for each combination under study. TPR gives the ratio of packets that were duplicated and were classified as a duplicate, versus the total amount of packets that were duplicates; i.e., a value of 1 means that all the duplicated packets have been detected. On the other hand, FPR stands for the ratio between the packets that have wrongly classified as duplicates, versus the total amount that were not duplicates; i.e., a value of 0 means that no non-duplicate packet was labelled as a duplicate. Consequently, an ideal classifier would yield to a TPR of 1 and a FPR of 0.

Total table size	**Number of pools**	**Trace A**		**Trace B**	
		TPR	FPR	TPR	FPR
1K	1	0.957	0	0.935	0
2K	1	0.994	0	0.974	0
	2	0.957	0	0.934	0
4K	1	0.999	0	0.995	0
	2	0.994	0	0.974	0
	4	0.957	0	0.935	0
8K	1	0.999	0	0.996	0
	2	0.999	0	0.995	0
	4	0.994	0	0.974	0
	8	0.957	0	0.935	0
16K	1	**1.000**	0	0.998	0
	2	0.999	0	0.996	0
	4	0.999	0	0.995	0
	8	0.994	0	0.974	0
	16	0.957	0	0.935	0
32K	1	**1.000**	0	**1.000**	0
	2	**1.000**	0	0.998	0
	4	0.999	0	0.996	0
	8	0.999	0	0.995	0
	16	0.994	0	0.974	0
	32	0.957	0	0.935	0

Table 4.7: Duplicate classification accuracy for different real traces varying the hash table configuration

The results obtained demonstrate that for a fixed total size of table, instantiating several pools have a negative impact over the classification accuracy. This may be a consequence of the hash function being balanced enough, at least along the temporal sliding-window we are taking into account. Furthermore, for a fixed number of pools in the table, the accuracy is increased with the total size

of the table. It is worth mentioning that no non-duplicate packets were classified as being a duplicate for any combination, i.e., the FPR is always zero, so this implementation can be applied in real-world monitoring tasks without a significant information lost.

Importantly, the implementation with a 32K packets table distributed among one unique pool has proven to obtain the accuracy of a perfect classifier, so increasing the table size would not lead to better results. Consequently, the effort of using other memory management alternatives in order to increase the hash table's size (remember that 32K packets was a limit imposed by the kernel dynamic allocation mechanism) such as kernel-level huge-pages mapping or low-level kernel tuning becomes a non-urgent task. However, different traffic profiles may experience different accuracy results so this must be part of the future work.

4.4.2 Performance

Once the accuracy of our duplicated packet removal implementations has been assessed, it is time to measure the impact of this additional process over HPCAP's packet capture performance. As a reminder, in a fully-saturated 10 GbE link receiving minimal-sized, i.e., 64 bytes, packets, the time available for processing each packet is 67.2 ns. In section 4.2.2 we checked that requesting a timestamp value for every incoming packet has a negative impact in packet capture performance, so it is expected that our heavier duplicate classification process will have an even more negative effect.

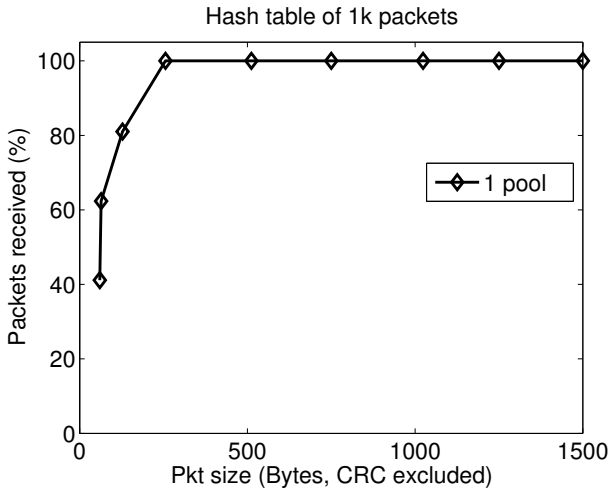
The diverse plots inside Fig. 4.15 represent the percentage of packets captured when the duplicate detection is activated for different hash table configurations. The plots show the impact of this process when packets of constant size are captured for different sizes. Importantly, those plots show the worst-case scenario, in which no duplicates packet appear. This was easily achieved using the FPGA-based generator, already mentioned in Section 3.4.2, configured so that it increased each packet's data. The worst case scenario gives the impact over capture performance when none of the computational power invested in duplicate detection has a profit, as no packet is non-copied in the buffer for upper-layer processing.

The results obtained show that, for a fixed total table size, adding more pools damages capture performance. This seem natural, as the more pools are available, the more packet comparisons must be done for each incoming packet and thus the higher the processing time will be. If this result is placed in context with the previous accuracy evaluation results, it is concluded that the lower the amount of pools is the higher the capture performance and classification accu-

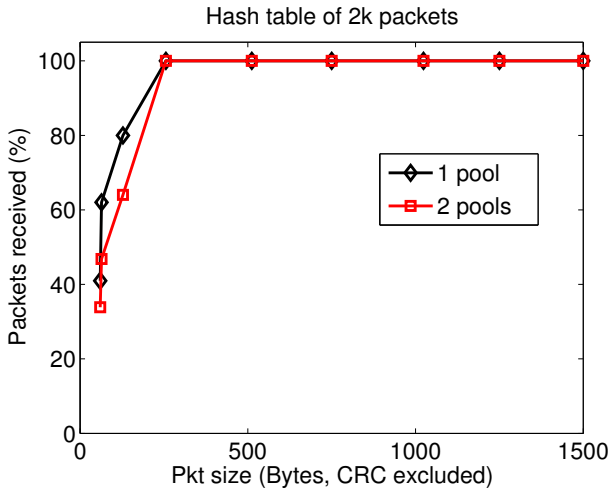
racy, so the more optimal is the duplicate removal implementation.

Additionally, results show that increasing the total table size has nearly-negligible negative impact on capture performance: for table sizes of 16K and 32K packets a small performance loss is experienced if results are compared for a fixed number of pools. However, in the case of the most accurate configuration, i.e., a 32K table with an unique pool, the performance experienced is similar to the one shown by the rest of size tables with just one pool.

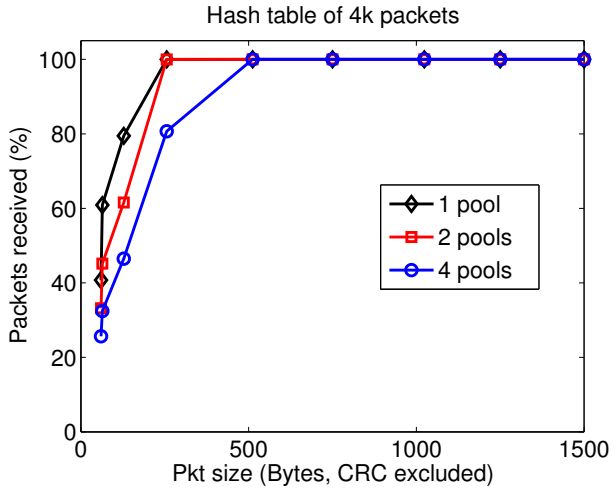
Results also give a minimum packet size above which packet capture performance is optimal. This size is 256 bytes for those configurations with one or two pools, and higher for the rest. Importantly, a packet capture performance test was made using the optimal configuration for capturing the traffic contained in traces A and B, and no packet loss was experienced.



(a) Hash table of 1K packets

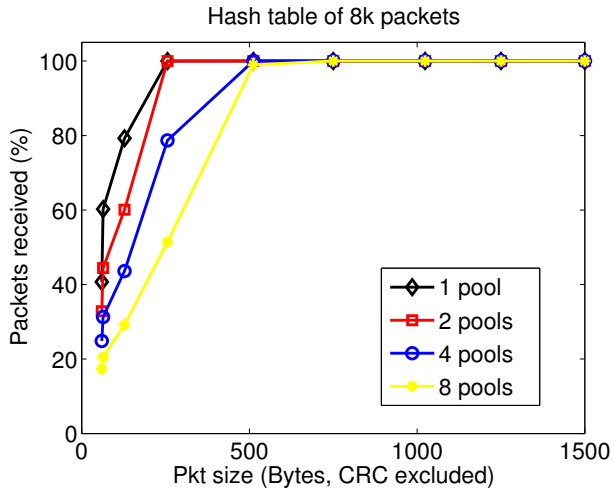


(b) Hash table of 2K packets

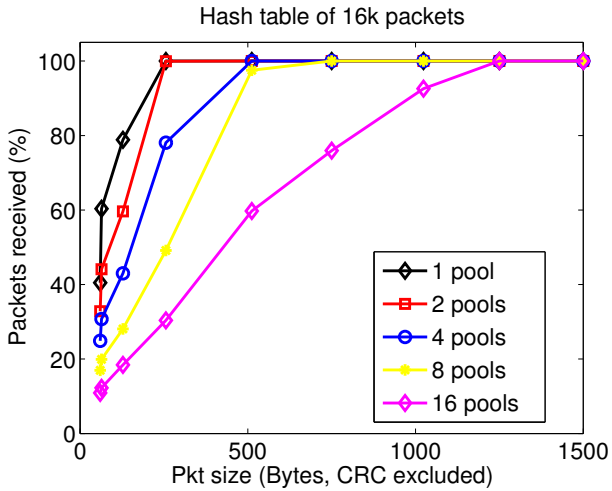


(c) Hash table of 4K packets

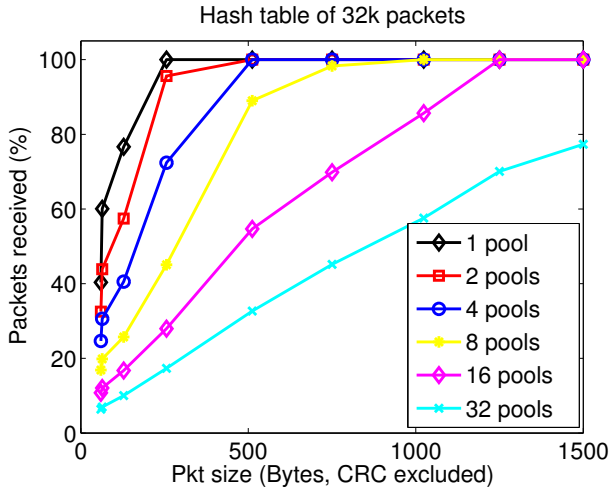
Figure 4.15: $(a), (b), (c)$ Duplicate removal performance evaluation (10 Gb/s link, fully-saturated with different packet sizes) for different hash table configurations



(d) Hash table of 8K packets



(e) Hash table of 16K packets



(f) Hash table of 32K packets

Figure 4.15: $(d), (e), (f)$ Duplicate removal performance evaluation (10 Gb/s link, fully-saturated with different packet sizes) for different hash table configurations

4.5 Conclusions

The first section of this chapter has been devoted to give an overview over HPCAP's structure and design. The rest of the sections have been devoted to study and evaluate those feature that differentiate HPCAP amongst other high-performance packet capture alternatives. The conclusions of those studies are manifold:

Timestamping

Some high-performance packet capture engines use techniques such as batch processing in order to enhance their packet capture performance, but this is at the expense of packet timestamping accuracy. After alerting research community and measuring the problem, two approaches were proposed to mitigate timestamping degradation:

- UDTS/WDTS algorithms that distribute the inter-batch time gap among the different packets composing a batch. Those solutions offer good levels of accuracy when the link is fully saturated, but the error is greatly increased when if there are gaps between packets. However, they could be used by those packet capture engines that, while relying on batch processing, need to increase their timestamping accuracy.
- 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. This is the policy implemented by the HPCAP driver, and offers reasonable timestamping accuracy in any scenario.

As future work, we propose a combination of both approaches according to the link load: i.e., using WDTS when the link is nearly saturated for distributing the timestamp between groups of packets and, otherwise, using KPT policy for independently timestamping each incoming 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).

Packet storage

The most significant findings are:

1. Out-of-the-box tools (i.e., vanilla network drivers and `tcpdump`) are not

enough to sniff and store packets at 10 Gb/s using off-the-shelf systems. Indeed, the obtained performance is far below 10 Gb/s. The application of some ideas discussed in this work improves the performance of such tools up to rates that potentially may be useful in networks with low utilization.

2. Optimizations must be applied to improve sniffing performance and make the most of commodity multi-core servers and modern NICs. Indeed, there are several open-source packet sniffing engines, capable of achieving full-packet sniffing at 10 Gb/s on a commodity server. Affinity planning appears to be a key factor, which is relevant for both optimized and out-of-the box applications.

3. Importantly, some of these optimizations have collateral effects: batch processing degrades timestamping accuracy, whereas the use of multiple receiving queues may cause packet re-ordering. Thus, depending on the application requirements, such optimizations must be carefully chosen.

4. Although a single commodity hard drive is not able to achieve enough write throughput to cope with a 10 GbE link, we may improve the storage performance by using skillfully tuned RAID volumes. With this tuning, the average writing throughput is beyond 10 Gb/s using 9 high-end mechanical disks.

5. The write throughput of commodity hard-drives has shown significant oscillations over the mean across time. While in file systems such as xfs those oscillations are modest, others show remarkable excursions.

6. The case which is usually considered to be the most demanding in terms of packet sniffing (i.e., minimal-size packets), is the least demanding scenario in terms of packet storage throughput. Similarly, the best-case for packet sniffing (i.e., maximal-size packets) becomes the most demanding scenario in terms of packet storage throughput.

7. Obtaining maximum performance in a NTSS does not only imply properly tuning the sniffing and storage processes alone, but their interaction must also be carefully planned.

In conclusion, this study has provided both the research community and practitioners with a roadmap not only to understand and use state-of-the-art NTSS solutions based on off-the-shelf systems, but also to implement and deploy their own systems. We expect the lessons and ideas we share here may open new opportunities to the use of off-the-shelf systems in areas traditionally reserved for high-end and expensive hardware.

Duplicate removal

The appearance of duplicated packets in a monitored network is common problem that apart from consuming valuable resources may damage all the analysis tasks carried out above this traffic. In order to identify those duplicated packets as soon as possible and minimize their resource consumption, a duplicate removal methodology has been implemented inside the HPCAP kernel level.

Importantly, carrying out the packet duplicate task at kernel level may benefit from information that is calculated by the NIC's hardware and transferred to the CPU, but is commonly ignored. Specifically, HPCAP may benefit on the Toeplitz hash value calculated at hardware level to carry out a hash-based pre-filtering policy.

For different configuration options, an accuracy and worst-case packet capture performance analysis has been carried, leading to an optimal configuration in both terms: a 32K packets hash table with a unique pool. This configuration has proven to give an accuracy similar to an ideal classifier, and experiences no packet losses for traffic with an average size above 256 bytes.

5 M³OMon: A framework on top of HPCAP

As an attempt to make network managers' life easier, we present M³Omon, a system architecture that helps to develop monitoring applications and perform network diagnosis. M³Omon behaves as an intermediate layer between the traffic and monitoring applications that provides advanced features, high performance and low cost. Such advanced features leverage a multi-granular and multi-purpose approach to the monitoring problem. Multi-granular monitoring gives answer to tasks that use traffic aggregates to identify an event, and requires either flow records or packet data or even both to understand it and, eventually, take the convenient countermeasures. M³Omon provides a simple API to access traffic simultaneously at several different granularities—i.e., packet-level, flow-level and aggregate statistics. The multi-purposed design of M³Omon allows not only performing tasks in parallel that are specifically targeted to different traffic-related purposes (e.g., traffic classification and intrusion detection) but also sharing granularities between applications—e.g., several concurrent applications fed from flow records that are provided by M³Omon. Finally, the low-cost characteristic is brought by off-the-shelf systems (the combination of open-source software and commodity hardware) and the high performance is achieved thanks to modifications in the standard NIC driver, low-level hardware interaction, efficient memory management and programming optimization.

5.1 Introduction

Network managers' life has become progressively more laborious given the ever-increasing users' demands for both traffic volumes and further quality of experience, the arrival of novel and heterogeneous applications, and peaks in

operational and capital expenditures [GDFM+12]. To complicate matters, there is a lack of synergy between traffic capture engines and network management applications (e.g., network anomaly and intrusion detection systems, NIDS, or traffic classification tools). As a result, traffic capture devices do not incorporate the necessary flexibility to actually process the traffic, which is the ultimate goal of any network manager. To tackle this issue we present and make public M^{3}Omon, a monitoring framework that exploits the interaction between the traffic capture and processing tasks, which provides: (i) a simplification of network monitoring tools, (ii) a significant performance increase and (iii) a CAPEX reduction thanks to the use of off-the-shelf systems—the combination of open-source software and commodity hardware [BDKC10].

5.1.1 Novel features: multi-granular / multi-purpose

Network trouble shooting typically entails the following three steps. First, traffic aggregates are used to spot an anomalous situation such as sudden traffic peak. Second, flow level traces are employed, for example, to identify the troublesome agents (IP address/ranges, port numbers). Third, inspection of a traffic trace is made in order to further diagnose the problem (for example, duplicated or lost packets). We refer to this kind of applications that leverage more than one granularity of data as multi-granular. However we find that the literature fails to provide practitioners with an intermediate software layer that actually provides such multi-granular data access through an unified API. We note that this is a very challenging problem because flow record logging must happen concurrently with trace storage and this is very demanding in terms of processing, parallelism and disk throughput. Our novel M^{3}Omon software efficiently tackles this issue by providing an API whereby each application running over it may ask for data at different granularities. Remarkably, this is performed in commodity hardware at very high speed. Therefore, from the point of view of a practitioner, there is not difference between asking for a packet or a flow-record. Specifically, M^{3}Omon defines in its current version three levels of granularity, namely time-series aggregates (e.g., MRTG (Multi-Router Traffic Grapher)-like series), flow records and packet traces. All of these granularities may be retrieved in real-time or after being stored in a hard drive. This is a clear departure from the current state of the art that has focused on packet capture, storage and flow record creation as separate processes. Clearly, doing all these activities in parallel is most challenging at high-speed, due to the very demanding parallel processing that must be achieved in a constrained general-purpose architecture.

Additionally, monitoring applications may actually be distributed in several physical machines, which carry out the most diverse monitoring tasks—e.g. intrusion detection and traffic classification applications. In most cases no syner-

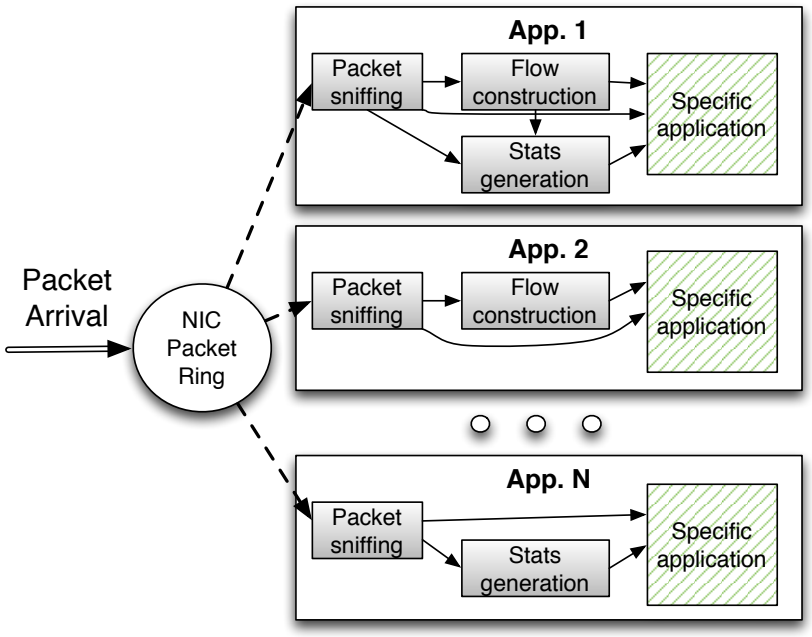
gies between monitoring applications are exploited at all, which implies a severe performance loss. For example, a firewall and a traffic classification application (say, for billing purposes), that both perform flow records creation in separate machines. We strongly believe that there are many synergies between monitoring applications, which can be exploited to decrease the CAPEX and increase the efficiency of monitoring systems. Indeed, M^{3}Omon performs data pre-processing at several granularities, which are made available for all the applications running on top of it. As an example, M^{3}Omon provides flow records to all the applications that require such flow-level data, thus saving the extra effort of flow record creation on a per-application basis. This novel characteristic entails a significant advantage because nowadays more and more monitoring applications use flow records [LSBG13]. Several M^{3}Omon threads must access the packet-level data at the same time —i.e., packet capture and storage, flow construction and statistic generation. This is very challenging in terms of parallel processing due to the high-speed. We note that synchronization at high-speed is very hard to achieve as it strongly penalizes performance. Therefore, even the low-level kernel interaction must be carefully planned in the traffic capture and storage chain, which drove the development of an ad-hoc driver called HPCAP, was previously described in Chapter 4.

Figure 5.1 presents a workflow description that portrays the conventional decoupled monitoring applications scenario compared to our current proposal. On the one hand, Figure 5.1(a) depicts the conventional approach for a monitoring probe, where several applications individually retrieve traffic from the NIC, separately pre-process the data on their own and finally execute a specific monitoring task. On the other hand, Figure 5.1(b) highlights our novel approach, where the proposed framework provides a common pre-processed data source for all the specific applications. This data source is the output of M^{3}Omon, which is the only module capturing packets and pre-processing them. After that, each specific application addresses its respective final task in the same way than the conventional approach. As a consequence, repeated efforts that were previously carried out by each specific monitoring task are now moved into M^{3}Omon, resulting in a decreased complexity and higher efficiency of multi-granular monitoring applications.

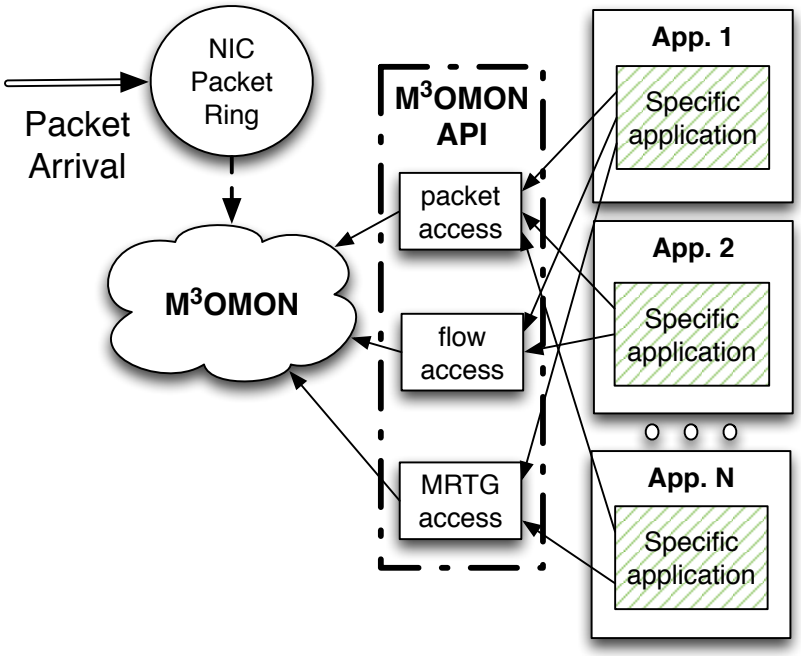
5.1.2 High-performance in off-the-self systems

With the previous characteristics in mind, we note that current monitoring probes render useless if their capacity is not in the multi-Gb/s range [Sys13]. Indeed, line-rate monitoring in 10GbE links involves processing at up to tens of million packets per second (Mp/s) and millions of flows per second (Mf/s).

In order to provide the multi-granular and multi-purpose features, involving



(a) Conventional approach



(b) M^{3}Omon

Figure 5.1: Contrast between our approach and a conventional one

both packets and flow records, several novel optimization techniques have been applied at three different levels. First, each module of M^{3}Omon and their corresponding client monitoring applications are bound to a different CPU core, thus, exploding low-level hardware affinities [GDMR+13]. Second, HPCAP driver follows the recently-proposed modifications for both standard NIC driver and default network stack optimization [GDMR+13], and, additionally, the number of write operations to hard drives is minimized to optimize storage throughput. Third, we have performed software optimization, efficient memory management, and tailored data structures in the M^{3}Omon layer itself.

We have thoroughly evaluated the performance of HPCAP and M^{3}Omon on a general-purpose server. The results show that the system is able to deal with 10GbE links even in challenging scenarios with small packet size. Moreover, to show the applicability of our system, we present a network traffic monitoring tool, named DetectPro, implemented over the proposed framework, which is able to provide monitoring statistics, report alarms and afterwards perform forensic analysis based on packet-level traces, flow-level records and aggregate statistic logs. DetectPro has been put in production in commercial networks from several banks and ISPs [nau13]. We believe that such experiences highlight the importance of the interaction between traffic measurements and statistics at different granularities, especially traffic aggregates and flow records.

As an additional contribution, we release the code of the proposed driver, HPCAP, and intermediate pre-processing software layer, M^{3}Omon, under an open-source license [Hig13], which may be useful for the research community for comparison purposes and moving forward in the development of high-performance tasks on off-the-self systems.

5.1.3 Contributions

Finally, as a summary, the contributions of M^{3}Omon are manifold: (i) it features an API to facilitate the development of multi-granular applications (first M in its name); (ii) it provides a novel mechanism to construct and share data at different granularities between applications thus saving duplicated efforts (multi-purpose, second M); (iii) it works at multi-Gb/s rates after a carefully low-level hardware interaction, a new NIC driver design, and software optimization (third M); and, (iv) for the first time, all of this runs on off-the-self systems (that is the O) thus providing low-cost and additional flexibility and scalability, available under an open-source license. The rest of this chapter details all these M^{3}Omon functionalities and their implementation.

5.2 System Overview

Let us detail our architecture, which is made up of the three different blocks shown in Figure 5.2: HPCAP, M^{3}Omon and an end-user API.

The HPCAP block consists of one kernel-level module, which implements a *traffic sniffer*, responsible for capturing incoming packets at line-rate. This module instantiates a kernel-level thread in charge of polling the NIC for new packets, and copies them into an intermediate packet buffer. As HPCAP design goals and features have been already described in Chapter 4, this module will not be discussed in this chapter.

M^{3}Omon runs on top of HPCAP. It consists of a set of user-level processing modules which simultaneously fed from the traffic sniffed by HPCAP. Such user-level modules are in charge of delivering packet-level, flow-level and MRTG-like data accessible by any end-user application—according to the multi -granular and -purpose features.

Finally, M^{3}Omon provides an API that allows monitoring applications to access the different granularity data both in a real-time and offline fashion. This means that applications running on top of M^{3}Omon can focus on final monitoring tasks such as DPI, statistical classification or security analysis starting from a common data base. This architecture allows users to instantiate a variable number of modules, referred as *application layer*, each of them responsible for a specific monitoring task.

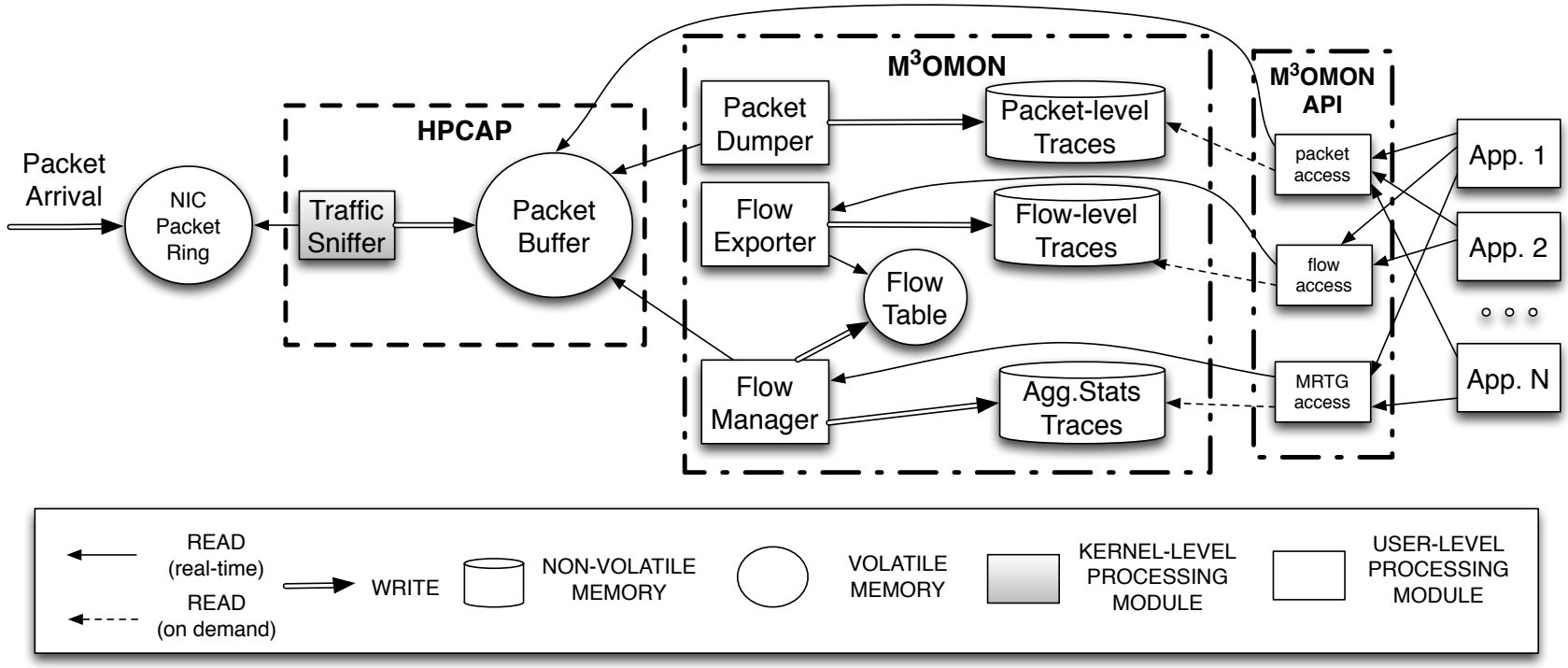


Figure 5.2: M^{3}Omon's Architecture

Note that the different tasks and modules (sniffing, flow handling, statistics collecting, multiple level traces dumping, specific monitoring) are simultaneously running on different CPU cores. Performance is not degraded due to a careful scheduling and to the use of CPU affinity techniques—i.e., the execution of each thread bound to a different core. Such schedule allows the system to make the most of contemporary computer architectures.

5.2.1 M^{3}Omon

M^{3}Omon in turns consists of three different modules in charge of generating triple-grain monitoring information, namely: Packet Dumper, Flow Manager and Flow Exporter.

Packet dumper: The Packet dumper module is responsible for generating packet-level traces in disk. The module is implemented as a HPCAP listener that consumes packets from the intermediate packet buffer through the driver's API. It reads fixed-size blocks of bytes (e.g., 1 MB) from the buffer and writes them into disk. Note that working with byte-blocks instead of packets minimizes the amount of write operations. This fact combined with page-alignment of those byte blocks allows the module to obtain the maximum write performance. It is worth remarking that the destination volume has to be capable of consuming the desired data throughput. In order to avoid concurrent write accesses to disk leading to a performance degradation, it is advisable that the storage device for packet traces is independent from the devices that any other processes may use.

To give more manageability and compatibility, packet traces are split in fixed-size files (e.g., 2 GB). The captured traces may be processed offline—e.g., with forensic analysis purposes. As the non-volatile storage space in the system is limited, an independent periodic process (e.g. a script invoked through the `cron` API) is in charge of deleting old capture files when the volume is nearly-full. This deletion process only implies removing filesystem's i-nodes and it has proven not to interfere in the system's performance. Note that, using the multi-granular approach, such traces may be infrequently accessed and may be bounded to specific time intervals as flow-level and MRTG statistics provide the most relevant information needed for monitoring.

Flow Manager: Concurrently, the Flow Manager module is in charge of two tasks: flow reconstruction and statistic collection. Again, the module acts as another HPCAP listener, reading packets from the buffer one-by-one. For each packet, its corresponding flow information and the aggregate counters are updated. This process is computationally heavy and its implementation has been tailored in several ways to achieve line-rate throughput.

To store flows, this module uses a table indexed with a hash over the 5-tuple, handling collisions with linked lists. A flow is marked as expired when it does not present packets during a given time interval (e.g., 30 seconds) or when it has been explicitly finished with TCP FIN/RST flags. Note that the timeout expiration process requires a garbage-collector mechanism, but scanning the whole hash table is computationally unaffordable. For this reason, we keep a list of active flows with each node containing a pointer to the flow record in the hash table. This active list is sorted by the last packet's timestamp in decreasing order. When a flow is updated with the information of a new packet, the corresponding node in the active list is moved to the end, keeping the list sorted with negligible computational cost. Thus, the garbage-collector only checks the first active flows in order to expire inactive flows. Expired flows (for both timeout and flags) are queued to be exported for the next module.

All the memory used during the process (structures, nodes, lists and hash-table) is pre-allocated in a memory pool in order to reduce insertion/deletion times. Once a data structure is no longer required, the memory is returned to the pool but not de-allocated. Thus, such data structure can be reused without a new allocation process. This policy significantly increases the performance of the system.

Importantly, the Flow Manager module periodically (e.g., every second) generates the MRTG statistics both writing them to a file and sending them through a multicast socket. The multicast socket allows several applications to concurrently access the exported real-time data with no additional cost, which perfectly fits the multi-purpose philosophy. Note that the use of multicast sockets allows that the applications making use of the exported data to be located on different machines, thus distributing the monitoring process. The MRTG statistics are also written to disk in case offline access to the data is needed. Such statistics are generated every second for three metrics: packets, bytes and active flows.

Flow Exporter: Finally, the Flow Exporter module instantiates a different thread which is in charge of exporting the expired flow records both writing them to disk and using a multicast socket as previously described. The multicast sockets give the same advantages for flow processing as aforementioned. Note that different multicast groups are used for the MRTG and flow data and for each network interface. This way, upper-layer applications can subscribe only to the data they desire. Flows may be exported in either an extended NetFlow or standard IPFIX formats. Each flow record contains: 5-tuple, MAC addresses, first/last packet timestamps, counters of bytes and packets, average/standard deviation/minimum/maximum for both packet length and interarrival times, TCP statistics (e.g., counters of flags or number of packets with TCP zero-window advertisements), the first 10 packet lengths and interarrivals and, if required, the first N bytes of payload, which is configurable.

5.2.2 M^{3}Omon's API

M^{3}Omon provides a simple and efficient API to access the multi-granular data from monitoring applications. The API provides real-time and offline access to the data gathered by the system, namely: raw packets (PCAP format), MRTG statistics and flow records. It has been designed taking as a reference the de-facto standard PCAP library.

To access real-time packet-level data, M^{3}Omon allows applications to hook as HPCAP listeners and read packets using a packet loop function similar to `pcap_loop` implemented in the PCAP library. Additionally, monitoring applications may need to process captured traffic on demand (e.g forensic analysis). In this case, the API offers a similar packet loop mechanism that accesses the packet data once users define the time interval they wish to analyze. Note that offline access to the packet traces is scheduled as low I/O priority process, so that write performance is minimally affected, at the expense of a higher access latency.

To access the exported flow records in a real-time fashion, the API provides a method to loop over the flow records subscribing to the corresponding multicast group. This API allows the gathering of flow records either in the same machine and in a distributed way. Additionally, the stored flow records may be accessed on demand through a method that loops over the flow records in a given time interval. Note that the flow table shown in Figure 5.2 and its state is not accessible through the API due to security, consistency and mutual exclusion policies.

The M^{3}Omon API provides similar methods to access MRTG information both real-time and on demand. To access MRTG in a real-time fashion, a method to loop over the MRTG registers given a multicast MRTG group is provided. To access MRTG data on demand, a time interval must be specified to loop over the data.

This approach provides great flexibility and allows monitoring applications to obtain data at any of the three granularities in an efficient and if possible distributed way. Due to the very own nature of the API implementation several applications may access any of the offered data with minimum processing overhead, in full compliance with both the multi-granular and multi-purpose approaches.

5.3 Performance Evaluation Results

Our experimental setup consists of two servers, one receiver and one sender, directly connected with an optical fiber link. Both servers are based on two Intel Xeon E52630 equipped with six cores per processor running at 2.30 GHz and with 96 GB of DDR3 RAM at 1333 MHz. The motherboard model is Supermicro X9DR3-F with two processor sockets (or NUMA nodes) and three PCIe 3.0 slots directly connected to each processor. The NIC (10 GbE Intel NIC based on 82599 chip) is connected to a slot assigned to the first processor. Regarding the system storage, 12 Serial ATA-III disks that made up a RAID-0 are controlled by a LSI Logic MegaRAID SAS 2208 card have. These disks are Hitachi HUA723030ALA640 with SATA-3 interface and 3 TB of capacity. On the other hand, the operating system is an Ubuntu server 64-bit version with a 3.2.16 Linux kernel, with XFS filesystem—as preliminary experiments showed that is the best choice in terms of performance and scalability.

In order to inject traffic, we have developed a tool built on top of PacketShader's [HJPM10] API capable of: (i) generating tunable-size Ethernet packets at maximum speed, and, (ii) replaying PCAP traces at variable rates. For our experiments, we have used both synthetic and real traffic. Synthetic traffic consists of TCP segments encapsulated into fixed-size Ethernet frames, forged with incremental IP addresses and TCP ports. Note that synthetic traffic allows us to test worst-case scenarios in terms of byte and packet throughput, but they are not useful for testing the flow-related modules. The real traffic trace was sniffed at an OC192 backbone link of a Tier-1 ISP located between San Jose and Los Angeles (both directions), available from CAIDA [WAcAa].

In order to evaluate the performance of the flow-related modules, a key metric is the number of concurrent flows rather than the throughput in packets or bytes. Replaying the backbone trace at line-rate leads to a throughput of 9.59 Gb/s[1], 1.65 Mp/s, with a maximum of 2.25 million concurrent. All the results shown in this section have been obtained by replaying the corresponding traffic along a 10 minutes period.

First, we have assessed the performance of a simple packet sniffer application (as provided by `M3Omon`'s API) and the packet dumper thread. Table 5.1 shows the mean throughput and standard error of the mean when repeating the 10-minutes experiments 50 times, for both applications and for fixed-size line-rate synthetic traffic. The table shows that both applications only loose packets in the worst-case scenario (i.e., 60-bytes packets, as CRC is deleted at NIC level). It is also shown that above this packet size all packets can be success-

[1] This is the maximum achievable speed due to the preamble and inter-frame gaps that the Ethernet protocol requires.

fully captured stored into disk in a full-saturated 10 Gb/s link.

Packet size (bytes, CRC excluded)	Throughput (Gb/s) $\bar{x} \pm SE_{\bar{x}}$		
	Theoretical Max.	Packet sniffer	Packet dumper
60	7.14	4.92 ± 0.02	4.81 ± 0.02
64	7.27	7.27 ± 0	7.27 ± 0
128	8.42	8.42 ± 0	8.42 ± 0
256	9.14	9.14 ± 0	9.14 ± 0
512	9.55	9.55 ± 0	9.55 ± 0
1024	9.77	9.77 ± 0	9.77 ± 0
1514	9.84	9.84 ± 0	9.84 ± 0

Table 5.1: Packet sniffer and dumper modules performance when sending synthetic line-rate traffic (D=packet dumper, S=packet sniffer)

The effect of processor affinity on each module of M^{3}Omon has been studied as well. In our case, although the NIC is plugged into a PCIe slot attached to NUMA node 0, the intermediate packet buffer is allocated in node 1's memory. Thus, it seems reasonable that applications making use of that buffer benefit from being executed in node 1. Table 5.2 empirically shows this effect. The table shows the mean and standard error of the mean (through 50 experiment repetitions) for both system's throughput and packet loss when receiving the CAIDA trace at link-speed. Note that, as the dumper application accesses the packets in a byte-block fashion, it experiences a much lower performance decrease if scheduled in the wrong node as it minimizes memory accesses: 1% compared to the 35%-45% of packet lost when flow-related module's threads are not properly scheduled. Those results show the relevance of a proper processor scheduling policy in terms of system's performance. Table 5.2 also shows the performance obtained by the complete M^{3}Omon system, when all of the modules are scheduled in the optimal slot. In such case, the system experiences a packet loss of 1.1%, leading to a global throughput of 9.48 Gb/s.

Active Modules	Core schedule: NUMA node 0						NUMA node 1						Throughput (Gb/s) $\bar{x} \pm SE_{\bar{x}}$	Packet loss (%) $\bar{x} \pm SE_{\bar{x}}$
	0	1	2	3	4	5	6	7	8	9	10	11		
D	K						D						9.59 ± 0	0 ± 0
	K	D											9.41 ± 0.04	0 ± 0.1
P+E	K						P	E					9.59 ± 0	0 ± 0
	K	E					P						6.23 ± 0.05	34.2 ± 0.4
	K	P					E						4.99 ± 0.04	47.1 ± 0.3
	K	P	E										5.53 ± 0.04	41.6 ± 0.3
M^{3}Omon	K						P	E	D				9.48 ± 0.05	1.1 ± 0.2
M^{3}Omon + apps.	K	O_P	O_F	F	F	F	P	E	D	F	F	F	9.15 ± 0.06	4.7 ± 0.1
	K	O_P	O_F	F	F	F	P	E	D	S	F	F	8.97 ± 0.05	6.1 ± 0.3
	K	O_P	O_F	F	F	F	P	E	D	S	S	F	8.97 ± 0.07	6.2 ± 0.3
	K	O_P	O_F	F	F	F	P	E	D	S	S	S	8.87 ± 0.07	6.5 ± 0.4
	K	O_P	O_F	S	S	S	P	E	D	S	S	S	7.98 ± 0.05	15.3 ± 0.4

Table 5.2: Throughput and packet loss of the different modules in the system while receiving the CAIDA-trace sent at line-rate on a 10 Gb/s link (K=kernel sniffer, D=packet dumper, P=flow process, E=flow export, S=packet sniffer app., F=flow reader app., O_P=offline packet reader, O_F=offline flow reader)

Once M^{3}Omon's performance has been assessed, we proceed to analyze the effect of instantiating additional end-user applications on top. Table 5.2 shows the overall performance when instantiating two forensic (offline) applications –one for packets and one for flows– and using all of the available cores for real-time flow record processing. In such case, a mean packet loss of 4.7% is experienced with a mean throughput of 9.15 Gb/s. This packet loss is due to the extensive usage of the non-volatile volume of this scenario: the packet dumper writes packets, and the offline packet reader simultaneously accesses the same volume. Note that packet-oriented applications can only be executed in the same machine we are sniffing traffic from. This way, we have tested the system when adding a different number of packet sniffing applications, while keeping the slot for the two forensic applications, and instantiating real-time flow processing applications in the free cores. Table 5.2 shows that adding additional packet sniffling applications may degrade system's mean throughput down to 7.98 Gb/s. Such results show that end-users may instantiate several listeners, although we note that a scalable monitoring policy should prioritize MRTG and flow-based monitoring rather than packet-based.

In terms of CPU usage, the kernel-sniffing thread (K in the table) uses 99.9% of its core. This is also true for the flow processing thread (P) and the real-time packet sniffing threads (S). On the other hand, the packet dumper thread (D) uses 60% of a core, and flow exportation threads (E) use 32% of their core. Regarding the offline data access application (O_P and O_F), they are both executed with the lowest I/O priority, leading to a core CPU usage of 42% and 23% respectively. Both real-time and offline MRTG data access applications use less than 1% of one core, so they can be executed at any core not being fully occupied –and this is the reason why they do not appear in the table. With respect to memory consumption, the flow-pool elements used by the flow manager and exporter threads has been set so that the system supports up to 450K concurrent connections entailing a use of 8GB of memory. The rest of elements of M^{3}Omon use a negligible amount of memory compared to the previous one.

It is worth remarking that, due to the multicast export policy, the use of flow-based or MRTG-based applications does not affect system performance –regardless the NUMA node of choice. Interestingly, such applications can be executed on external machines if needed. In our test scenario, the total throughput for flow and MRTG exportation employs nearly 1 Gb/s, namely a considerable bandwidth is necessary for sending the flow/MRTG data through a distributed environment.

5.4 Industrial application samples

This section gives an overview of two industrial applications that have been developed and deployed on top of HPCAP: `DetectPro` and `VoIPCallMon`.

5.4.1 DetectPro

Let us illustrate the functionality and applicability of the above introduced framework `M`^{3}`Omon` with an example of a monitoring tool deployed in real bank networks. `DetectPro` leverages `M`^{3}`Omon` to monitor network traffic without being concerned about lower level tasks (packet capturing, flow building and statistic aggregation), focusing only in network analysis. That is, `DetectPro` exploits three-level grained traces to detect anomalous events (aggregated statistics), locate hosts/network segments/services involved in the anomaly (flow records) and discover the root cause behind the anomaly (packet traces), minimizing the human intervention and coping with multi-Gb/s rates.

`DetectPro` reads aggregate statistics to diagnose both short-term and long-term changes [MGDA12] and reports the corresponding alarms, including information of detected anomalies such as start and end times of the alarm as well as the values of bit/packet/flow rates that caused the alarm. When an alarm is triggered, `DetectPro` automatically starts inspecting flow records to extract information about the network activity during the alarm period. Hence, it generates several reports containing, for instance, the distribution of hosts/network segments/services with the highest byte, packet, flow counts as well as other metrics (e.g., TCP flags, retransmissions), and the largest flows in terms of packets and/or bytes. Finally, the application selects and inspects packet traces corresponding to the alarm period. Note that such traces are automatically selected with the information previously obtained —e.g., filtering packets belonging to the busiest host.

In the following, we describe two case studies requiring the exploitation of multi-granular features by `DetectPro` to fully characterize anomalies detected in real network traffic.

First, we analyze an anomalous event observed in the traffic from a large commercial network. On June 3rd, 2013 `DetectPro` automatically launched an alarm, that indicated an increment in the number of concurrent flows from 200K up to 300K in a couple of minutes, returning to the initial steady state. Figure 5.3 shows some of the outputs (concurrent flows, bytes and packets time series) during the anomalous event. We can observe that neither bytes nor packets time series show anomalies, unlike concurrent flow series. This suggests that

the number of connections had increased in this time interval but the increment in the involved bytes and packets was not relevant.

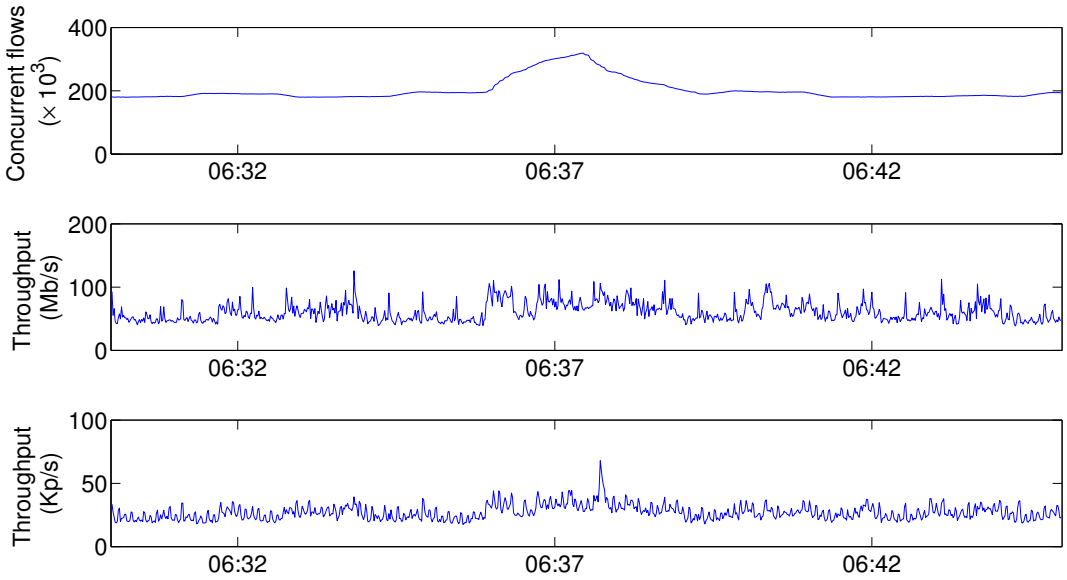


Figure 5.3: Time series for concurrent flows, bytes and packets during an anomalous event

Once the alarm was triggered, `DetectPro` accessed flow records obtaining several metrics and statistics which helped network manager to discover the traffic involved in the event. For instance, it reported the most active IP addresses and TCP/UDP ports in terms of concurrent flows during the time interval of the anomalous event. From such reports, it was observed that DNS (Domain Name System) connections to/from two given hosts represented the 90% of the total flows during the anomalous event. Then, network managers may use the packet inspection feature of `DetectPro` to select some packets of the anomalous connections and infer the root causes behind the problem. In particular, the problem was related to Mozilla Firefox browser's DNS queries (updates, markups, popup blocking) whose responses were not properly answered.

Secondly, we describe the use of multi-granular monitoring for the characterization of a phenomenon detected in CAIDA San Jose datasets [WAcAa]. Figure 5.4 shows three time series representing concurrent flows, bytes and packets by replaying the traces of October 18th and November 15th of 2012, in both traffic directions (A and B). While replaying the trace corresponding to November 15th, an alarm was triggered. In direction A, flow concurrence reaches peaks of more than 15 million flows per second, which is more than ten times the expected value. Similarly, in direction B, flow concurrence doubles the expected value.

As stated before, `DetectPro` uses flow records to identify the hosts in-

volved in this increase of flow concurrence. The results of this flow-level analysis reveal that hosts in the subnets represented as $40.10.0.0/16$ and $238.138.39.0/24$, in directions A and B respectively, generated a number of SYN-flag activated packets that exceeded in more than one order of magnitude those coming from the rest of the hosts discovered in these traces. Finally, network managers may use `DetectPro` to select some packets of such anomalous connections and finally assess that the packet pattern for such hosts corresponds to two SYN flood DoS attacks against web servers.

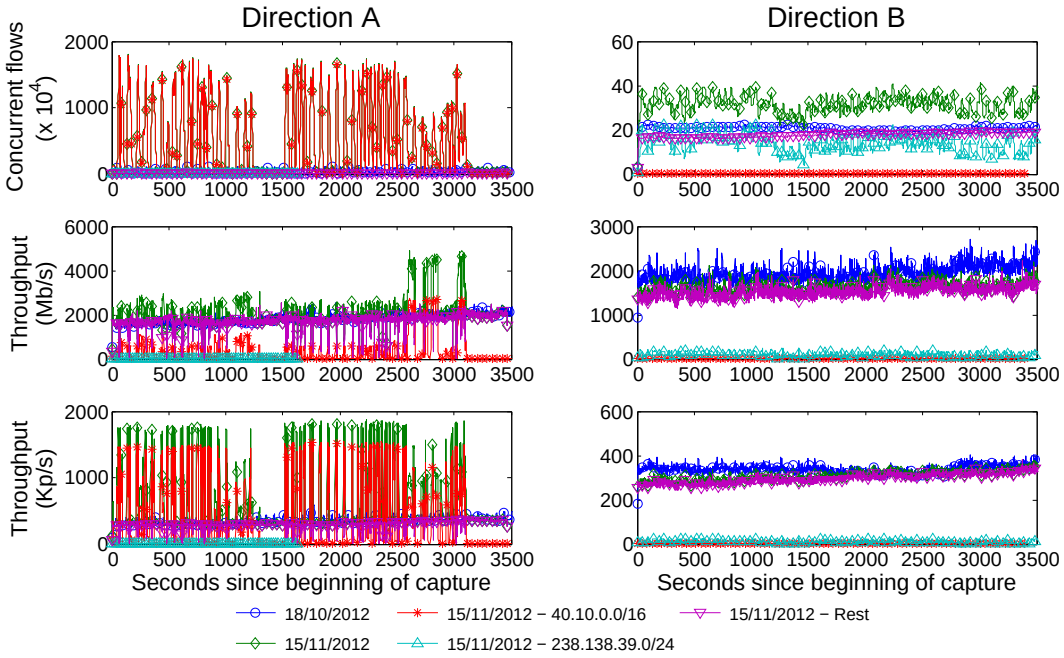


Figure 5.4: Flow concurrence and throughput in terms of Mb/s and Kp/s for both directions

5.4.2 VoIPCallMon

VoIP is increasingly replacing the old PSTN (Public Switched Telephone Network) technology. In this new scenario, there are several challenges for VoIP providers. First, VoIP requires a detailed monitoring of both users' QoS and QoE (Quality of Experience) to a greater extent than in traditional PSTNs. Second, such monitoring process must be able to track VoIP traffic in high-speed networks, nowadays typically of multi-Gb/s rates. Third, recent government directives require that providers retain information from their users' calls. Similarly, the convergence of data and voice services allows operators to provide new services such as full-data retention, in which users' calls can be recorded for either

quality assessment (call-centers, QoE), or security purposes (lawful interception). This implies a significant investment on infrastructure, especially on large-scale networks which require multiple points of measurement and redundancy.

In this context, we introduce `VoIPCallMon` as a novel methodology, architecture and system to fulfil the existing challenges in high-performance VoIP monitoring. As distinguishing features, `VoIPCallMon` provides very high performance being able to process VoIP traffic on-the-fly at high bitrates, novel services, and significant cost reduction by using commodity hardware with minimal interference with operational VoIP networks. In what follows, let us discuss the functionality of each of the `VoIPCallMon` modules, which are shown and labeled in top left part of Figure 5.5. `VoIPCallMon` software is written in C language over a Linux-based system:

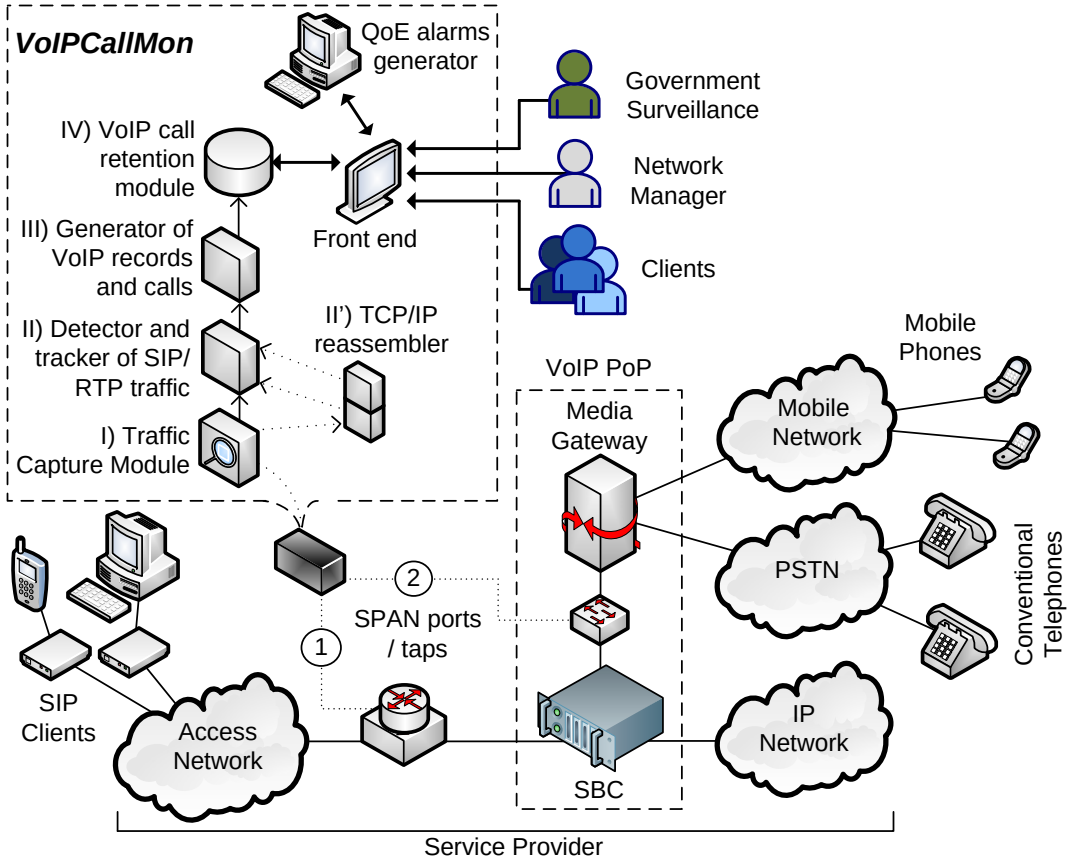


Figure 5.5: SIP VoIP network (bottom) and `VoIPCallMon` architectures (top)

- **Traffic capture module:** VoIPCallMon requires to keep track and correlate both SIP and RTP data flows but note that these flows do not share the same 5-tuple, and that, however, such tuples are used by RSS technology to distribute packets among different receive queues at NIC and

kernel level. This prevents the use of multiple receive queues in VoIP monitoring as a signalling flow (e.g., SIP) and its corresponding payload flow (e.g., RTP) may potentially end up to different queues and, thus, cores. Consequently, the capture process needed to be carried out in a way so that maximum performance was achievable with one single receive queue. Furthermore, the QoS metrics calculated for each call register involve calculations with the packet's timestamps, which needed to be as accurate as possible. For those reasons, the network capture tasks was carried out using the HPCAP driver.

- **Detector and tracker of SIP/RTP traffic:** Once incoming packets are captured, and via the M^3=Mon framework built on top of HPCAP, it is very easy to instantiate a new thread and feed it with the traffic obtained by the capture module. Note that, to achieve maximum performance, this new thread must be carefully scheduled so that it does not interfere with the capture process and so that it benefits from memory locality.
  In order to obtain useful information for each call, both the SIP and RTP streams must be carefully analysed. SIP packets contain important information to characterize a call such as the call identifier (call-ID), caller and callee identifiers (from and to fields), as well as information to establish the RTP sessions, essentially source/destination IP addresses and port numbers (4-tuple) and codec of the RTP stream. Once a RTP connection is established via SIP, the system begins to monitor the call's associated RTP stream for calculating its associated QoS metrics and to decode the voice streams. In order to manage all this information, there are two different hash tables used to index the calls data, so each record can be accessed from the two tables. The first table is indexed by SIP call-ID, which is useful when a call is updated by a SIP packet; whereas the second one is indexed by RTP 4-tuple, which is useful when a RTP packet is processed. Note that these two tables are necessary because RTP packets do not have call-ID or other SIP information, whereas SIP packets do not have the same IP addresses/ports as RTP packets.
- **Generator of VoIP records and calls:** When a call is expired, it is deleted from the detection module's active call list and hash-tables, and redirected to this third module which generates all the required information about the call. This includes the information required by the European Union directive 2006/24/EC [SG08]: the ID of both edges of the communication (from and to SIP fields), and the start and end times of the call. Furthermore, the module includes several flow parameters useful to assess the users' QoE for a given call such as the used codecs, count of packets and bytes, throughput, max/min/mean/standard deviation of both packet size and inter-arrival time, round-trip-time, jitter and packet loss rate. For a further thorough explanation of the calculated parameters the reader is

referred to [Sch12].

- **VoIP call retention module:** Periodically, the output of the previous module, i.e., calls records and pointers to the RTP files (if such service is required) are dumped to a MySQL database. Thus, clients, network managers or any other agent (i.e., law enforcement and intelligence agencies) may retrieve calls, measurements or alarms using a given key (e.g., the caller/callee ID, phone number/user name) via a front-end.

Figure 5.6 shows the number of active calls during a 30-minute experiment as well as the call generation rate in one of the case-study scenarios. The number of active calls gives more than 52,000 concurrent calls with a rate of 442 new calls per second in the stationary state, which in this case yields a VoIP rate of 8.9 Gb/s. Importantly, the amount of simultaneously active calls supported by a VoIP system can limit its performance if resource consumption and contention is not carefully analysed. Those results are far beyond the requirements from a Spanish VoIP operator with 15 points of presence across the country whose data we had access to: this operator managed between 60,000 and 100,000 calls during the busy hour whose mean call hold time is about 120 seconds. That is translated into between 17 and 28 calls per second and between 2,040 and 3,360 simultaneous active calls.

Importantly, in order to achieve performance levels that allow `VoIPCallMon` operate in high-speed networks, all modules and their interactions have been carefully optimized. For a deeper and more detailed description, we recommend reading [GDSdRR$^+$14]. Importantly, the system's performance has been assessed by testing each of the modules' performance independently, and once the proved multi-gigabit capabilities, the global system's performance was tested. The performance evaluation stated that `VoIPCallMon`'s limits are above 10 Gb/s, surpassing other systems presented in the literature, ergo merging low-cost and high-performance in a proposal.

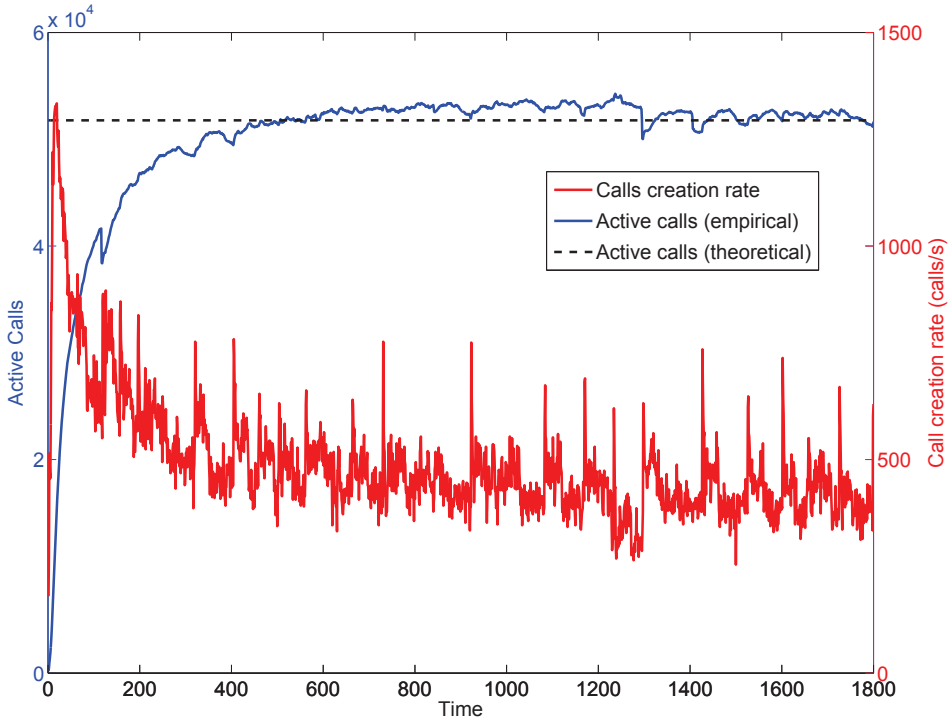


Figure 5.6: Active calls and new calls managed by `VoIPCallMon` during a 30-minute experiment

5.5 Related Work

In this section we present those systems that share with M^{3}Omon its target of facilitating the labor of network monitoring. Let us highlight in each case the offered flexibility, scalability, performance and kind of hardware required.

We now turn our interest out to systems and applications that provide a more elaborated set of measurements. We acknowledge Tstat [FMM+11], an open-source monitoring tool with more than a decade of development. Tstat captures traffic, classifies it with remarkable accuracy, and generates a number of statistics aggregates at different granularities. Similarly, the authors in [MSD+08] also state the importance of forensic analysis of traffic and highlight that is enormously helpful to store packet traces to inspect them in case any problem arises. They proposed a system, TM, that collects only the first packets of each connection of the traffic under study. The rationale behind this approach is that connections follow a long-tail distribution whereby a small set of connections accounts for most of the traffic and conversely there are many connections with very little traffic. Thus, by capturing only a few packets per connection the throughput and storage requirements are dramatically reduced. On the downside, many packets are ruled out, however the authors claim that the most useful packets for monitoring purposes are the first ones, therefore the savings in space and computational burden pay off the accuracy lost. In this regard, it is worth remarking that M^{3}Omon is compatible with the operation of TM, by simply modifying the store module of M^{3}Omon to follow the packet discarding rules that TM proposes.

By comparing these proposals to M^{3}Omon, we note that neither Tstat nor TM are a framework or architecture for multi-granular and multi-purpose monitoring, and their performance are far from multi-Gb/s rates. Although Tstat highlighted the importance of multi-granular monitoring, it does not give any support to the applications running over. It leaves data available to be accessed later in an off-line fashion, being the developer who accesses the data and builds such application from the ground up. Although TM does provide user with traffic packets by simple mechanisms (specifically a tuned database), there are neither references to any richer set of data granularities nor multi-purpose deployments. By paying attention to performance, both Tstat and TM may work over commodity hardware, but they lack of optimization at packet reception level and their performance would be limited to 1 Gb/s rates.

Following a completely different approach to all the previous presented works (including ours), the authors in the technical report presented in [ČKB+10] entrusted dedicated hardware with the task of high-performance monitoring. In particular, they developed a FPGA-based design name HAMOC. Similarly to Tstat and TM, HAMOC does not focus on exploiting the multi-purpose capacities of a monitoring system as M^{3}Omon does. Remarkably, HAMOC offers a rich set of ap-

plications modified to work over their FPGA-design, but such applications' works are isolated without sharing efforts to pre-process data. In contrast, M^{3}Omon features a driver that allows several threads to access to the same packet distribution queue when typically drivers permit the opposite, distributing traffic to different queues accessed each of them by a unique thread. Certainly, HAMOC pays attention to the generation of flow records at high-speeds, but there is no reference to packet storage, any other broader statistics or framework to access such data. Turning to similarities, HAMOC does share with M^{3}Omon multi-Gb/s performance rates. HAMOC showed rates close to 10 Gb/s both in the subtask of packet capture and application layer.

Regarding the process of coding monitoring applications, we emphasize Blockmon [dPHB+13b]. Blockmon allows building a monitoring application out of blocks, where each of them represents a different subtask in a given application. For example, reading a PCAP trace or filtering traffic at 4-layer may be subtasks for an application that inspects DNS traffic. Thus, Blockmon may ease the development of multi-granular applications over our proposal. That is, an application that correlates data at different granularities may be written as a sequence of Blockmon's blocks with all the support that Blockmon's library offers. The application examples provided with Blockmon use only packet-level information and run over PFQ capture engine. Using multiple queues, the system is able to process about 5 Gb/s in the worst-case scenario of small packet size—about 15% less compared to PFQ's capture process [dPHB+13b].

As examples of successful implementations of final applications, we remark some related to traffic classification [GES12, SdRRG+12] and NIDS [JLM+12, VPI11], which are able to process the incoming traffic at 10 Gb/s rates. All of these proposals and those forthcoming may benefit from the flexibility and scalability that M^{3}Omon offers to the applications running on top.

5.6 Conclusions

We have proposed a novel monitoring system architecture that provides network managers and network analysts with mechanisms to deploy more flexible, scalable, and affordable monitoring applications over high-speed networks.

The design is based in five key aspects which in turn comprise the contributions of this work: (i) a novel and optimized layer between the traffic sniffer and the monitoring application themselves that provides advanced characteristics (M^{3}Omon), (ii) a framework to access multi-granular data, (iii) an improved NIC driver (HPCAP) for monitoring purposes, and all of them (iv) running as an off-the-self system at (v) high-speed.

M^{3}Omon allows applications to run in parallel reading traffic at different data granularities—time-series aggregates, flow records and packet traces. Such granularities have been constructed only once, to be shared by the set of applications running over M^{3}Omon. We have termed this as multi-granular and multi-purpose features, and they provide outstanding flexibility and scalability over the state-of-the-art.

Such features are easily exploited thanks to the provided framework, and viable thanks to an improved NIC driver as well as low-level hardware interactions, efficient memory management tuning, and programming optimization. This work has comprehensively explained all these implementation details so that it results useful for both network managers and analysts in their duty to develop novel multi-granular monitoring tools. In this regard, we have shown a real monitoring application, DetectPro, which illustrates its successful exploitation in production environments in real networks.

To conclude, we remark that our system runs over commodity hardware as open-source software available under an open-source license [Hig13], which gives additional adaptability, scalability, and cost-aware developments. That is, commodity hardware is easily upgraded over time and replicated over the infrastructure, and its cost is lower than dedicated solutions. High-speed is the other characteristic of our proposal, we note that the performance evaluation has shown that it reaches multi-Gb/s rates. We believe that our proposal paves the way for future migration of high-performance tasks to off-the-self systems.

6 Network monitoring in virtualized environments

On this chapter, we assess the feasibility of moving high-performance network processing tasks to a virtualized environment. For such purpose, we analyse the possible configurations that allow feeding the network traffic to applications running inside virtual machines. For each configuration, we compare the usage of different high-performance packet capture engines on virtual machines, namely PF_RING, Intel DPDK and HPCAP. Specifically, we obtain the performance bounds for the primary task, packet sniffing, for physical, virtual and mixed configurations. We also developed HPCAPvf, a counterpart of HPCAP for virtual environments, and made it available under a GPL license.

The maturity of the telecommunication market plus the emergence of new services and applications have lead to a harsh competition between the different actors involved. The amount of services and applications available to end-users makes it necessary for those services' providers to deploy quality-assessment policies in order to distinguish their product among the rest. As a consequence, network processing and analysis becomes a central task that demands processing elements capable of reaching a humongous amount of data rates while keeping the cost as low as possible.

In order to deal with the network processing tasks some solutions based on specialized hardware have been developed. Those solutions are traditionally built on top of FPGAs, Network Processors or even ASICs. These solutions answer the high-performance needs for specific network monitoring tasks, e.g. routing or classifying traffic on multi-Gb/s links [YKL04, FSLB+14]. However, this approach causes operators' amenities being populated with a huge amount of heterogeneous hardware boxes, which complicates and raises the price of maintenance processes. The scalability of such an infrastructure is limited, as it involves adding new hardware boxes which has a set of constraints in terms of physical space, cooling or power consumption. Additionally, those solutions im-

ply high investments: such hardware's elevated cost rises CAPEX, while OPEX are increased due to the difficulty of their operation, maintenance and evolution if new services arise.

Off-the-shelf systems, the combination of commodity hardware and open-source software, have emerged as an alternative to reduce the limitation imposed by specialized hardware [GDMR+13]. The advantages of those systems lay in the ubiquity of their components, which makes it easy and affordable to acquire and replace them. Those systems are not necessarily cheap, but their wide range of application allows their price to benefit from large-scale economies and makes it possible to achieve great degrees of experimentation. Furthermore, such systems offer extensive and high-quality support. However, the increased demands for network processing capacity would be translated into a big number of machines, probably from different vendors, which complicates and raises the price of maintenance processes and empowers the appearance of interoperability issues.

All those maintenance and scalability issues damage the profitability that networked service providers may experience when applying either specialized hardware or off-the-shelf systems to carry their network processing tasks. Looking for a solution, industry and academia have turned an eye to virtualized solutions [YHB+11]. The evolution and ubiquity of virtualization techniques along the past years have proven to play a fundamental roll in terms of expenditure reduction, as they allow providing a unified homogeneous layer on top which new services and applications could be deployed.

Virtualization technology has been improving and developed so much the last years that network processing on virtual machines are close to reach the same performance than on physical ones. The scope of applicability is enormous and there are many potential advantages. The most evident is increasing the infrastructure's profitability due to a reduction in the equipment investment by acquiring large-scale manufacturers' products that may host several virtual elements. This policy entails a reduction in the amount of physical machines required, and thus saving in terms of physical space and power consumption. Another relevant advantage of virtualization techniques is the possibility of dynamically adjusting network applications and resources based on specific clients' requirements. Besides, virtualization contributes to open the market for small companies and academia by minimizing risk and thus encouraging innovation. For all those reasons, network manufactures have been working during the last years on the development of the concept of Network Function Virtualization (NFV) [Net14b]. This new paradigm aims to unify the environments where network applications shall run by means of adding a virtualization layer. This novel philosophy also allows merging independent network applications using unique hardware equipment, thus reducing the amount of physical machines re-

quired, speeding up network applications maturation cycle, easing maintenance procedures and expenditures, and rapidly adjust network applications and resources based on specific clients requirements. The development of this novel NFV philosophy has been favored by other trending technologies such as Cloud Computing and Software Defined Networking (SDN) [AFG+10, MRF+13].

However, in order to make the most of NFV, mechanisms that allow obtaining maximum network processing throughput in such virtual environments must be developed. In a bare-metal scenario, researchers have recently focused on developing high-performance packet capture engines [GDMR+13]. This work assesses the feasibility and provides performance bounds when moving high-performance network processing tasks to a virtualized environment. As our goal implies fetching packets from high-speed networks and at least storing them in non-volatile volumes, this work has focused on optimizing the communication of both network and storage devices with the corresponding virtual machines. We evaluate different virtualization alternatives, namely PCI passthrough and Network Virtual Functions (NVF) available on state-oh-the-art systems. From the network side, we have evaluated the use of already existing high-performance packet capture engines. In terms of non-volatile storage we have translated the experiences we had with physically connected RAID volumes to the virtual case. Additionally, we have discussed the details and performance results for instantiating a network virtual probe and a network monitoring agent, so researchers and practitioners may benefit from our results in order to build their high-performance NFV-based applications.

The rest of this chapter is structured as follows: first, we discuss the state of the art of packet capture and storage systems in section 6.1. Second, we explain diverse virtualization approaches for I/O devices in section 6.2. Section 6.3 discusses how to create a virtual probe with the knowledge acquired in the previous sections, while section 6.4 presents the instantiation of a virtual network-monitoring agent. Finally, section 6.5 presents the conclusions and the future roadmap generated by this work.

6.1 High-performance network processing

The need for more flexible and profitable network processing equipment have made both industry an academia to pay attention to off-the-shelf systems [BDKC10], understood as the combination of commodity hardware and open-source software. The inherent nature of those systems supply some interesting characteristics such as high flexibility, reduced CAPEX due to the benefits

of manufactures' page-scale economies, and minimal OPEX thanks to the extensive documentation and functional testing carried out by the corresponding vendors. The NIC vendor's driver plus a network stack traditionally compose an off-the-shelf system applied for network processing tasks. This approach provides a high degree of flexibility, as the network stack is made up of independent layers that allow distributing the traffic to the corresponding final applications. However, this approach fails in terms of performance, as each incoming packet has to traverse a subset of the existing layers, meaning that additional copies and resource re-allocation are required. Thus, this flexibility the standard solution offers limits its applicability in high-speed scenarios.

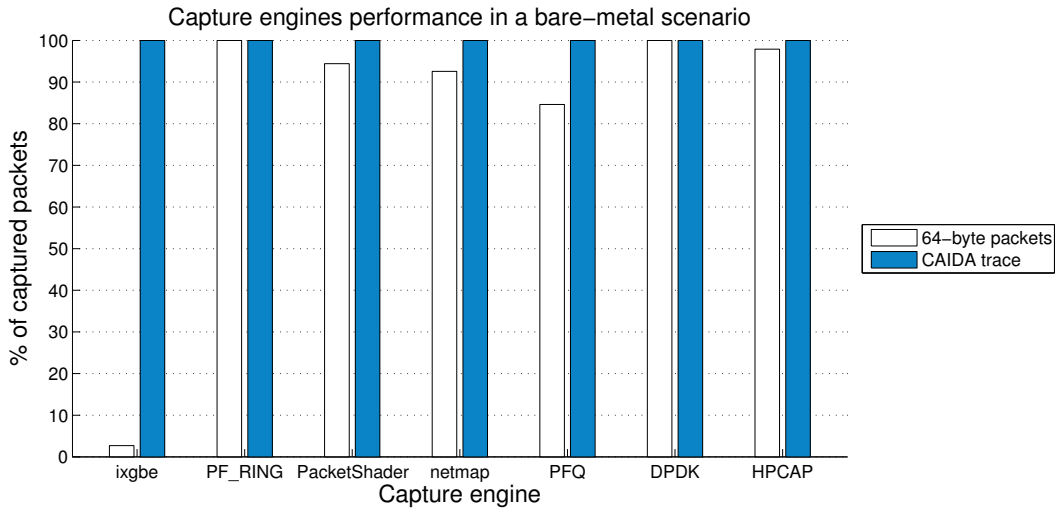


Figure 6.1: Percentage of packets captured for different traffic patterns in a fully-saturated 10 Gb/s link obtained by different solutions

Fortunately, if they are carefully scheduled and tuned, off-the-shelf systems are still eligible for operating in high-speed networked scenarios. With this is mind, some implementations have been developed which are referred as high-performance packet capture engines in the literature [GDMR+13, Mor12, Int14b]. Solutions such as PF_RING, PacketShader, netmap, PFQ, Intel DPDK or HPCAP were created as high-performance counterparts for the traditional network driver-plus-stack alternative. All those capture engines reach high processing rates due to the application of diverse optimizations along the capture process. A detailed description of those optimizations and the ones used by each capture engine is included in Chapter 3. In a nutshell, some of those optimizations are memory pre-allocation and efficient reuse, prefetching network packets' data into the system's caches, and circumventing the traditional network stack in order to exploit parallelism. Specifically, PF_RING, PacketShader and Netmap achieve wire-speed using few cores (even with only one reception queue), but they lack

of flexibility. At the same time, HPCAP obtains comparable results while additionally provides accurate timestamping [MSdRR+12], being designed for optimizing the packet-storage process [MSdRR+14b] and providing a multi-purpose multi-granular monitoring framework [MSdRR+14a]. Importantly, those additional features force HPCAP to uses two CPU cores per receive queue, while the other engines use one core per queue. Conversely, PFQ needs a larger number of queues and cores to achieve high throughput, but provides the biggest flexibility due to a customized packet aggregation and delivery mechanism.

Fig. 6.1 shows the performance level obtained for packet capture in a full-saturated 10 Gb/s link for the worst-case scenario (64 byte packets) and an average scenario (a CAIDA trace from a ISP's backbone link between Chicago and Seattle obtained the 19^{th} June 2014, with an average packet size of 965 bytes [WAcAa]). All the performance tests presented along this chapter have been made replaying this traffic in 30-minutes experiments. Specifically, all tests have been carried out using a server with two Xeon E5-2630 processors running at 2.6 Ghz and 32 GB of DDR3 RAM running a Linux Fedora 20 with a 3.14.7 kernel. The NIC used was an Intel card with 82599 chipset connected to a PCIe Gen3 slot. The traditional approach, represented by the default `ixgbe` Intel driver in addition to the de facto standard PCAP library has also been included in the comparison. The results obtained show that all capture engines are capable of capturing 100% of the packets when replaying the CAIDA trace at top-speed. In the worst-case scenario, `ixgbe` captures only 2.7% of the incoming packets, while PF_RING and Intel DPDK capture 100% of them, HPCAP captures 97.9% of the packets and the rest solutions obtain slightly lower results.

Nevertheless, come studies have proven that packet capture may result insufficient for an effective network analysis. The authors in [MSD+08] highlight the relevance of storing network for a later in-depth analysis. Even an intuitively simple task such as sniffing and storing the traversing traffic is a challenge when dealing with 10 Gb/s speeds or higher due to the great amount of resources and computational power needed. In this light, it is natural to focus on how the already mentioned capture systems behave when storing the captured traffic. Among all the capture solutions mentioned, only PF_RING and HPCAP have taken into the account the packet storage task. Their corresponding applications, `n2disk` and `hpcapdd` respectively were studied and tested in [MSdRR+14b]. Results for optimal disk throughput configuration in a bare-metal scenario are presented in Table 6.1. This work also discussed and tested how to tune a storage device to reach multi-Gb/s write data rates.

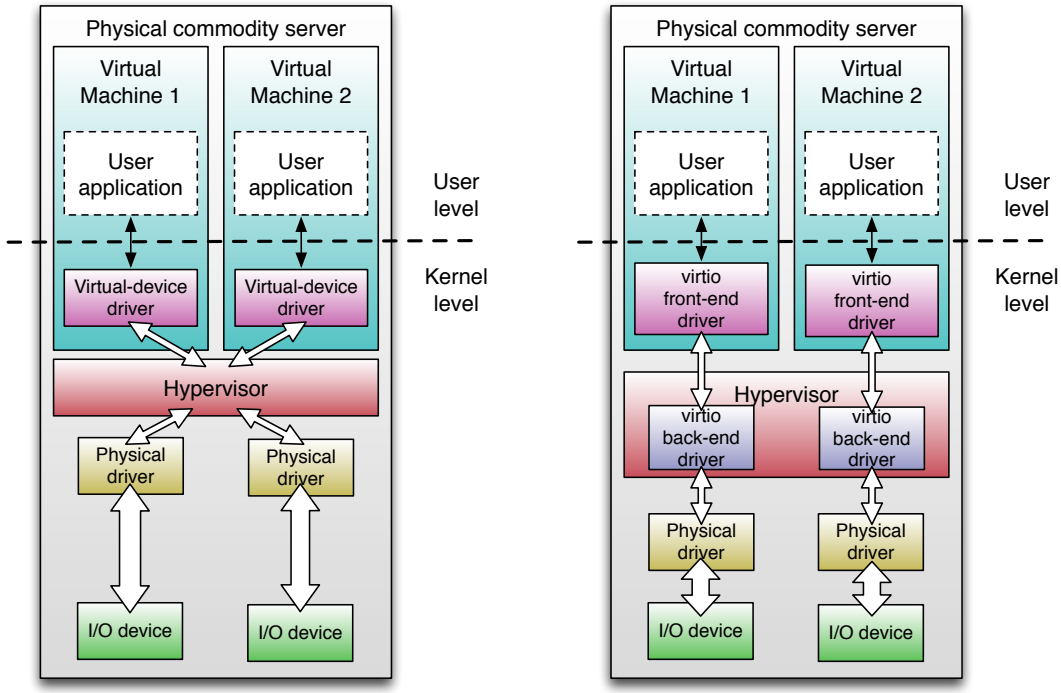
All the above-mentioned high-performance capture and storage solutions enable network managers to deploy high-performance network applications on top of them. Nevertheless, the application of the existing packet capture engines has been limited in the literature to the bare-metal case. That is, a physical

server to which the NIC is connected through a PCIe slot as shown in Fig. 6.3(a). With this in mind, this work focuses on the study on how to effectively migrate already high-performance network processing solution from bare-metal to virtual scenarios. Specifically, we will focus on the use of Intel DPDK and HPCAP, as they are the only two packet capture solutions that support working with NIC's virtual functions. That involved, in the case of DPDK, developing a packet storage solution as none had been already built on top of it.

6.2 Virtualized environments and I/O processing

The use of virtual machines imposes a computational overhead on any application being executed on top of them. For this reason, if VM (Virtual Machine) are used for computing intensive applications, the performance obtained by the target application is usually damaged. If we desire to obtain maximum performance, the creation and schedule of each VM must be carefully made. First, the amount of cores of the VM should be such that allows the VM to be executed inside a single NUMA node in the physical server, and those virtual cores should be configured to be mapped to independent physical cores. Second, the way a VM accesses the host's physical memory can be properly configured in order to reduce the amount of page misses and memory swapping. Most contemporary VM hypervisors support execution of their virtual machines with their memory attached to a certain NUMA node. Moreover, the authors in [KKA13] highlight the how use of Linux's huge pages for allocating the VM's memory chunk improve the VM's performance. By using Linux huge pages both the packet capture engines and the network applications built on top of them running on a VM would experience a performance increase. Finally, if the host's hardware configuration supports it, as it was our case, virtualization enhancement instructions should be enabled. All the VMs used along the performance experiments presented in this chapter were created taking these facts into account. Regarding the operating system running inside the VMs, we used the same version as in the physical server, a Linux Fedora 20 with a 3.14.7 kernel. The choice of the operating systems to be used both in the physical and virtual machines is relevant, as not all combinations support the use of Linux huge pages both in the physical server for creating the VM as mentioned and inside the VM. Note that some of the capture engines used such as PF_RING and Intel DPDK make use of those huge pages, so this requirement is nontrivial.

However, not only memory and computational configurations are relevant when running a network application on a virtual machine. Network applications have high I/O data-transfer requirements. An application processing the traffic coming from a 10 GbE link may have to process up to 14.88 Mp/s in a worst-case scenario. Thus, I/O configuration becomes critical for this kind of applications. That is even worse when writing the incoming data into a non-volatile storage device, such as a set of hard drives. In that case the storage device has to be able to consume up to 10 Gb/s in the storage's worst-case scenario. The rest of this section discusses diverse alternatives for mapping the I/O devices into virtualized environments. Note that most of the concepts exposed along this section apply to any I/O device regardless they are network, storage, etc.



(a) Full-virtualization (b) Paravirtualization

Figure 6.2: Mapping an I/O device using full-virtualization and `virtio`'s paravirtualization

6.2.1 Full-virtualization

Providing a hardware-independent abstraction layer that allowed virtual devices or machines to operate regardless the underlying hardware configuration motivated the idea of virtualization. In line with this, the natural way a physical I/O device is used from a VM also has a virtualization layer. Thus, the VM has access to a virtual device, which is necessarily managed by a specific driver usually provided by the virtualization solution manufacturer. The VM will then access this virtual device completely unaware of its virtual nature. For each I/O request received by the virtual device, its driver will forward such request to the hypervisor. The hypervisor will then process the request and communicate with the corresponding physical device to attend the VM's request. Note that the hypervisor carries out a resolution process to match which of the available physical devices was the request directed to. This configuration is depicted in Fig. 6.2(a).

For example, when a network device is fully-virtualized, incoming packets are captured by the physical's NIC driver running on the physical server and traverse the system's network stack before being delivered to the hypervisor's network module. Once acquired by the hypervisor, packets are delivered to the target VM depending on the virtual network configuration. Note that this data path requires at least two additional copies: from the physical server to the hypervisor and from the hypervisor to the virtual machine, which obviously degrades the performance achieved by the capture system. The performance degradation experienced by network applications running in this configuration has pushed academia and industry to tune and to optimize the hypervisor's packet handling policy. In this line, authors of [HRW14] introduce NetVM, a module for speeding up hypervisor's packet forwarding mechanism based on the use of Intel's DPDK. This work obtains promising results when performing packet forwarding between virtual machines instantiated in the same physical server, but does not deal with the problem of sniffing traffic from the wire.

Note that when using a fully-virtualized I/O device, the hypervisor is the central communication element. Consequently, the hypervisor becomes the bottleneck and a single point-of-failure for network processing tasks, limiting their applicability for I/O intensive applications.

6.2.2 Paravirtualization and VirtIO

Paravirtualization emerged as an alternative to the full-virtualization approach. This philosophy is similar to the full-virtualization approach in terms of having both front-end and back-end drivers instantiated at guest and hypervisor's levels respectively. The main difference is that the guest's operating system is

aware of the corresponding devices being virtualized, while in a full-virtualization approach the guest is unaware of this condition. Furthermore, front-end and back-end drivers communicate using an efficient high-performance approach. `Virtio` [Rus08] has emerged as the *de* facto standard for this communication between guest and a KVM hypervisor. Note that KVM-based hypervisors offer a virtualization layer that is aware of the underlying hardware [KKL+07], in contrast with traditional virtualization approaches. Specifically, `virtio` implements a flexible API for this communication via a set of queue structures and callback function handlers. This configuration is represented in Fig. 6.2(b).

`Virtio` supports different kind of I/O devices such as block, console or PCI devices. In the recent years, support for networking devices has also been added [MW12]. Furthermore, authors of [RLM13] introduce a paravirtualized extension for devices managed by the `e1000` 1 Gb/s driver and the corresponding adaptations at hypervisor level in order to improve the network processing performance. By applying their proposals they leverage the system throughput: from 300 Mb/s using the conventional approach to nearly 1 Gb/s using a VALE software switch in the worst-case scenario (64 byte UDP packets); and results from 2.7 Gb/s up to 4.4 Gb/s when transmitting TCP traffic between VMs in the same physical server. Although the good results obtained by this approach, it still far from our 10 Gb/s goal. For this reason, we have not taken network device virtualization using `virtio` into account for the traffic capture process.

On the other hand, the promising results obtained by `virtio` when applied to data storage drivers lead to take it into account for the traffic storage process. Specifically, we have compared the write throughput obtained when using a RAID-0 volume with different number of disks for the bare-metal and `virtio` approaches. Our RAID-0 array is composed of high-end mechanical hard disks, Hitachi HUA723030ALA640 with SATA-3 interface and 3 TB of capacity. The results obtained in those tests are shown in Table 6.1 when varying the amount of disks. The RAID configuration parameters vary with the number of disks in the array, and as being out of the scope of this chapter we refer the readers to Chapter 4.3 for a detailed discussion. Results show a slight improvement when instantiating the RAID-0 array in the VM using the `virtio` driver. We justify those unexpected results due to the write-requests buffering and scheduling policies that the `virtio-blk` driver performs [BYBF+12].

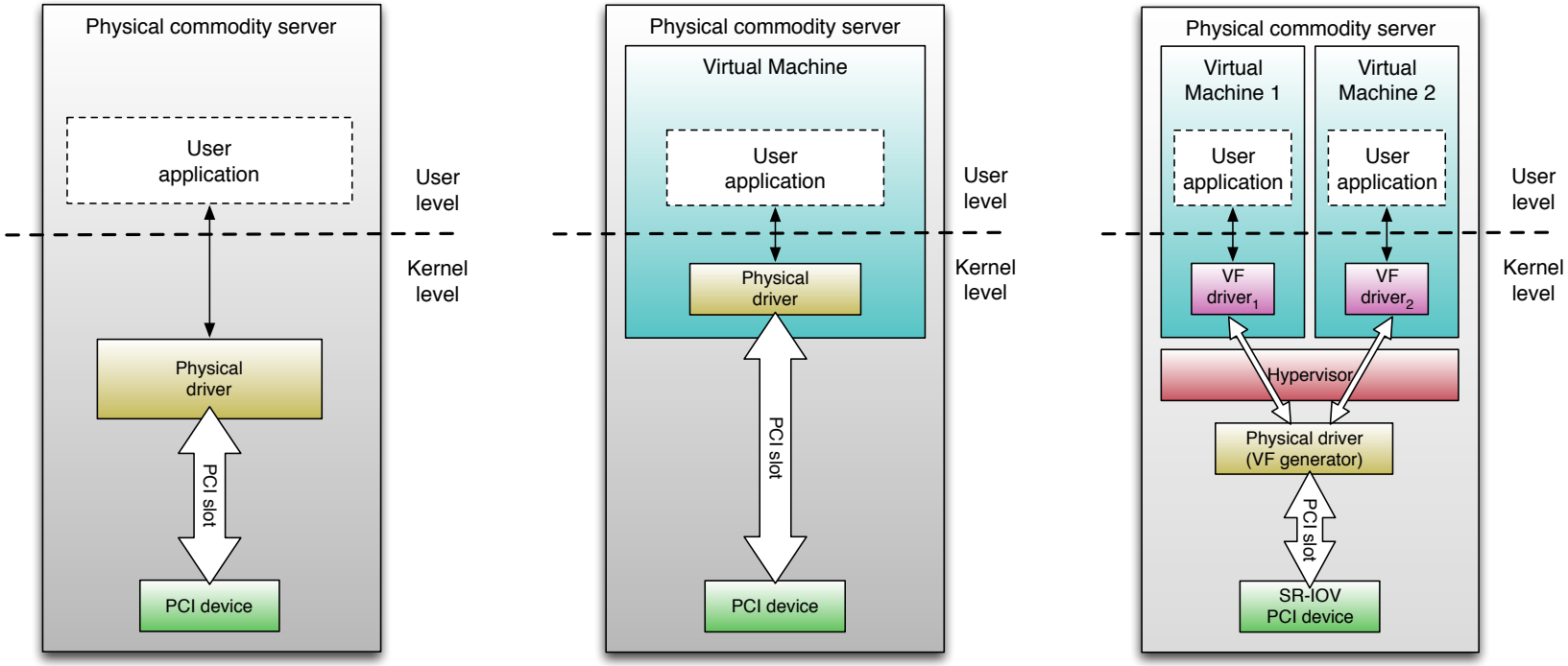
Keeping in mind that the goal of this work is processing 10 Gb/s traffic in VMs, we show the numerical results of the write throughput experiments in Table 6.1. Specifically, we show average throughput obtained and the confidence interval for the mean write throughput with a 0.99 level of significance. In order to prevent packet losses, we look forward to a combination of disks such that the confidence interval for the mean is above our 10 Gb/s goal, so a properly sized buffer will suffice to keep the system rate constant. Results obtained for 1

disk are near to the 1.25 Gb/s rate offered by the manufacturer. If performance scaled linearly, an amount of 8 disks would suffice to reach our target rate. However, although the scaling is very near linear, the results show that 8 disks do not meet our requirement for the mean confidence interval, and thus we conclude that an amount of 9 disks is required for both the bare-metal and the `virtio` configurations.

Number of disks	**Mode**	**Throughput (Gb/s)**	
		Average	**Conf. interval** (α=0.01)
1	BM	1.27	(1.27, 1.28)
	VirtIO	1.29	(1.25, 1.34)
	PT	1.27	(1.27, 1.28)
8	BM	9.93	(9.89, 9.96)
	VirtIO	9.96	(9.79, 10.13)
	PT	9.91	(9.85, 9.96)
9	BM	11.16	(11.09, 11.23)
	VirtIO	11.21	(11.03, 11.39)
	PT	11.15	(11.09, 11.21)

Table 6.1: Write throughput summary results for RAID 0 configuration in a bare-metal (BM) and virtualized (via PCI passthrough, PT, and VirtIO) scenarios

An interesting feature that paravirtualized devices have is that as the mapping in the VM is done by accessing a character/block device in the physical server, several VM could simultaneously access the same device and thus share the contents of the paravirtualized I/O. However, this must be made with caution as no access protection mechanisms are provided and data corruption may arise if no synchronization mechanism is used. For example, a paravirtualized RAID volume could be shared between two VM if exclusively one of them writes data into the volume while the second only accesses for reading purposes.



(a) Bare-metal (b) PCI passthrough (c) PCI-VF passthrough

Figure 6.3: Using a I/O device in bare-metal scenario (leftmost), and in a virtual machine using both physical and virtual functions' PCI pass-through

6.2.3 PCI passthrough

Most important hardware vendors such as Intel, AMD and ARM, implement I/O memory management units (IOMMU) and a set of techniques for I/O management. The name of these techniques may vary from one vendor to other (VT-d for Intel and ARM, Vi for AMD). Those features supply the protection and support required form virtual machines to safely access those memory regions corresponding to a certain I/O device, which is also referred as I/O passthrough or PCI passthrough when applied to PCI devices. The case in which a PCI device is mapped by a VM using PCI passthrough is depicted in Fig. 6.3(b). By using PCI passthrough, the access from the VM to the device presents a minimal overhead, as all intermediate virtualization layers disappear. Note that, in this configuration, the VM sees the device just as they were physically connected, which implies that the driver used must be the same one that manages such device in a bare-metal configuration. Consequently, this allows network applications being executed in virtual machines to benefit from the high-performance packet capture solutions developed for bare-metal scenarios. The application of PCI passthrough has been successfully applied in high-performance computing scenarios [YLWH14].

As our goal is sniffing and storing network packets at maximum rates, two potential application points for PCI passthrough appear: the NIC and the RAID controller. Specifically, write throughput performance results obtained when mapping the RAID controller card in a VM via PCI passthrough were shown in Table 6.1. Note that, in this configuration, the RAID volume is directly managed by the VM. Results show that mapping the RAID controller via PCI passthrough provides write throughput comparable to the bare-metal scenario. Importantly, although `virtio` results may present a slightly higher average throughput, they also present a higher variance as shown by the 0.99 confidence intervals for the mean.

Capture engine	% of packets processed			
	BM		PT	
	64 B	CAIDA	64 B	CAIDA
ixgbe	2.7	100	1.9	62.7
PF_RING	100	100	100	100
DPDK	100	100	100	100
HPCAP	97.9	100	85.2	100

Table 6.2: Percentage of packets captured for different traffic patterns in a fully-saturated 10 Gb/s link obtained by different solutions in a bare-metal (BM) and PCI passthrough (PT) configuration

Alternatively, Table 6.2 shows the performance results when capturing the

incoming packets in a virtual machine with the NIC mapped via PCI passthrough. Those results were obtained when capturing traffic from a fully-saturated link in both the worst-case (with minimal-sized packets) and an average-case, using the default `ixgbe`, PF_RING, Intel's DPDK and HPCAP drivers. The amount of packets captured by `ixgbe` falls from 2.7% under the bare-metal configuration to 1.9% using PCI pass-through in the worst-case scenario, and from 100% to 37.3% when replaying the previously mentioned CAIDA trace. On the other hand, PF_RING and Intel DPDK show no performance degradation when used via PCI passthrough, as they capture 100% of the packets on all scenarios. When it comes to HPCAP, packet capture performance is damaged when using PCI pass-through in the worst-case scenario, in which the amount of packets captured falls from 97.9% to 85.2%. The performance penalty experienced when introducing PCI passthrough can be blamed on the execution of the capture software over a virtual machine.

6.2.4 PCI virtual functions

Note that, using PCI passthrough for mapping I/O devices to VMs has an inherent constraint: only one VM can make use of each mapped I/O device. Thus, PCI passthrough presents a scalability problem when increasing the amount of VMs in use that can only be solved by adding additional PCI devices which is also limited, as the amount of PCIe slots a server has is limited. With the goal of promoting virtualization performance and interoperability the PCI Special Interest Group developed a series of standards, which they called Single-Root I/O Virtualization (SR-IOV). Those standards use the term PF (Physical Function) when referring to a PCI device connected to a physical PCI slot. They also introduce the concept of VF (Virtual Function) as a way for PCI devices to offer a lightweight register interface for managing part of the data interchanged with the physical PCI device. A VF is a lightweight PCI device, that is, it has an I/O memory area assigned to it with a set of control-related registers that allow the end system to use the VF as a regular PCI (usually with a reduced functionality). Importantly, the driver managing this new virtual device is usually different from the one managing the corresponding physical device, as it must be aware of the peculiarities the virtual device presents. After creating the corresponding VF, they could be mapped by a VM via PCI passthrough just as if they were purely physical devices. This configuration is represented in Fig. 6.3(c). A relevant advantage of VF over PF is that, depending on the underlying hardware, a single PF can be attached to a number of VF. This behavior allows the system manager to solve the scalability issue by attaching new virtual machines to new VF without needing to increase the amount of hardware in the system.

A device supporting this feature is Intel's 82599 and its more recent versions,

but we had no access to a RAID controller card supporting VF when we carried out our experiments. Those NICs refer to their VF by introducing the concept of Virtual Machine Device Queues (VMDq), and have a 1-to-1 correspondence with the VF instantiated. Through a set of hardware registers, the system manager can configure the way the incoming traffic is distributed or replicated to each VF. Note that this configuration allows connecting an arbitrary number of VMs to a single physical device. The amount of virtual functions generated per physical device is limited by the hardware device, being this limit 32 for our Intel 82599 adapter. As mentioned above, only VF-aware drivers can be used to manage the NIC's VF, which limits the amount of capture engines available. Specifically, Intel's DPDK has native support for working with VF, and they also supply a VF-aware counterpart to the `ixgbe` driver, named `ixgbevf`. Additionally, we developed a VF-aware version of HPCAP, that we named HPCAPvf, following all the design principles that guided HPCAP's design [MSdRR+14a]. Those three drivers have consequently been the only ones we have been able to test for their use with VF.

VF generator	Capture Engine	% of packets captured
ixgbe	ixgbevf	0.1
	DPDK	37.6
	HPCAPvf	36.3
DPDK	ixgbevf	1.0
	DPDK	100.0
	HPCAPvf	82.7

Table 6.3: Packet capture performance obtained when capturing from a 10 Gb/s link fully-saturated with 64-byte packets using VF for different VF-generators

On the other hand, the task of creating and managing the VFs belongs to the driver managing the physical device. Above all the drivers previously mentioned capable of managing a physical NIC, only Intel's ones offer this feature. Consequently, when using VF only `ixgbe` and Intel DPDK are eligible. Importantly, the choice among those two drivers for generating the VF has an impact on the performance obtained by possible network applications running inside the VMs. Table 6.3 shows the effect of such choice when using different VF-aware for packet capturing in the worst-case scenario, that is a fully-saturated 10 Gb/s with 64-byte packets. Results show that using DPDK as VF generator improves the packet capture performance obtained from the VM side compared to the performance when using `ixgbe` for generating those VF. Specifically, when capturing packets in a VM using the `ixgbevf` driver using DPDK as VF generator raises the amount of packets captured from 0.1% to 1%. In a similar manner, when using DPDK and HPCAPvf in the VM side with `ixgbe` as VF generator, only

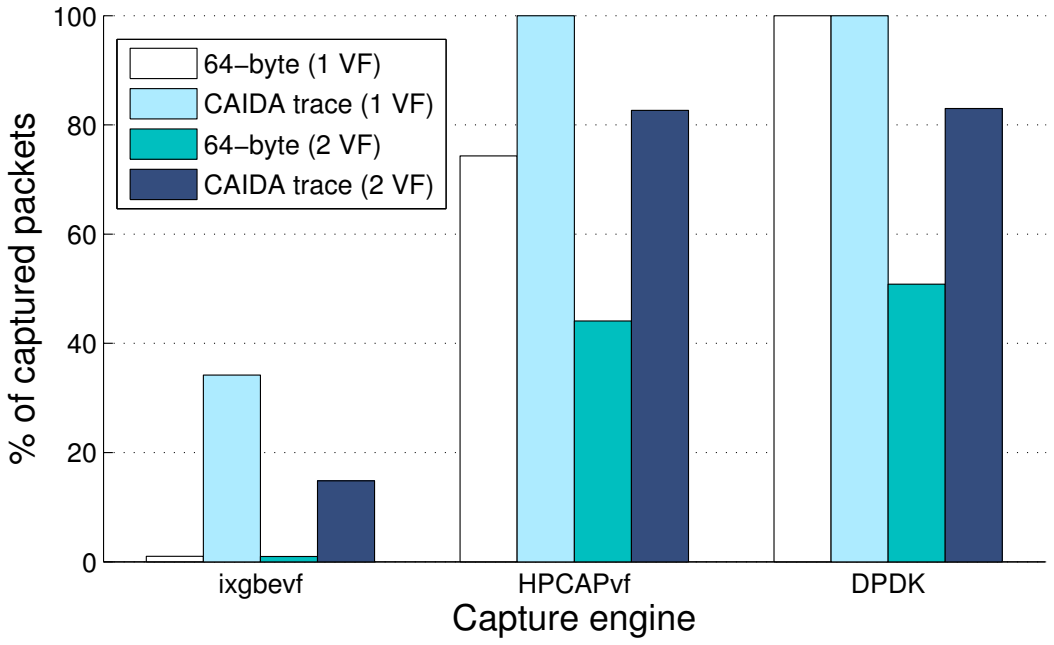


Figure 6.4: Packet capture performance obtained by different VF alternatives when capturing different traffic in a fully-saturated 10 Gb/s link

37.6% and 36.3% are respectively captured, but those figures are increased to 100% and 82.7% respectively when DPDK is used as VF generator.

When instantiating several VF through a single physical interface, the default traffic distribution policy is based on the MAC and IP addresses of each VM's interface the VF are connected to. Thus, each VF would only receive the traffic targeted to its corresponding VM. This would limit the use of this VF approach in scenarios where different network applications running on independent VMs need to be fed the same input traffic, which is the desirable case for scalable network monitoring. However, NICs such as Intel's 82599 supply a set of registers controlling packet mirroring and Multicast Promiscuous Enable (MPE) for each PF allowing to redirect the traffic assigned to a certain VF to another one. By enabling those options, any VF could receive all the traffic traversing the physical device, or the traffic corresponding to a different VM, regardless it is targeted to its VM or not. Note that enabling those features in the Intel 82599 NIC implies a hardware-level packet replication, which minimizes the impact on the capture process' performance. Depending on the physical driver used to generate the NIC's VFs, activating the packet mirroring and MPE bits may be done by tuning the driver's source code (that is the case when using `ixgbe` for generating the VFs) or by using a user-level application giving access to the NIC's registers (such as the `testpmd` application offered by Intel's DPDK).

Importantly, if several VMDqs are to be fed the same incoming packets, the NIC will have to issue additional copies for each additional VMDq, and overall packet capture performance may be degraded. This effect is shown in Fig. 6.4: adding a second VF to each physical device reduces the overall packet capture throughput obtained. When using `ixgbevf` the amount of packets captured is 1% when using either one or two VFs in the worst-case scenario, but falls from 34.2% with one VF to 14.8% with two when replaying the CAIDA trace. Intel DPDK also suffer performance loss as it is capable of capturing all of the packets in both scenarios when only one VF is instantiated, but it captures 50.8% of the 64-byte packets and 83.0% of the CAIDA ones when two VFs are used. Finally, HPCAP's performance falls in both cases: from 74.3% to 44.1% in the worst case and from 100% to 82.7% for the CAIDA trace.

6.3 Virtual network probe

Now that the diverse possibilities for mapping I/O devices into virtual machines have been explored, it is time to apply this knowledge in real-life scenarios.

The first use case we consider is the instantiation of that we called VNP (Virtual Network Probe): that is, a virtual machine that will be able to both capture the desired network's packets and store them into non-volatile storage. A virtual probe will thus have to simultaneously deal with network and disks I/O system in the most efficient way. Importantly, this virtual probe should be instantiated in a new virtual machine independent from the already running ones. Note that the possibility of completely acquiring each of the network and storage devices will be decisive when choosing the I/O virtualization alternative for each device. For example, PCI passthrough for network I/O will only be feasible if there are unused NIC in the system. Otherwise a VF must be used, but that means that the system manager left a VF free for the probe, or a driver re-installation may be required.

Amongst the results shown in the previous section, there are two possibilities for mapping each I/O source to a VM: the NIC could be mapped via PF passthrough or by creating a VF and using passthrough to connect it to the VM; while the RAID controller card could be paravirtualized with `virtio` or mapped via PF passthrough. The performance results for each of the possible combinations compared to the bare-metal case are summarized in Table 6.4. Note that in terms of packet capture, minimal-sized packets represents the worst-case scenario as the amount of packets per second will be maximal, while in terms of storage the bigger the packets are the higher the effective data throughput will be.

I/O configuration		Capture engine	% of packets processed			
			64-byte packets		CAIDA trace	
Network	Storage		Capture only	Capture and storage	Capture only	Capture and storage
Bare-metal		DPDK	100.0	95.8	100.0	100.0
		HPCAP	97.9	95.7	100.0	100.0
PT	PT	DPDK	100.0	97.6	100.0	100.0
		HPCAP	85.2	82.3	100.0	100.0
	VirtIO	DPDK	100.0	95.3	100.0	100.0
		HPCAP	85.2	82.8	100.0	100.0
VF	PT	DPDK	100.0	94.2	100.0	100.0
		HPCAPvf	82.7	82.6	100.0	100.0
	VirtIO	DPDK	100.0	68.0	100.0	100.0
		HPCAPvf	82.7	82.3	100.0	100.0

Table 6.4: Percentage of packets processed for diverse virtual I/O configurations in a network probe for packet capture only and packet capture and storage

Table 6.4 shows light performance degradation when capturing and storing the traffic with respect to the capture only case. This effect is due to the synchronization logic that has to be added between the capture and storage processes. As a consequence of being designed with the capture and storage goal in mind, HPCAP suffers a lighter degradation than Intel's DPDK, but still obtaining worse results when instantiated in a virtual machine in the worst-case scenario. Importantly, the use of the storage system mapped via `virtio` seems to entail the biggest performance degradation, specially when used in combination with Intel's DPDK managing network virtual functions. Additionally, the use of PCI VFs imposes a relevant performance degradation regardless the capture engine used of up to 5% more packets lost.

Network managers and practitioners will find the results of Table 6.4 useful, as each combination has different interesting properties that may be required in a specific situation. Nevertheless, in light of the results obtained we remark that the best capture and storage results in a virtual machine are obtained when both I/O resources are mapped to the VM via PF passthrough. Thus, this is the configuration we recommend when instantiating a maximum-performance VNP, as shown in Fig. 6.5. Different configurations to the ones studied along this work but as already mentioned they either lead to poorer performance or focus on inter-VM communication rather than network-to-VM.

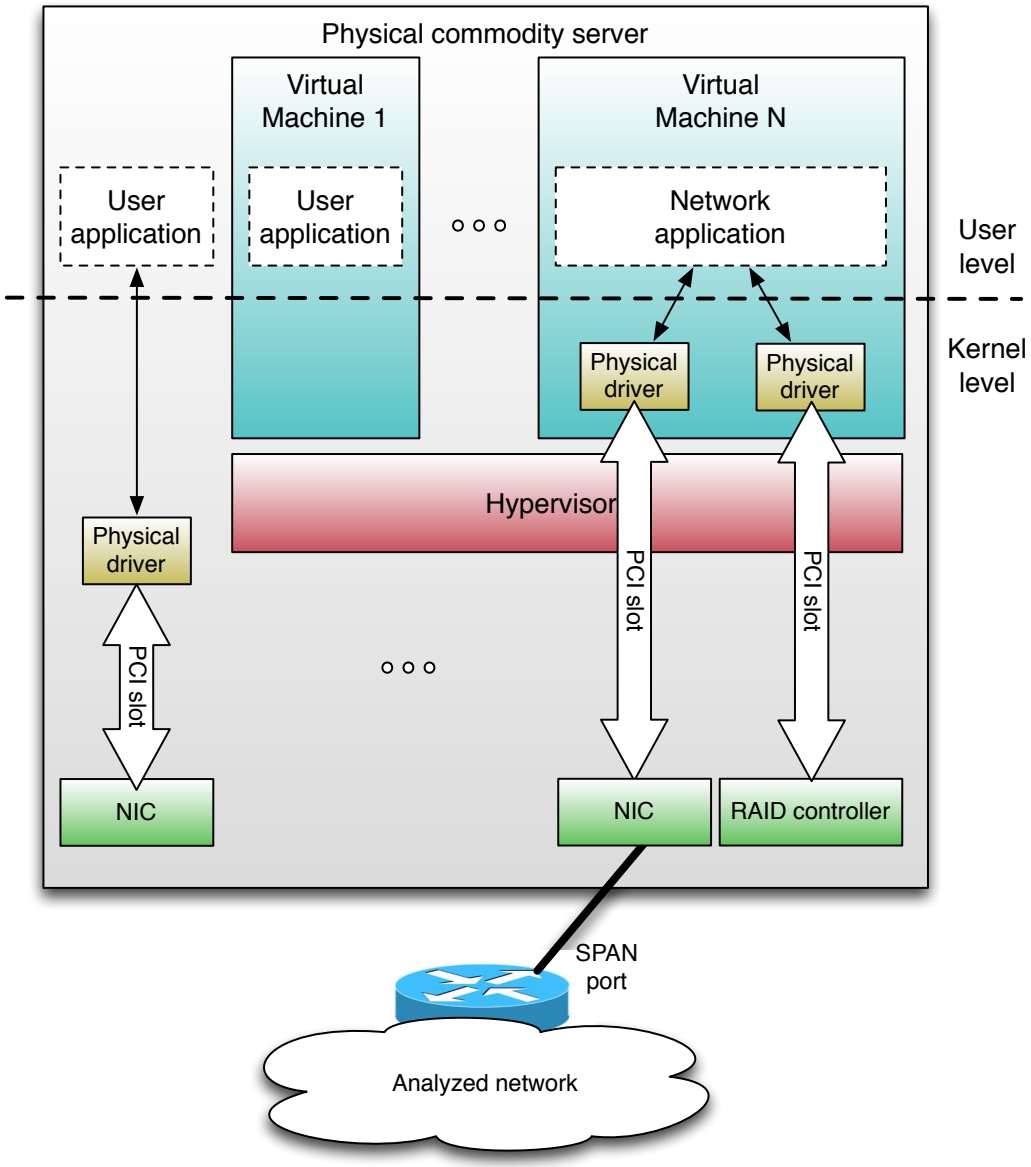


Figure 6.5: Example deployment of a virtual network probe (VNP)

6.4 Virtual network monitoring agent

As a different use-case, we propose the creation of that we named a VNMA (Virtual Network Monitoring Agent): the dynamic creation of a VM for a (probably limited) period of a time that may monitor all the traffic arriving to the NIC, a subset of it, or even the traffic generated by other VMs. The way we have defined our VNMA necessarily requires the network device to create a VF to be attached to such VNMA. This approach, based on the instantiation of several VFs when loading the NIC's physical driver allows instantiating several VMs in the same physical server whose traffic would be isolated from one to the other. Once everything is set, our VNMA could be deployed and configured so that it receives the desired traffic: the one not directed to the other VFs, the one to/from a certain set of VFs, or all of it.

The potential of this approach is that we can have a physical server hosting several VMs carrying out different tasks, e.g. an Apache server, an OpenFlow switch, or a certain network application using a VF-aware high-performance capture engine; and instantiate our VNMA for passively monitoring those VMs or the network itself without interfering on the other tasks. This scenario is depicted in Fig. 6.6. The only requirement for this system to work is to force all the VMs to work with VF-aware devices. As this is automatically done by Linux, this should not represent a severe requisite and allow even legacy applications to be trivially migrated. This approach may seem specially useful for those companies providing cloud or virtual computing services.

Let us assume for a while that we have a physical server with one or more active VMs receiving traffic from a 10 Gb/s. We also assume that that the traffic traversing the 10 Gb/s is equally distributed between all the active virtual machines, e.g. if there are tree VMs each one will receive 3.3 Gb/s. In this scenario, we want to instantiate a VNMA capable of monitoring the traffic traversing the link and, in particular, addressed to the rest of VMs. As it was explained when presenting the concept of VF, Intel's 82599 NIC allows creating mirroring rules so that the each packet can arrive to its corresponding VM and up to one more VF. However, this replication process is done at NIC level, consuming time and memory resources. Consequently, if the number of packets to be replicated is high the capture performance of our VNMA will be lower to the ones obtained when using one VF for capturing non-replicated traffic (the ones in Table 6.4). That is the effect shown by the results included in Table 6.5: although with only one VM and the VNMA active each of them should receive 100% of the incoming traffic, they receive 44.0% of it. Similarly, the amount of traffic captured with more VMs is reduced in the VNMA side, as packet replication has to be done from several sources. Those results have been carried out for packet capture only using Intel's DPDK (although the results are similar for HPCAPvf) in order

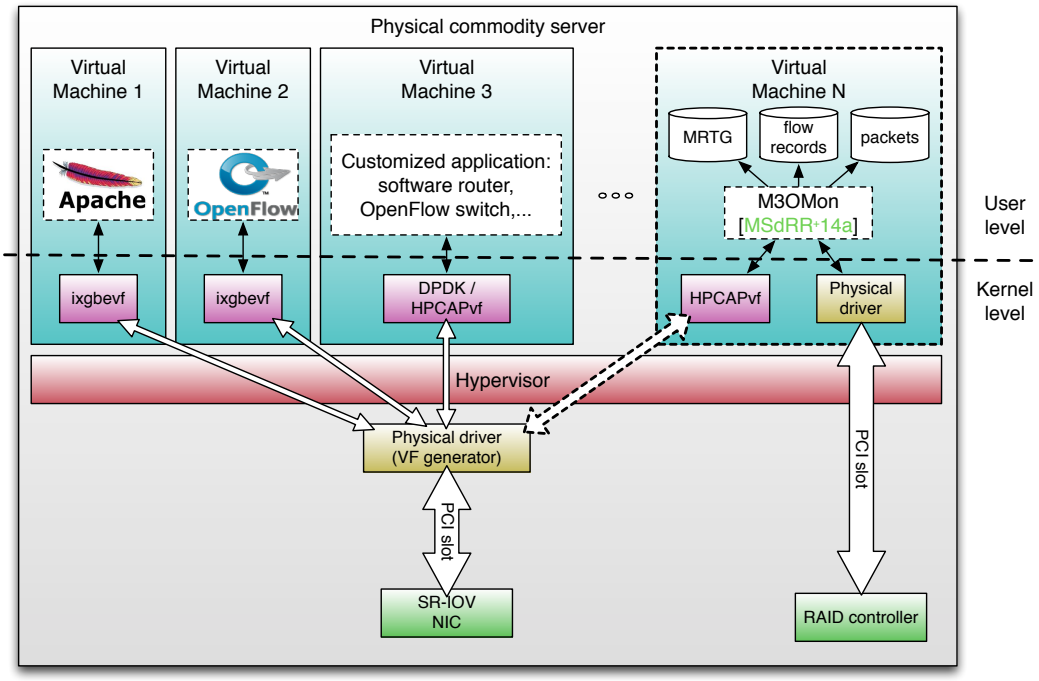


Figure 6.6: Example deployment of a virtual network monitoring agent (VNMA)

to illustrate the packet replication effect on performance. Results in Table 6.5 refer only to the worst packet-capture scenario, i.e., 64-byte packets.

# VMs	% of packets processed					
	VNMA in NUMA 0		VNMA in NUMA 1		Theoretical	
	VNMA	VMs	VNMA	VMs	VNMA	VMs
1 VM	44.0	44.0	41.8	41.8	100.0	100.0
2 VMs	31.3	15.6	42.1	21.0	100.0	50.0
3 VMs	30.4	10.1	41.1	13.7	100.0	33.3

Table 6.5: VNMA packet capture performance varying the number of additional VMs hosted in the same server and the NUMA node where the VNMA is attached to with minimal-size traffic

Table 6.5 shows another interesting effect, as is the placement of the VMs and the VNMA in terms of the NUMA node they are executed on. In the results obtained along the previous sections, the VMs where always attached to NUMA node 0 as both the NIC and the RAID controller card are connected to this node and thus performance would be maximal. Note that, as we have assumed that the performance power of our VNMA has to be greater than the one of each VM, we place the VNMA in a NUMA node alone using the six cores available, and we place the rest of VMs in the opposite NUMA node with two cores per

VM. As in those experiments one packet has to be addressed to each NUMA node, i.e., one to its corresponding VM and its copy to the VNMA, it could be thought that the NUMA placement policy is irrelevant. However, when packet replication is involved performance is favored when the VFs whose packets are to be replicated are mapped to VMs in the NUMA node closest to the NIC. This effect is also appreciated in the results for packet capture and storage, which are presented in Table 6.6. Note that, in this case the RAID controller card is also connected to NUMA node 0, but no node change is required for the VNMA because the amount of packets captured is far from the RAID's performance limits.

Being conscious of the performance degradation issue related to the packet replication effect, Table 6.6 shows the performance results for packet capture and storage for both Intel's DPDK and HPCAPvf. In this case, results do not change with the policy used to map the RAID controller and so it is not included in the table for simplicity. The experiments shown in this table have been carried with three VMs and the VNMA simultaneously active. Importantly, as the packet replication effect greatly damages packet capture performance, the results obtained for packet capture and storage are similar to those for packet capture only.

Capture engine	VNMA's NUMA node	% of packets processed	
		64-byte packets	CAIDA trace
DPDK	0	22.8	38.4
	1	50.4	76.8
HPCAPvf	0	30.7	48.0
	1	47.4	76.7

Table 6.6: VNMA capture and storage performance varying the traffic, the VF-aware capture engine used and the NUMA node where the VNMA is attached to

Note that the results offered in along this section refer to worst-case scenarios in which packet replication inflicts maximal performance degradation. Performance results near to the ones shown in Table 6.4 can be obtained if the configuration of the incoming is such that only a small fraction of it is targeted to the VMs in the same physical server and thus packet replication effect is minimized, which would be the case for real-life scenarios. Specifically, performance results prove the feasibility of our VNMA proposal to work in real scenarios with 1 Gb/s networks or under-used 10 Gb/s networks.

6.5 Conclusions

The results obtained along the experiments presented in this work assess the feasibility of migrating the usage of high-performance packet capture engines into virtualized environments. We have discussed the different configurations by which an I/O device can be used inside a virtual machine, and obtained the performance bounds for each of those configurations. Experimentation has allowed us to identify the different bottlenecks that may arise in a variety of network processing virtualized scenarios. Moreover, we have given a set of guidelines that allow exploiting the functionality of generating virtual network functions, enabling a set of interesting scenarios. Differently from PCI passthrough, the use of VF allows end-users to scale in the amount of network applications running on a single hardware, with the consequent saving in terms of space, cooling and power consumption.

We have presented a set of solutions available for use under GNU Linux and, in the case of HPCAP and Intel DPDK, accessible as free software. As an additional contribution, we have developed HPCAPvf, a VF-aware version of HPCAP, offering a set of interesting features and capabilities. HPCAPvf may be used in any Linux distribution with kernel version newer than or equal to 2.6.32 without modifying nor kernel nor hardware configuration. Furthermore, we have made available the source code of HPCAPvf under a GPL license[1].

[1] https://github.com/hpcn-uam/HPCAP

7 Conclusions and future work

This chapter has the purpose of highlighting and summarizing the main results of this PhD. thesis. That includes the practical implications, the divulgation results and the industrial applications of this work. Finally, a set of future work paths are presented as possible continuations of this work.

The use of off-the-shelf systems on high-performance networked tasks has opened an exciting scenario, on which this thesis has been focused. Thus, thesis has been oriented to achieve that any network task can be carried out by a flexible, extensible, adaptable and even inexpensive system. Examples of these tasks that have been enriched of this novel paradigm are applications such as software routers, anomaly and intrusion detection, traffic classification, and VoIP monitoring. Unfortunately, the development process of a high-performance networking task on commodity hardware from its foundation stone may result a non-trivial process composed of a set of thorny sub-tasks, each of which presents fine-tuned configuration details. In this light, this work has aimed at providing practitioners and researchers with a road-map to the exploration of this useful paradigm. Specifically, this thesis has contributed to the field in the following ways:

1. An extensive guide for researchers and practitioners to understand the main challenges that a general-purpose high-performance network application must face has been provided. Furthermore, a detailed description and guide has been given for each of the state-of-the-art alternatives. With this information, future users will find it easier to choose the existing solution that best fits their needs, or even to develop their own solution from scratch.

2. The research community has been alerted about the counterparts that some high-performance network processing solutions may entail, such as timestamping inaccuracy. Additionally, an extensive set of performance evaluation metrics have supplied, some of them showing that a proper hardware tuning can greatly improve the performance experienced by standard solutions.

3. The HPCAP engine provides accurate packet timestamping, line-rate packet storage and duplicate removal with minimal interference. Furthermore, the buffer-oriented architecture of HPCAP has been created in order to pipeline the process carried out over each packet, so that part of the process is carried out at kernel level and the rest is made at user-level. This pretends to alleviate the strict timing constraints existing when doing some computational task over each incoming packet in a high-speed network. Extensive performance and functional assessment studies have been carried out, in order to provide a solid validation of the developments made.

4. A multi-granular multi-purpose framework has been built on top of HPCAP. With the help of M^{3}OMon, network developers may easily develop new high-performance applications feeding from different data sources with diverse aggregation profiles, namely: network packets, flow records or time series.

5. The feasibility of migrating the knowledge acquired in physical environments to virtual ones has been assessed, and the overhead implied by the application of virtualization techniques has proven attractively low.

6. A NVF-compliant version of HPCAP, HPCAPvf, has been successfully developed and its performance bounds of limitations have been established.

7. In order to serve as a start point for the development of new applications, as well as for the shake of repeatability, the software developments carried out along this thesis work have been published under an Open-Source license.

7.1 Results dissemination and publications

(i) In Chapter 2 diverse hardware architectures for network processing were presented, together with their pros and cons. Apart from their industrial and academic application, the development of two network processing prototypes, namely `Argos` and `Twin`$^{-1}$, meant the acquisition of valuable experience both in terms of using diverse hardware architecture and some

fundamental requirements that a general-purpose network monitoring system should address.

The results obtained in this chapters have directly led to the following publications:

- J. Garnica, V. Moreno, I. Gonzalez, S. Lopez-Buedo, F.J. Gomez-Arribas and J. Aracil. *ARGOS: A GPS Time-Synchronized Network Interface Card based on NetFPGA*. In 2nd North American NetFPGA Developers Workshop, August 2010.
  Non-indexed workshop.
- V. Moreno, J. Garnica, F.J. Gomez-Arribas, S. Lopez-Buedo, I. Gonzalez, J. Aracil, M. Izal, E. Magana and D. Morato. *High-accuracy network monitoring using ETOMIC testbed*. In the 7th EURO-NF Conference on Next Generation Internet (NGI 2011), June 2011.
  Non-indexed conference.
- V. Moreno, F.J. Gomez-Arribas, I. Gonzalez, D. Sanchez-Roman, G. Sutter, S. Lopez-Buedo. *Comparativa del uso de HLLs en FPGA, GPU y Multicore para la aceleracion de una aplicacion de red IP*. In the XI Edicion Jornadas de Computacion Reconfigurable y Aplicaciones (JCRA2011), September 2011. **Non-indexed conference.**

And also those results have led, in an indirect way, to the following publications:

- I. Csabai, A. Fekete, P. Haga, B. Hullar, G. Kurucz, S. Laki, P. Matray, J. Steger, G. Vattay, F. Espina, S. Garcia-Jimenez, M. Izal, E. Magana, D. Morato, J. Aracil, F.J. Gomez, I. Gonzalez, S. Lopez-Buedo, V. Moreno, J. Ramos. *ETOMIC Advanced Network Monitoring System for Future Internet Experimentation*. In 6th International Conference on Testbeds and Research Infrastructures for the Development of Networks & Communities (TridentCom 2010), May 2010. **Non-indexed conference.**
- V. Lopez, J.L. Anamuro, V. Moreno, J. Lopez de Vergara, J. Aracil, C. Garcia, J.P. Fernandez-Palacios and M. Izal. *Implementation of Multilayer techniques using FEDERICA, PASITO and OneLab network infrastructures*. In the 17th IEEE International Conference on Networks (ICON2011), December 2011. **Conference rank:** B (obtained from the 2013 CORE Conference Ranking).

(ii) In Chapter 3 the limitations of both the default NIC driver and networking stack were highlighted, and then, we detailed how to tune them to boost up performance. Fortunately, the research community has presented different solutions, named packet captures engines, that have put in practice these tunings and directly provide application layers developments with captured

packets, thus saving much of the effort to manage and understand low-level interactions. In this light, it has been thoroughly explained how these engines work, pinpointed the most appropriate ones according to a trade-off between performance and different user requirements, which is summarized in a quantitative way in Fig. **??**.

Importantly, a guide to easily set up any packet capture engine has been provided in Appendix A. This guide includes a number of code examples through which those practitioners not interested on low-level details but on developing applications over commodity hardware may rapidly achieve remarkable competence on the area.

Finally, the most successful systems based on these novel capture engines which have achieved not only remarkable performance results, but also become commercial solutions, have been studied and listed. First, this study allows us to provide newcomers to the application-layer development with a set of advices to ease their work on this area. And second, we have presented a detailed state-of-the-art of implementations already in place that may turn out useful for comparison purposes, setting of current performance bounds and as a catalyst for the arrival of new applications based on this paradigm.

The results obtained in this chapters have directly led to the following publication:

- V. Moreno, J. Ramos, P.M. Santiago del Río, J.L. Garcia-Dorado, F.J. Gomez-Arribas and J. Aracil. *Commodity Packet Capture Engines: tutorial, cookbook and applicability*. In IEEE Communications Surveys & Tutorials, accepted for publication.
  Journal impact factor: 6.490. **Journal rank:** 2/78 (Q1). **Category:** Telecommunications (obtained from the 2013 JCR).

And also those results have led, in an indirect way, to the following publication:

- J.L. Garcia-Dorado, F. Mata, J. Ramos, P.M. Santiago del Río, V. Moreno, and J. Aracil . *High-performance network traffic processing systems using commodity hardware*. In Lecture Notes in Computer Science, vol. 7754, Springer Berlin Heidelberg, 2013.
  Non-indexed publication.

(iii) Chapter 4 has presented our proposal for the high-performance network processing problem, and one of the main contributions of this thesis, which is the HPCAP packet capture engine. Specifically, this capture engine has been made accessible under a Open-Source code license, an can be found on the Github website [HPC15]. This chapter has not been devoted only to a functional description of HPCAP, but also to an evaluation of the diverse

features that distinguish HPCAP among other approaches, which are: accurate packet timestamping, high-performance network traffic storage and the possibility of duplicated packets removal.

The results obtained in this chapter have directly led to the following publications:

- V. Moreno, P.M. Santiago del Río, J. Ramos, J.J. Garnica, and J.L. Garcia-Dorado. *Batch to the future: Analyzing timestamp accuracy of high-performance packet I/O engines*. On IEEE Communications Letters, **16** (2012), no.11, 1888.1891.
  Journal impact factor: 1.160. **Journal rank:** 30/78 (Q2). **Category:** Telecommunications (obtained from the 2012 JCR).
- V. Moreno, P.M. Santiago del Río, J. Ramos, J.L. Garcia-Dorado, I. Gonzalez, F.J. Gomez-Arribas and J. Aracil. *Packet storage at multi-gigabit rates using off-the-shelf systems*. On the 16th IEEE International Conference on High Performance Computing and Communications (HPCC 2014), August 2014.
  Conference rank: B (obtained from the 2014 CORE Conference Ranking).
- V. Moreno, J. Ramos, J.L. Garcia-Dorado, I. Gonzalez, F.J. Gomez-Arribas and J. Aracil. *Testing the capacity of off-the-self systems to store 10GbE traffic*. Submitted to the Network Testing Series of IEEE Communications Magazine.
  Journal impact factor: 4.460. **Journal rank:** 3/78 (Q1). **Category:** Telecommunications (obtained from the 2013 JCR).

(iv) Chapter 5 presents M^{3}OMon as a framework built on top of HPCAP providing final users with an easy way of deploying new network applications feeding from up to three sources of network data: packets, flow records, or time series. A thorough and exhaustive performance evaluation on the capabilities offered by this framework is also presented, showing the feasibility of building line-rate applications on top of it. The M^{3}OMon framework has also been made available under an Open-Source license in the samples folder of the HPCAP's GitHub repository [HPC15].

The results obtained in this chapter have directly led to the following publication:

- V. Moreno, P.M. Santiago del Río, J. Ramos, D. Muelas, J.L. Garcia-Dorado, F.J. Gomez-Arribas, and J. Aracil. *Multi- granular, multi-purpose and multi-Gb/s monitoring on off-the- shelf systems*. On International Journal of Network Management, **24** (2014), no. 4, 221-234.
  Journal impact factor: 0.517. **Journal rank:** 60/78 (Q4). **Category:**

Telecommunications (obtained from the 2013 JCR).

And also those results have led, in an indirect way, to the following publication:

- J. L. Garcia-Dorado, P.M. Santiago del Río, J. Ramos, D. Muelas, V. Moreno, J.E. Lopez de Vergara, and J. Aracil. *Low-cost and high-performance: VoIP monitoring and full-data retention at multi-Gb/s rates using commodity hardware*. On International Journal of Network Management, **24** (2014), no. 3, 181-199.
  Journal impact factor: 0.517. **Journal rank:** 60/78 (Q4). **Category:** Telecommunications (obtained from the 2013 JCR).

(v) Finally, Chapter 6 has been focused on the application of the knowledge and tools acquired in the previous parts of this thesis, in the Network Function Virtualization philosophy. Specifically, this chapter has been devoted to the study of under which circumstances high-performance network processing can be achieved in virtualized scenarios, supported by an extensive set of tests. Furthermore, this chapter also contributes to the research community by spreading the availability under an Open-Source license of HPCAPvf on GitHub [HPC15].

The results obtained in this chapter have directly led to the following publication:

- V. Moreno, R. Leira, I. Gonzalez, F.J. Gomez-Arribas. *Towards high-performance network processing in virtualized environments*. Submitted to the 17th IEEE International Conference on High Performance and Communications (HPCC2015).
  Conference rank: B (obtained from the 2014 CORE Conference Ranking).

7.2 Industrial applications

The results of this thesis work have not been restricted to the research and academia fields, but have also had a direct impact in industrial environments. This has been possible due to the existence of synergies between Spanish Universities and companies, with the aim of promoting technology transfer actions. In this case, the results of this work are being applied and delivered by Naudit HPCN S.L. [Nau15]. Naudit HPCN is a technology-base startup company created as a spin-off of two Spanish public universities: the *Universidad Autónoma de Madrid* (UAM) and the *Universidad Pública de Navarra* (UPNA). It is included as part of the Campus of International Excellence initiative [UAM].

Through Naudit HPCN, it has been possible to carry out several innovation and technology transfer projects, which have, up to date, involved clients such as the Spanish Ministry of Industry and the Spanish National Post and Telegraph Society (Correos); Internet service providers such as Telefónica-Movistar and British Telecom; or banking institutions as BBVA Latin-America, Banco Popular and Inversis.

More specifically, the results of this work have been applied into industrial scenarios:

(i) **Network traffic storage:** HPCAP has been deployed in several scenarios in order to accomplish one of its main design goals: storing network traffic from high-speed networks. It has been deployed in British Telecom, Banco Popular, Banco Sabadell and the Spanish National Post and Telegraph Society.

(ii) **Multi-granular network processing:** the M^{3}OMon framework has been installed, by means of its sample application, DetectPro (see 5.4.1), in several places. Those places include the Latin-American network infrastructure from BBVA Bank and the Spanish National Post and Telegraph Society.

(iii) **Multi-media traffic analysis:** the VoIPCallMon application (see 5.4.2), also developed on top of HPCAP is currently being installed in Telefonica's Latin-American network.

(iv) **GPU-based traffic management:** `Twin`$^{-1}$, the duplicate removal and traffic distribution presented in 2.2.2 has been installed in Telefonica-Movistar Imagenio's (the operator's name for their TV services) core network.

7.3 Future work

Several lines have been identified as possible continuation of the current works:

- **Supporting new NICs:** there are some new NICs, from Intel and other vendors, that should be supported in the future. Those NICs include interesting features such larger descriptor rings, opening the possibility of enhancing system's capture performance as well as more advanced Virtual-Functions management. Furthermore, the results obtained along this thesis should also be applied in new NICs supporting higher link speeds: 40 Gb/s, 100 Gb/s and beyond.
- **Optimized packet transmission:** the HPCAP engine and all their derivations presented in this work are focused on packet reception. The same low-level concepts that have explained along this thesis are also applicable for packet transmission, and would thus be a natural continuation. In fact, this continuation could be done following different paths, all of them with interesting applications:
    - Focusing on bandwidth: obtaining the maximum transmission bandwidth when replaying a previously stored —big— network trace. This would result extremely interesting for network stress-testing.
    - Focusing on temporal accuracy: the higher the network speed is, the tighter temporal constraints become and thus packet timing becomes a crucial aspect. Having an accurate way of defining packet inter-departure time can become vital in order to reproduce network failure scenarios.
    - Synthetic traffic generation: diverse high-level applications require not high-throughput traffic generations, but a high control over the generated traffic. Creating a flexible system on top of HPCAP capable of generating traffic with diverse characteristics (packet-size, quintuples, inter-departure times) can enhance low-intrusive active network monitoring tasks.
- **Further analysis on timestamping error sources:** the dependence of software timestamping techniques from the system's clocks and scheduling policies implies the appearance of several error sources in packet timestamping. A further study on the timescale in which those error are made depending on several system-load factors could help developing a policy capable of telling the final user how loaded could their system be for a certain level of accuracy. Moreover, we believe that timestamp error have strong auto-correlations. Analysing the cyclic nature of those auto-correlations could help deciding the maximum burdens in which timestamp

keeps a certain level of accuracy. This could be very interesting for active monitoring techniques based on the transmission of pairs, batches or trains of packet through a network.

- **Discovering new virtualization possibilities:** it is more common everyday for new hardware devices to include function virtualization support. We had no access to a SR-IOV RAID controller card during this study, but we find interesting to complete our tests including this possibility in our experimental repository. We also see interesting on exploring the NFV-possibilities offered by other hardware alternatives such as FPGAs, GPGUs, etc.
- **Surpassing HPCAP's current limitations:** along this thesis the pros and cons of HPCAP have presented. When applying our system in industrial scenarios we realized the is still interesting work to be done:
  - When HPCAP works with several NICs at the same time, the kernel buffer memory limited is still 1GB. Although we made it easily configurable so the final user can decide the amount of memory dedicated to each NIC, a drastic reduction on the buffers size has a negative impact of the interface's packet capture performance. Consequently, we suggest to include support for defining the buffer regions using Linux's hugepages techniques, which could eliminate this restriction.
  - If the network's traffic to be stored is high, it can only be kept in the non-volatile storage system for a short period of time. Consequently, creating policies that allow deciding whether a packet's data is interesting to be stored or not, or even how much of each packet should be stored can mean a significant difference.

Some of these future-work lines are, in fact, under development at the present time.

8 Conclusiones y trabajo futuro

Este capítulo tiene como objetivo subrayar y resumir los principales resultados alcanzados por esta tesis doctoral. Abarca por tanto las implicaciones prácticas de dichos resultados, la divulgación científica llevada a cabo y las aplicaciones industriales de este trabajo. Finalmente, se presentan una serie de posibles líneas de trabajo que supondrían una continuación natural de esta tesis.

La utilización de elementos hardware estándar en tareas de procesamiento de redes de alta velocidad ha abierto un escenario con muchas posibilidades. El principal objetivo de la tesis ha sido demostrar que en este escenario incluso la tarea más compleja puede ser llevada a cabo de una manera flexible, extensible y con un coste comedido. Ejemplos de tareas de red que se han favorecido de este novedoso paradigma son róuters software, detección de anomalías e intrusiones, clasificación de tráfico y monitorización de voz sobre IP. Desafortunadamente, el proceso de desarrollo de una aplicación de red de altas prestaciones en hardware estándar desde cero resulta ser un proceso muy tedioso que implica una serie de espinosas tareas, cada una de las cuales requiere llevar a cabo configuraciones a muy bajo nivel. En este contexto, este trabajo ha pretendido allanar a aquellos investigadores y desarrolladores que lo deseen, el camino que les permita explorar y explotar este mundo. Concretamente, esta tesis ha contribuido al campo de las siguientes maneras:

1. Se ha creado una extensa guía dirigida a profesionales del sector e investigadores, con el objetivo de entender los principales retos a los que una aplicación de procesamiento de tráfico de red debe enfrentarse hoy en día. Adicionalmente, se ha llevado a cabo un estudio descriptivo y evaluación de cada una de las alternativas disponibles en el estado del arte. Con la información aportada, potenciales usuarios tendrán una base de

conocimiento sólido y de resultados empíricos que les ayudará a discernir qué solución se ajusta más a sus necesidades, o incluso desarrollar su propia solución.

2. Se ha alertado a la comunidad investigadora sobre los efectos colaterales que algunas soluciones de procesamiento de tráfico de red a alta velocidad llevan asociadas, como por ejemplo la falta de precisión en el marcado temporal de los paquetes. También se han presentado experimentos funcionales y de rendimiento que han demostrado que una modificación a bajo nivel en la configuración puede incrementar enormemente el rendimiento de soluciones tradicionales de procesamiento de red.

3. El motor HPCAP ha sido diseñado con vistas a proveer un marcado de tiempo de paquetes preciso, almacenamiento del tráfico a tasa de línea y la eliminación de paquetes duplicados, con mínima interferencia tanto a nivel funcional como de rendimiento. Adicionalmente, la arquitectura orientada a buffers de HPCAP pretende segmentar el procesamiento llevado a cabo sobre cada paquete, de manera que una parte recaiga a nivel del kernel del sistema, y el resto a nivel de usuario. Esta filosofía pretende aliviar en la medida de lo posible, las estrictas restricciones temporales existentes para procesar cada paquete en redes de alta velocidad. Cada una de las características distintivas red HPCAP han sido extensivamente evaluadas y probadas, con vistas a proveer un sólido marco de referencia para desarrollos futuros.

4. Se ha construido sobre HPCAP un marco de propósito general y de alto rendimiento. Con la ayuda de M^{3}OMon, los desarrolladores de aplicaciones de red podrán crear fácilmente aplicaciones que se alimenten de distintas fuentes de datos de red: los propios paquetes, registros de flujos o series temporales.

5. Se ha demostrado la posibilidad de migrar todo el conocimiento adquirido para el procesamiento de tráfico de red en entornos físicos a entornos de virtualización. También se ha comprobado empíricamente que el impacto sobre el rendimiento experimentado por dicha migración puede ser atractivamente bajo.

6. Se ha desarrollado HPCAPvf, una versión de HPCAP compatible con la filosofía de virtualización de funciones de red, que se encuentra en pleno apogeo hoy en día. Al igual que para la versión original, se han llevado a cabo una serie de experimentos para evaluar los límites de rendimiento de esta versión.

7. Con el fin de servir como punto de partida para el desarrollo de nuevas aplicaciones así como para fomentar la reproducibilidad de los experimentos presentados, los desarrollos software llevados a cabo en esta tesis se

han publicado bajo licencia de códio abierto.

8.1 Diseminación y divulgación de los resultados alcanzados

(i) En el Capítulo 2 se han evaluado diferentes arquitecturas hardware para el procesamiento de tráfico de red, junto con sus pros y sus contras. Aparte de su aplicación industrial y en entornos de investigación, el desarrollo de los prototipos de procesamiento de red `Argos` y `Twin`$^{-1}$ han permitido la adquisición de una valiosa experiencia en términos del uso de diversas alternativas hardware así como requisitos fundamentales que cualquier sistema de monitorización de red de altas prestaciones debería tener.

Los resultados obtenidos a lo largo de la elaboración de este capítulo han generado, de forma directa, las siguientes publicaciones:

- J. Garnica, V. Moreno, I. Gonzalez, S. Lopez-Buedo, F.J. Gomez-Arribas and J. Aracil. *ARGOS: A GPS Time-Synchronized Network Interface Card based on NetFPGA*. In 2nd North American NetFPGA Developers Workshop, August 2010.
  Workshop no indexado.
- V. Moreno, J. Garnica, F.J. Gomez-Arribas, S. Lopez-Buedo, I. Gonzalez, J. Aracil, M. Izal, E. Magana and D. Morato. *High-accuracy network monitoring using ETOMIC testbed*. In the 7th EURO-NF Conference on Next Generation Internet (NGI 2011), June 2011.
  Conferencia no indexada.
- V. Moreno, F.J. Gomez-Arribas, I. Gonzalez, D. Sanchez-Roman, G. Sutter, S. Lopez-Buedo. *Comparativa del uso de HLLs en FPGA, GPU y Multicore para la aceleracion de una aplicacion de red IP*. In the XI Edicion Jornadas de Computacion Reconfigurable y Aplicaciones (JCRA2011), September 2011.
  Conferencia no indexada.

Estos mismos resultados también han contribuido a la realización de las siguientes publicaciones:

- I. Csabai, A. Fekete, P. Haga, B. Hullar, G. Kurucz, S. Laki, P. Matray, J. Steger, G. Vattay, F. Espina, S. Garcia-Jimenez, M. Izal, E. Magana, D. Morato, J. Aracil, F.J. Gomez, I. Gonzalez, S. Lopez-Buedo, V. Moreno, J. Ramos. *ETOMIC Advanced Network Monitoring System for Future Internet Experimentation*. In 6th International Conference on Testbeds and Research Infrastructures for the Development

of Networks & Communities (TridentCom 2010), May 2010.
Conferencia no indexada.

- V. Lopez, J.L. Anamuro, V. Moreno, J. Lopez de Vergara, J. Aracil, C. Garcia, J.P. Fernandez-Palacios and M. Izal. *Implementation of Multi-layer techniques using FEDERICA, PASITO and OneLab network infrastructures*. In the 17th IEEE International Conference on Networks (ICON2011), December 2011.
Ranking de la conferencia: B (obtenida del Ranking de conferencias CORE 2013).

(ii) En el Capítulo 3 se han identificado las limitaciones de procesamiento de las NIC estándar, y se han detallado diversas alternativas para solventar esas limitaciones. Afortunadamente, la comunidad investigadora ha presentado diversas soluciones, denominadas motores de captura, que han puesto en práctica dichas optimizaciones para entregar los paquetes capturados a la capas de aplicación, ahorrando el esfuerzo computacional que supone comprender y gestionar las interacciones a bajo nivel implicadas. En este contexto, se ha explicado en detalle el funcionamiento de estos motores de captura, subrayando las características de cada uno de ellos que se han resumido en témirnos cualitativos en la tabla 3.1. En adición a esta comparativa, se ha elaborado un guía con las instrucciones básicas requeridas para poner en marcha cada uno de los motores de captura existentes, y que se incluye en el Apéndice A.

Finalmente, se ha presentado un resumen incluyendo los sistemas más reseñables que se han desarrollado sobre los motores de captura existentes. Este estudio permite, en primer lugar, que los recién llegados al desarrollo de aplicaciones de red se encuentre con un conjunto de experiencias de uso que le resultarán de gran utilidad. En segundo lugar, este resumen supone una referencia del estado del arte existente en el campo, estableciendo los límites de procesamiento actuales y sirviendo como catalizador de futuros desarrollos.

Los resultados obtenidos a lo largo de la elaboración de este capítulo han generado, de forma directa, la siguiente publicación:

- V. Moreno, J. Ramos, P.M. Santiago del Río, J.L. Garcia-Dorado, F.J. Gomez-Arribas and J. Aracil. *Commodity Packet Capture Engines: tutorial, cookbook and applicability*. In IEEE Communications Surveys & Tutorials, accepted for publication.
Journal impact factor: 6.490. **Journal rank:** 2/78 (Q1). **Category:** Telecommunications (obtained from the 2013 JCR).

Estos mismos resultados también han contribuido a la realización de la siguiente publicación:

- J.L. Garcia-Dorado, F. Mata, J. Ramos, P.M. Santiago del Río, V. Moreno, and J. Aracil . *High-performance network traffic processing systems using commodity hardware*. In Lecture Notes in Computer Science, vol. 7754, Springer Berlin Heidelberg, 2013.
  Non-indexed publication.

(iii) El objetivo del Capítulo 4 es el de presentar nuestra propuesta para el procesamiento de red de altas prestaciones y una de las principales contribuciones de esta tesis, el motor de captura HPCAP. Dicho motor de captura se ha hecho accesible bajo una licencia de software libre en la página web GitHub [HPC15]. El capítulo no se ha restringido a una descripción funcional de HPCAP, sino que también ha llevado a cabo una evaluación de las características que diferencias a HPCAP entre el resto de alternativas, que son: el marcado preciso de paquetes, el almacenamiento de tráfico a tasa de línea y la eliminación de paquetes duplicados.

Los resultados obtenidos a lo largo de la elaboración de este capítulo han generado, de forma directa, las siguientes publicaciones:

- V. Moreno, P.M. Santiago del Río, J. Ramos, J.J. Garnica, and J.L. Garcia-Dorado. *Batch to the future: Analyzing timestamp accuracy of high-performance packet I/O engines*. On IEEE Communications Letters, **16** (2012), no.11, 1888.1891.
  Journal impact factor: 1.160. **Journal rank:** 30/78 (Q2). **Category:** Telecommunications (obtained from the 2012 JCR).
- V. Moreno, P.M. Santiago del Río, J. Ramos, J.L. Garcia-Dorado, I. Gonzalez, F.J. Gomez-Arribas and J. Aracil. *Packet storage at multi-gigabit rates using off-the-shelf systems*. On the 16th IEEE International Conference on High Performance Computing and Communications (HPCC 2014), August 2014.
  Ranking de la conferencia: B (obtenida del Ranking de conferencias CORE 2014).
- V. Moreno, J. Ramos, J.L. Garcia-Dorado, I. Gonzalez, F.J. Gomez-Arribas and J. Aracil. *Testing the capacity of off-the-self systems to store 10GbE traffic*. Enviado al Network Testing Series of IEEE Communications Magazine.
  Journal impact factor: 4.460. **Journal rank:** 3/78 (Q1). **Category:** Telecommunications (obtained from the 2013 JCR).

(iv) En el Capítulo 5 se presenta M^{3}OMon, un marco de desarrollo construido a partir de HPCAP que provee una forma sencilla de crear y desplegar aplicaciones de red de alto rendimiento alimentándose de hasta tres fuentes de datos de red: paquetes, registros de flujos y series temporales. En el mismo capítulo se incluye también una exhaustiva evaluación del

rendimiento ofrecido por este marco de desarrollo. M^{3}OMon también se ha publicado bajo licencia de software libre, y se encuentra dentro de la carpeta de aplicaciones de ejemplo del repositorio de HPCAP en GitHub [HPC15].

Los resultados obtenidos a lo largo de la elaboración de este capítulo han generado, de forma directa, la siguiente publicación:

- V. Moreno, P.M. Santiago del Río, J. Ramos, D. Muelas, J.L. Garcia-Dorado, F.J. Gomez-Arribas, and J. Aracil. *Multi- granular, multi-purpose and multi-Gb/s monitoring on off-the- shelf systems*. On International Journal of Network Management, **24** (2014), no. 4, 221-234.
  Journal impact factor: 0.517. **Journal rank:** 60/78 (Q4). **Category:** Telecommunications (obtained from the 2013 JCR).

Estos mismos resultados también han contribuido a la realización de la siguiente publicación:

- J. L. Garcia-Dorado, P.M. Santiago del Río, J. Ramos, D. Muelas, V. Moreno, J.E. Lopez de Vergara, and J. Aracil. *Low-cost and high-performance: VoIP monitoring and full-data retention at multi-Gb/s rates using commodity hardware*. On International Journal of Network Management, **24** (2014), no. 3, 181-199.
  Journal impact factor: 0.517. **Journal rank:** 60/78 (Q4). **Category:** Telecommunications (obtained from the 2013 JCR).

(v) Finalmente, el capítulo 6 se ha centrado en la migración del conocimiento previamente adquirido en entornos de ejecución físicos a entornos virtualizados. Concretamente, se ha estudiado bajo qué circunstancias el procesamiento de red puede realizarse utilizando plataformas virtuales, a través de una serie de experimentos llevados a cabo. Adicionalmente, el desarrollo a lo largo de este capítulo se ha materializado en la puesta a disposición de cara a la comunidad científica de HPCAPvf bajo licencia de software libre.

Los resultados obtenidos a lo largo de la elaboración de este capítulo han generado, de forma directa, la siguiente publicación:

- V. Moreno, R. Leira, I. Gonzalez, F.J. Gomez-Arribas. *Towards high-performance network processing in virtualized environments*. Submitted to the 17th IEEE International Conference on High Performance and Communications (HPCC2015).
  Ranking de la conferencia: B (obtenida del Ranking de conferencias CORE 2014).

8.2 Aplicaciones industriales

Los resultados de esta tesis doctoral no se han restringido a ámbitos académicos, sino que también han tenido impacto en entornos industriales. Esto ha sido posible gracias a la existencia de sinergias entre las universidades españolas y las empresas, con el objetivo de fomentar iniciativas de transferencia tecnológica. En este caso, los resultados de este trabajo están siendo aplicados e implantados a través de Naudit HPCN S.L. [Nau15]. Naudit HPCN es un *startup* de base tecnológica creada como una spin-off de dos universidades púbicas: la *Universidad Autónoma de Madrid* (UAM) y la *Universidad Pública de Navarra* (UPNA). Además es parte de la iniciativa de Campus de Excelencia Internacional de la UAM [UAM].

A través de Naudit HPCN, ha sido posible el desarrollo de diversos proyectos de innovación y transferencia tecnológica que han implicado la participación de entidades como el Ministerio Español de Industria, la Compañia Nacional de Correos y Telégrafos (Correos); proveedores de servicio como Telefónica-Movistar y British Telecom; o instituciones de banca como la división Latinoamericana de BBVA, Banco Popular e Inversis.

En concreto, los resultados de este trabajo han sido aplicados en los siguientes escenarios:

(i) **Almacenamiento de tráfico de red:** HPCAP ha sido desplegado en diversos escenarios con el fin de servir a uno de sus objetivos de diseño: el almacenamiento de tráfico en redes de alta velocidad. Ha sido desplegado en British Telecom, Banco Popular, Banco Santander y Correos.

(ii) **Procesamiento de tráfico con múltiples granularidades:** el marco de desarrollo M^{3}OMon ha sido instalado, por medio de una de sus aplicaciones, `Detect-pro` (ver 5.4.1) en BBVA Latinoamérica y Correos.

(iii) **Análisis de tráfico multimedia:** la aplicación VoIPCallMon (ver 5.4.2) también desarrollada sobre HPCAP, se encuentra instalada en la red Latinoamericana de Telefónica.

(iv) **Gestión de tráfico basada en GPGPU:** `Twin`$^{-1}$, la aplicación de eliminación de duplicados y distribución de tráfico presentado en 2.2.2 ha sido instalada en la red de Imagenio de Telefónica-Movistar.

8.3 Trabajo futuro

A la luz de los resultados obtenidos y de las experiencia adquirida a lo largo del desarrollo de esta tesis, se proponen las siguientes líneas de trabajo futuro:

- **Soporte de nuevas NICs:** nuevos desarrollos de NICs, tanto de Intel como de otros fabricantes, tienen interesantes características que las hacen interesantes de integrar en HPCAP en el futuro. Estas características incluyen anillos de descriptores más grandes o un manejo más avanzado de las funciones virtuales. Adicionalmente, los resultados aquí presentados pretenden servir de base para llevar a cabo desarrollos de mayores prestaciones como tarjetas de 40 y 100 Gb/s.
- **Emisión de paquetes optimizada:** el motor de captura HPCAP y sus derivados se han centrado en la recepción de paquetes de red. No obstante, los mismos conceptos aquí expuestos son aplicables a la hora de optimizar la emisión de paquetes por la red. De hecho, esta configuración podría se enfocada de diversas formas:
  - Centrándose en el ancho de banda: obtener el máximo ancho de banda a la hora de enviar trazas de paquetes previamente almacenadas. Esta aproximación es interesante para realizar pruebas de stress de redes de comunicación.
  - Centrándose en precisión temporal: cuanto más elevada sea la velocidad de red, más elevadas se vuelven las restricciones temporales para el procesamiento de cada paquete, y la temporización se vuelve un tema más crucial. Tener un método preciso de definir tiempos entre la salida de los paquetes es vital con vistas a reproducir escenarios de fallo en la red.
  - Generación de tráfico sintético: diversas aplicaciones de alto nivel no requieren la generación de tráfico con elevadas velocidades, sino que requieren un alto control sobre el tráfico generado. Se propone por tanto la creación de un sistema flexible sobre HPCAP que permita reproducir tráfico con diversas características (tamaños de paquetes, quíntuplas, tiempos entre salida de paquetes, ...), que sería de gran utilidad para la realización de tareas de monitorización activa de redes.
- **Análisis de otras fuentes de error en el timestamp:** la dependencia de las técnicas de marcado de los relojes del sistema y las políticas de planificación del sistema suponen la aparición de diferentes fuentes de error. Se propone realizar un estudio más profundo sobre las escalas de tiempo en las que aparecen estos errores, dependiendo de diferentes factores

de carga del sistema. Dicho estudio ayudaría en la elaboración de una política que establezca una relación entre la carga del sistema y la precisión del marcado, de cara a que los usuarios finales sean conscientes de los niveles en los que se encuentran. Asimismo, creemos que existe una fuerte autocorrelación en los errores de marcado experimentados. El análisis de esta naturaleza cíclica de las autocorrelaciones ayudaría a decidir los intervalos de tiempo en los cuales se puede garantizar una precisión en el marcado. Este estudio sería muy atractivo de cara a técnicas de monitorización activa basadas en el envío de pares, grupos, ó trenes de paquetes en una red.

- **Descubrimiento de nuevas posibilidades de virtualización:** cada vez es más común que los nuevos dispositivos hardware incluyan soporte para virtualización. Concretamente, a lo largo de este estudio no se ha tenido acceso a una controladora RAID con soporte para SR-IOV, pero consideramos muy interesante completar nuestros experimentos incluyendo esta opción. También se considera interesante la exploración de la posibilidades para NFV que ofrecen otras arquitecturas hardware como las FPGAs, GPGPUs, etc.
- **Superar las limitaciones actuales de HPCAP:** a lo largo de esta tesis se han expuesto los pros y contras de HPCAP. Sin embargo, a la hora de implantar HPCAP en entornos industriales hemos encontrado que todavía hay trabajo importante que debe ser realizado:
    - cuando HPCAP trabaja con varias NICs a la vez, el tamaño del buffer a nivel del kernel sigue siendo de 1 GB. Aunque se ha hecho que sea fácilmente configurable la repartición de ese buffer entre las diferentes NIC activas con proporciones a elección del usuario, la drástica reducción que supone en el tamaño del buffer tiene efectos negativos en el rendimiento de la captura de las interfaces. Se sugiere por tanto incluir soporte para definir regiones de buffer utilizando las *"hugepages"* de Linux, que eliminaría esta restricción.
    - Si la cantidad de tráfico de red que se desee guardar es elevada, sólo se podrá mantener durante un corto periodo en los dispositivos de almacenamiento. Por ello, se propone el desarrollo de políticas que permitan decidir si los datos de un paquete son interesantes ó no de almacenar, o incluso cuántos datos de cada paquete interesa almacenar, y alargar por tanto el ciclo de vida de los datos con valor.

Algunas de las líneas de trabajo futuro presentadas ya se han empezado a desarrollar.

Bibliography

[AFG+10] M. Armbrust, A. Fox, R. Griffith, A.D. Joseph, R. Katz, A. Konwinski, G. Lee, D. Patterson, A. Rabkin, I. Stoica, and M. Zaharia, *A view of cloud computing*, Commun. ACM **53** (2010), no. 4, 50–58. 1.1, 6

[AGMM12] G. Antichi, S. Giordano, D.J. Miller, and A.W. Moore, *Enabling open-source high speed network monitoring on NetFPGA*, Proceedings of IEEE/IFIP Network Operations and Management Symposium, 2012. 1.1

[ALN12] S. Alcock, P. Lorier, and R. Nelson, *Libtrace: A packet capture and analysis library*, ACM SIGCOMM Computer Communication Review **42** (2012), no. 2, 42–48. 1.1

[Bar97] M. Barabanov, *A linux-based real-time operating system*, Ph.D. thesis, New Mexico Institute of Mining and Technology, 1997. 4.2.1

[BDKC10] L. Braun, A. Didebulidze, N. Kammenhuber, and G. Carle, *Comparing and improving current packet capturing solutions based on commodity hardware*, Proceedings of ACM Internet Measurement Conference, 2010. 1.1, 4.3, 5.1, 6.1

[BDPGP12] N. Bonelli, A. Di Pietro, S. Giordano, and G. Procissi, *On multi-gigabit packet capturing with multi-core commodity hardware*, Proceedings of Passive and Active Measurement Conference, 2012. 3.3, 3.3.4

[BGPA14] N. Bonelli, S. Giordano, G. Procissi, and L. Abeni, *A purely functional approach to packet processing*, Proceedings of the Tenth ACM/IEEE Symposium on Architectures for Networking and Communications Systems (New York, NY, USA), ANCS '14, ACM, 2014, pp. 219–230. 3.3.4

[BRE+15] A. Beifuß, D. Raumer, P. Emmerich, T.M. Runge, F. Wohlfart, B.E. Wolfinger, and G. Carle, *A study of networking software induced latency*, 2nd International Conference on Networked Systems, 2015. 4.2.1

[BRV09] T. Broomhead, J. Ridoux, and D. Veitch, *Counter availability and characteristics for feed-forward based synchronization*, Proceedings of IEEE Symposium on Precision Clock Synchronization for Measurement Control and Communication, 2009. 4.2.1

[BY96] M. Barabanov and V. Yodaiken, *Real-time linux*, Linux Journal **23** (1996). 4.2.1

[BYBF+12] M. Ben-Yehuda, E. Borovik, M. Factor, E. Rom, A. Traeger, and B. Yassour, *Adding advanced storage controller functionality via low-overhead virtualization*, USENIX Conference on File & Storage Technologies (FAST), 2012. 6.2.2

[CAI] CAIDA, *Traffic analysis research*, http://www.caida.org/research/traffic-analysis/ [14th April 2015]. 3.4.2

[CCA+11] A. Canis, J. Choi, M. Aldham, V. Zhang, A. Kammoona, J.H. Anderson, S. Brown, and T. Czajkowski, *Legup: High-level synthesis for fpga-based processor/accelerator systems*, Proceedings of the 19th ACM/SIGDA International Symposium on Field Programmable Gate Arrays, 2011. 2.1.2

[CCMO11] G. Costa, A. Cuzzocrea, G. Manco, and R. Ortale, *Data deduplication: A review*, Learning Structure and Schemas from Documents (Marenglen Biba and Fatos Xhafa, eds.), Studies in Computational Intelligence, vol. 375, Springer Berlin Heidelberg, 2011, pp. 385–412. 4.4

[CFH+10] I. Csabai, A. Fekete, P. Haga, B. Hullar, G. Kurucz, S. Laki, P. Matray, J. Steger, G. Vattay, F. Espina, S. Garcia-Jimenez, M. Izal, D. Morato E. Magana, J. Aracil, F.J. Gomez, I. Gonzalez, S. Lopez-Buedo, V. Moreno, and J. Ramos, *ETOMIC Advanced Network Monitoring System for Future Internet Experimentation*, 6th International Conference on Testbeds and Research Infrastructures for the Development of Networks & Communities (TridentCom 2010), 2010. 2.2.1

[CGS14] Y. Chen, J. Guo, and Z. Sun, *CPU-GPU System Designs for High Performance Cloud Computing*, High Performance Cloud Auditing and Applications, Springer Berlin, 2014. 2.1.3

[ČKB+10] P. Čeleda, R. Krejčí, J. Barienčík, M. Elich, and V. Krmíček, *HAMOC—hardware-accelerated monitoring center*, Tech. Report 9/2010, CESNET, 2010, http://http://archiv.cesnet.cz/doc/techzpravy/2010/hamoc/, [14th April

2015]. 5.5

[CKS+09] A. Callado, C. Kamienski, G. Szabo, B. Gero, J. Kelner, S. Fernandes, and D. Sadok, *A survey on Internet traffic identification*, IEEE Communications Surveys & Tutorials **11** (2009), no. 3, 37–52. 3.5.2

[Com14] NetFPGA Community, *NetFPGA Project*, 2014, www.netfpga.org, [14th April 2015]. 2.1.2, 2.2.1, 3.1.1

[CRKH05] J. Corbet, A. Rubini, and G. Kroah-Hartman, *Linux device drivers, 3rd edition*, O'Reilly Media, O'Reilly Media, 2005. 4.1.3

[DAF11] M. Daga, A.M. Aji, and W.C. Feng, *On the efficacy of a fused cpu+ gpu processor (or apu) for parallel computing*, Application Accelerators in High-Performance Computing (SAAHPC), 2011 Symposium on, 2011. 2.1.3

[DAR12] M. Dobrescu, K. Argyraki, and S. Ratnasamy, *Toward predictable performance in software packet-processing platforms*, Proceedings of USENIX Symposium on Networked Systems Design and Implementation, 2012. 3.1.1

[DBS06] J.P. Deschamps, G.J.A. Bioul, and G.D. Sutter, *Synthesis of arithmetic circuits*, John Wiley & Sons, Inc., 2006. 2.1.1

[DCF13] L. Deri, A. Cardigliano, and F. Fusco, *10 Gbit line rate packet-to-disk using n2disk*, Proceedings of Traffic Monitoring and Analysis Workshop, 2013, pp. 441–446. 4.3.3

[DDDS11] M. Danelutto, L. Deri, and D. De Sensi, *Network monitoring on multicores with algorithmic skeletons*, Proceedings of International Conference on Parallel Computing (PARCO), 2011. 3.5.2

[DdH+12] M. Dusi, N. d'Heureuse, F. Huici, A. di Pietro, N. Bonelli, G. Bianchi, B. Trammell, and S. Niccolini, *Blockmon: Flexible and high-performance big data stream analytics platform and its use cases*, NEC Technical Journal **7** (2012), no. 2, 102–106. 3.5.1, 3.5.2

[DEA+09] M. Dobrescu, N. Egi, K. Argyraki, B.-G. Chun, K. Fall, G. Iannaccone, A. Knies, M. Manesh, and S. Ratnasamy, *Routebricks: exploiting parallelism to scale software routers*, Proceedings of ACM SIGOPS Symposium on Operating Systems Principles, 2009. 3.3

[Der04] L. Deri, *Improving passive packet capture: Beyond device polling*, Proceedings of System Administration and Network Engineering Conference, 2004. 3.3

[Der05] _____, *nCap: wire-speed packet capture and transmission*, Proceedings of IEEE/IFIP Workshop on End-to-End Monitoring Techniques and Services, 2005. 3.3

[DKSL04] S. Dharmapurikar, P. Krishnamurthy, T.S. Sproull, and J.W. Lockwood, *Deep packet inspection using parallel bloom filters*, IEEE Micro **24** (2004), no. 1, 52–61. 1.1

[DMLB+13] Sanchez-Roman D., V. Moreno, S. Lopez-Buedo, G. Sutter, I. Gonzalez, F.J. Gomez-Arribas, and J. Aracil, *FPGA acceleration using High-Level Languages of a Monte-Carlo method for pricing complex options*, Journal of Systems Architecture **59** (2013), no. 3. 2.1.2

[DPD15] DPDK, *Data plane development kit*, 2015, http://dpdk.org, [14th April 2015]. 3.3.5

[DPHB+13a] A. Di Pietro, F. Huici, N. Bonelli, B. Trammell, P. Kastovsky, T. Groleat, S. Vaton, and M. Dusi, *Toward composable network traffic measurement*, Proceedings of IEEE INFOCOM, 2013. 3.5.1, 3.5.2

[dPHB+13b] A. di Pietro, F. Huici, N. Bonelli, B. Trammell, P. Kastovsky, T. Groleat, S. Vaton, and M. Dusi, *Toward composable network traffic measurement*, Procedings of IEEE INFOCOM, 2013. 5.5

[EAMEG11] E. El-Araby, S.G. Merchant, and T. El-Ghazawi, *A framework for evaluating high-level design methodologies for high-performance reconfigurable computers*, Parallel and Distributed Systems, IEEE Transactions on (2011). 2.1.2

[EAMEG13] _____, *Assessing productivity of high-level design methodologies for high-performance reconfigurable computers*, High-Performance Computing Using FPGAs, Springer New York, 2013. 2.1.2

[EIV07] A.K. Elmagarmid, P.G. Ipeirotis, and V.S. Verykios, *Duplicate record detection: A survey*, IEEE Transactions on Knowledge and Data Engineering (2007). 4.4

[End14a] Endace, *Endace EMULEX*, 2014, www.endace.com/, [14th

April 2015]. 3.1.1

[End14b] _____, *Packet capture performance evaluation*, 2014, http://www.emulex.com, [14th April 2015]. 1.1

[FD10] F. Fusco and L. Deri, *High speed network traffic analysis with commodity multi-core systems*, Proceedings of ACM Internet Measurement Conference, 2010. 3.3

[FML+03] C. Fraleigh, S. Moon, B. Lyles, C. Cotton, M. Khan, D. Moll, R. Rockell, T. Seely, and S.C. Diot, *Packet-level traffic measurements from the Sprint IP backbone*, IEEE Network **17** (2003), 6–16. 1.1

[FMM+11] A. Finamore, M. Mellia, M. Meo, M.M. Munafò, and D. Rossi, *Experiences of Internet traffic monitoring with Tstat*, IEEE Network **25** (2011), no. 3, 8 –14. 5.5

[FQKYS04] Z. Fan, F. Qiu, A. Kaufman, and S. Yoakum-Stover, *Gpu cluster for high performance computing*, Proceedings of the 2004 ACM/IEEE Conference on Supercomputing, 2004. 2.1.3

[FSLB+14] M. Forconesi, G. Sutter, S. Lopez-Buedo, J.E. Lopez de Vergara, and J. Aracil, *Bridging the gap between hardware and software open-source network developments*, IEEE Network **28** (2014), no. 5, 13–19. 6

[GDFM+12] J. L. García-Dorado, A. Finamore, M. Mellia, M. Meo, and M. Munafò, *Characterization of ISP traffic: Trends, user habits, and access technology impact*, IEEE Transactions on Network and Service Management **9** (2012), no. 2, 142–155. 1.1, 5.1

[GDMR+13] J.L. García-Dorado, F. Mata, J. Ramos, P.M. Santiago del Río, V. Moreno, and J. Aracil, *High-performance network traffic processing systems using commodity hardware*, Data Traffic Monitoring and Analysis, Lecture Notes in Computer Science, vol. 7754, Springer Berlin Heidelberg, 2013, pp. 3–27. 1.1, 4.3, 5.1.2, 6, 6.1

[GDSdRR+14] J. L. García-Dorado, P. M. Santiago del Río, J. Ramos, D. Muelas, V. Moreno, J. E. Lopez de Vergara, and J. Aracil, *Low-cost and high-performance: VoIP monitoring and full-data retention at multi-Gb/s rates using commodity hardware*, International Journal of Network Management **24** (2014), no. 3, 181–199. 3.5.1, 3.5.2, 5.4.2

[GES12] F. Gringoli, A. Este, and L. Salgarelli, *MTCLASS: Traffic classification on high-speed links with commodity hardware*, Proceedings of IEEE Conference on Communications, 2012. 5.5

[GLBS+12] I. Gonzalez, S. Lopez-Buedo, G. Sutter, D. Sanchez-Roman, F.J. Gomez-Arribas, and J. Aracil, *Virtualization of reconfigurable coprocessors in hprc systems with multicore architecture*, Journal of Systems Architecture **58** (2012), no. 6. 2.1.3

[Glo] HiTech Global, *NetFPGA10G*, http://www.hitechglobal.com/boards/PCIExpress_SFP+.htm [14th April 2015]. 4.2.1

[HD10] S. Hauck and A. DeHon, *Reconfigurable computing: the theory and practice of fpga-based computation*, Morgan Kaufmann, 2010. 2.1.2

[Hig13] High Performance Computing and Networking Group, Universidad Autónoma de Madrid (HPCN-UAM), *HPCAP and M^{3}Omon*, 2013, https://github.com/hpcn-uam/, [14th April 2015]. 5.1.2, 5.6

[HJPM10] S. Han, K. Jang, K S Park, and S. Moon, *PacketShader: a GPU-accelerated software router*, Proceedings ACM SIGCOMM, 2010. 3.1, 3.2.1, 1, 5, 7, 3.3, 3.3.2, 3.4.2, 3.5.1, 3.5.2, 4.1.1, 4.2.1, 5.3

[HPC15] HPCAP, *High-performance 10G network capture engine*, 2015, http://github.com/hpcn-uam/HPCAP, [14th April 2015]. iii, iv, v, iii, iv

[HRW14] J. Hwang, K.K. Ramakrishnan, and T. Wood, *NetVM: High performance and flexible networking using virtualization on commodity platforms*, 11th USENIX Symposium on Networked Systems Design and Implementation (NSDI 14) (Seattle, WA), USENIX Association, April 2014, pp. 445–458. 6.2.1

[inf] *The infodups tool for duplicate detection*, https://github.com/Enchufa2/nantools, [14th April 2015]. 4.4.1

[Int12] Intel, *82599 10 Gbe controller datasheet*, 2012, http://www.intel.com/content/www/us/en/ethernet-controllers/82599-10-gbe-controller-datasheet.html, [14th April 2015]. 3.1, 4.4

[Int14a] _____, *Intel Data Plane Development Kit (Intel DPDK) Programmer's Guide*, 2014, http://www.intel.com/content/dam/www/public/us/en/documents/guides/intel-dpdk-programmers-guide.pdf, [14th April 2015]. 3.3.5

[Int14b] _____, *Intel Data Plane Development Kit (Intel DPDK) Release Notes*, 2014, http://www.intel.com/content/dam/www/public/us/en/documents/release-notes/intel-dpdk-release-notes.pdf, [14th April 2015]. 3.3, 3.3.5, 6.1

[Int14c] _____, *IXP4XX Product Line of Network Processors*, 2014, www.intel.com/p/en_US/embedded/hwsw/hardware/ixp-4xx, [14th April 2015]. 3.1.1

[JLM+12] M. Jamshed, J. Lee, S. Moon, I. Yun, D. Kim, S. Lee, Y. Yi, and K.S. Park, *Kargus: a highly-scalable software-based intrusion detection system*, Proceedings of ACM Conference on Computer and Communications Security, 2012. 5.5

[KC00] K. Keutzer and D. G. Chinnery, *Closing the gap between asic and custom: an asic perspective*, Design Automation Conference, 2000. 2.1.1

[KKA13] A.O. Kudryavtsev, V.K. Koshelev, and A.I. Avetisyan, *Prospects for virtualization of high-performance x64 systems*, Programming and Computer Software **39** (2013), no. 6, 285–294 (English). 6.2

[KKH+04] M.S. Kim, H.-J Kong, S.-C. Hong, S.-H. Chung, and J.W. Hong, *A flow-based method for abnormal network traffic detection*, Proceedings of IEEE/IFIP Network Operations and Management Symposium, 2004. 1.1

[KKL+07] A. Kivity, Y. Kamay, D. Laor, U. Lublin, and A. Liguori, *kvm: the linux virtual machine monitor*, Proceedings of the Linux Symposium (Ottawa, Ontario, Canada), vol. 1, June 2007, pp. 225–230. 6.2.2

[KMC+00] E. Kohler, R. Morris, B. Chen, J. Jannotti, and M.F. Kaashoek, *The Click modular router*, ACM Transactions on Computer Systems **18** (2000), no. 3, 263–297. 3.5.1, 3.5.2

[Kra12] M. Krasnyansky, *UIO-IXGBE*, 2012, https://opensource.

qualcomm.com/wiki/UIO-IXGBE [14th April 2015]. 3.3

[KWH06] M.S. Kim, Y. J. Won, and J. W. Hong, *Characteristic analysis of Internet traffic from the perspective of flows*, Computer Communications **29** (2006), no. 10, 1639–1652. 1.1, 3.4.1

[LCSR11] M. Laner, S. Caban, P. Svoboda, and M. Rupp, *Time synchronization performance of desktop computers*, Proceedings of IEEE Symposium on Precision Clock Synchronization for Measurement Control and Communication, 2011. 4.2.1

[Lin14] Linux Foundation, *NAPI*, 2014, www.linuxfoundation.org/collaborate/workgroups/networking/napi, [14th April 2015]. 3.1.2

[LLC+12] Y.D. Lin, P.C. Lin, T.H. Cheng, I.W. Chen, and Y.C. Lai, *Low-storage capture and loss recovery selective replay of real flows*, IEEE Communications Magazine **50** (2012), no. 4, 114–121. 4.3

[LLK14] S. Lee, K. Levanti, and H.S. Kim, *Network monitoring: Present and future*, Computer Networks **65** (2014), no. 1, 84–98. 1.1

[Lov02] R. Love, *Kernel korner: Kernel locking techniques*, Linux Journal (2002). 4.1.2

[LSBG13] B. Li, J. Springer, G. Bebis, and M. H. Gunes, *A survey of network flow applications*, Journal of Network and Computer Applications **36** (2013), no. 2, 567 – 581. 1.1, 5.1.1

[LSI14] LSI, *APP3000 Network*, 2014, http://www.lsi.com/products/mobile-communication-processors/pages/app-network-processors.aspx, [14th April 2015]. 3.1.1

[Luc14] Alcatel Lucent, *FP3: Breakthrough 400G network processor*, 2014, http://www3.alcatel-lucent.com/products/fp3/, [14th April 2015]. 3.1.1

[LXB11] G. Liao, Znu. X., and L. Bnuyan, *A new server I/O architecture for high speed networks*, Proceedings of Symposium on High-Performance Computer Architecture, 2011. 3.2.1, 1, 3, 4

[MAB+08] N. McKeown, T. Anderson, H. Balakrishnan, G. Parulkar, L. Peterson, J. Rexford, S. Shenker, and J. Turner, *Openflow: enabling innovation in campus networks*, ACM SIGCOMM Com-

puter Communication Review **38** (2008), no. 2, 69–74. 1.1

[MGAG+11] V. Moreno, F.J. Gomez-Arribas, I. Gonzalez, D. Sanchez-RomÃ¡n, D. Sutter, and S. Lopez-Buedo, *Comparativa del uso de HLLs en FPGA, GPU y Multicore para la aceleracion de una aplicacion de red IP*, XI Edicion Jornadas de ComputaciÃ³n Reconfigurable y Aplicaciones (JCRA2011), 2011. 2.2.2, 4.4

[MGDA12] F. Mata, J. L. García-Dorado, and J. Aracil, *Detection of traffic changes in large-scale backbone networks: The case of the Spanish academic network*, Computer Networks **56** (2012), no. 2, 686 – 702. 5.4.1

[MGGA+11] V. Moreno, J. Garnica, F.J. Gomez-Arribas, S. Lopez-Buedo, I. Gonzalez, J. Aracil, M. Izal, E. Magana, and D. Morato, *High-accuracy network monitoring using etomic testbed*, 7th EURO-NF Conference on Next Generation Internet (NGI 2011), June 2011. 4.2.1

[Mic] Microsoft, *Receive Side Scaling*, `http://msdn.microsoft.com/en-us/library/windows/hardware/ff567236(v=vs.85).aspx` [14^{th} April 2015]. 3.1

[MLCN05] M. Mellia, R. Lo Cigno, and F. Neri, *Measuring IP and TCP behavior on edge nodes with tstat*, Computer Networks **47** (2005), no. 1, 1–21. 1.1

[Mor12] V. Moreno, *Development and evaluation of a low-cost scalable architecture for network traffic capture and storage for 10Gbps networks*, Master's thesis, Universidad Autónoma de Madrid, 2012, `http://www.ii.uam.es/~vmoreno/Publications/morenoTFM2012.pdf`, [14^{th} April 2015]. 3.3, 3.4.2, 6.1

[MPN+10] C.R. Meiners, J. Patel, E. Norige, E. Torng, and A.X. Liu, *Fast regular expression matching using small TCAMs for network intrusion detection and prevention systems*, Proceedings of USENIX Conference on Security, 2010. 1.1

[MRF+13] C. Monsanto, J. Reich, N. Foster, J. Rexford, and D. Walker, *Composing software-defined networks*, Proceedings of the 10th USENIX Conference on Networked Systems Design and Implementation (Berkeley, CA, USA), nsdi'13, USENIX Association, 2013, pp. 1–14. 1.1, 6

[MSD+08] G. Maier, R. Sommer, H. Dreger, A. Feldmann, V. Paxson, and F. Schneider, *Enriching network security analysis with time travel*, ACM SIGCOMM, 2008, pp. 183–194. 4.2, 4.3, 5.5, 6.1

[MSdRR+12] V. Moreno, P. M. Santiago del Río, J. Ramos, J.J. Garnica, and J. L. García-Dorado, *Batch to the future: Analyzing timestamp accuracy of high-performance packet I/O engines*, IEEE Communications Letters **16** (2012), no. 11, 1888–1891. 1.1, 4, 3.4.1, 3.4.2, 6.1

[MSdRR+14a] V. Moreno, P. M. Santiago del Río, J. Ramos, D. Muelas, J. L. García-Dorado, F. J. Gomez-Arribas, and J. Aracil, *Multi-granular, multi-purpose and multi-Gb/s monitoring on off-the-shelf systems*, International Journal of Network Management **24** (2014), no. 4, 221–234. 3.5.1, 6.1, 6.2.4

[MSdRR+14b] V. Moreno, P.M. Santiago del Rio, J. Ramos, J.L. Garcia-Dorado, I. Gonzalez, F.J. Gomez-Arribas, and J. Aracil, *Packet storage at multi-gigabit rates using off-the-shelf systems*, 16th IEEE International Conference on High Performance Computing and Communications (HPCC2014), August 2014. 3.3, 3.4.2, 3.5.1, 3.5.2, 4.3.3, 6.1

[MW12] G. Motika and S. Weiss, *Virtio network paravirtualization driver: Implementation and performance of a de-facto standard*, Computer Standards & Interfaces **34** (2012), no. 1, 36 – 47. 6.2.2

[NA08] T.T.T. Nguyen and G. Armitage, *A survey of techniques for Internet traffic classification using machine learning*, IEEE Communications Surveys & Tutorials **10** (2008), no. 4, 56–76. 3.5.2

[Nap10] Napatech Inc., *A guide to building high performance capture and replay appliances*, Tech. report, 2010. 4.3

[nau13] naudit, *Detect-Pro*, 2013, `http://www.naudit.es/index.php?s=3&p=5&l=1`, [14th April 2015]. 5.1.2

[Nau15] *Naudit High Performance Computing and Networking*, 2015, `http://www.naudit.es/`, [14th April 2015]. 7.2, 8.2

[ND10a] J. Nickolls and W.J. Dally, *The gpu computing era*, Micro, IEEE (2010). 2.1.3

[ND10b] _____, *The GPU computing era*, IEEE Micro **30** (2010), no. 2, 56–69. 3.5.1, 3.5.2

[net14a] netmap, *The fast packet I/O framework*, 2014, http://info.iet.unipi.it/~luigi/netmap, [14th April 2015]. 3.3.3

[Net14b] Network Functions Virtualisation (NFV) ETSI Industry Specification Group (ISG), *Network Functions Virtualisation (NFV); NFV Performance & Portability Best Practises*, Tech. report, ETSI, 2014. 6

[nto14] ntop, *Libzero for DNA*, 2014, www.ntop.org/products/pf_ring/libzero-for-dna/, [14th April 2015]. 3.3.1

[OHL+08] J.D. Owens, M. Houston, D. Luebke, S. Green, J.E. Stone, and J.C. Phillips, *Gpu computing*, Proceedings of the IEEE (2008). 2.1.3

[Pac12] PacketShader, *Packet I/O Engine*, 2012, http://shader.kaist.edu/packetshader/io_engine/index.html, [14th April 2015]. 3.3.2

[PFQ15] PFQ, *PFQ homepage*, 2015, http://netserv.iet.unipi.it/software/pfq, [14th April 2015]. 3.3.4

[PMAO04] M. Polychronakis, E.P. Markatos, K.G. Anagnostakis, and A Oslebo, *Design of an application programming interface for IP network monitoring*, Proceedings of IEEE/IFIP Network Operations and Management Symposium, 2004. 1.1

[PP11] Y. Park and K.H. Park, *High-performance scalable flash file system using virtual metadata storage with phase-change ram*, IEEE Transactions on Computers **60** (2011), no. 3, 321–334. 4.3.2

[PT05] D. Pellerin and S. Thibault, *Practical fpga programming in c*, Prentice Hall Press, 2005. 2.1.2

[PVA+12] A. Papadogiannakis, G. Vasiliadis, D. Antoniades, M. Polychronakis, and E.P. Markatos, *Improving the performance of passive network monitoring applications with memory locality enhancements*, Computer Communications **35** (2012), no. 1, 129–140. 3.2.1

[RCC12] L. Rizzo, M. Carbone, and G. Catalli, *Transparent acceleration of software packet forwarding using netmap*, Proceedings of IEEE INFOCOM, 2012. 3.5.1, 3.5.2

[RDC12] L. Rizzo, L. Deri, and A. Cardigliano, *10 Gbit/s line rate packet processing using commodity hardware: survey and new proposals*, 2012, Online: http://luca.ntop.org/10g.pdf [14th April 2015]. 1.1, 3.3, 3.3.1

[Riz12a] L. Rizzo, *netmap: a novel framework for fast packet I/O*, Proceedings of USENIX Annual Technical Conference, 2012. 3.1.2, 3.2.1, 3, 3.3, 3.3.3, 3.4.2

[Riz12b] _____, *Revisiting network I/O apis: The netmap framework*, ACM Queue **10** (2012), no. 1, 30–39. 3.3

[Riz12c] _____, *Revisiting network I/O APIs: the netmap framework*, Communications of the ACM **55** (2012), no. 3, 45–51. 4.1.1

[Riz14] _____, *Portable packet processing modules for OS kernels*, IEEE Network **28** (2014), no. 2, 6–11. 3.3

[RLM13] L. Rizzo, G. Lettieri, and V. Maffione, *Speeding up packet i/o in virtual machines*, Proceedings of the Ninth ACM/IEEE Symposium on Architectures for Networking and Communications Systems (Piscataway, NJ, USA), ANCS '13, IEEE Press, 2013, pp. 47–58. 6.2.2

[Rus08] R. Russell, *Virtio: Towards a de-facto standard for virtual i/o devices*, SIGOPS Operating Systems Review **42** (2008), no. 5, 95–103. 6.2.2

[Sch12] H. Scholz, *IETFInternet-Draft: RTP stream information export using IPFIX*, Tech. report, Network Working Group, 2012. 5.4.2

[SdRRG+12] P. M. Santiago del Río, D. Rossi, F. Gringoli, L. Nava, L. Salgarelli, and J. Aracil, *Wire-speed statistical classification of network traffic on commodity hardware*, Proceedings of ACM Internet Measurement Conference, 2012. 3.5.1, 3.5.2, 5.5

[SE10] J. Sanders and Kandrot E., *Cuda by example: An introduction to general-purpose gpu programming*, Addison Wesley, 2010. 2.1.3

[SG08] Ilger M. Stampfel G, Gansterer WN, *Implications of the EU data retention directive 2006/24/EC*, Proceedings of Sicherheit, 2008. 5.4.2

[SGV+10] G. Szabó, I. Gódor, A. Veres, S. Malomsoky, and S. Molnár,

Traffic classification over Gbit speed with commodity hardware, Journal of Communications Software and Systems **5** (2010), no. 3, –. 3.5.1, 3.5.2

[SWF07] F. Schneider, J. Wallerich, and A. Feldmann, *Packet capture in 10-Gigabit Ethernet environments using contemporary commodity hardware*, Proceedings of Passive and Active Measurement Conference (PAM'13), 2007. 3.5.2

[Sys12] Cisco Systems, *White paper: Introduction to Cisco IOS NetFlow*, 2012, http://www.cisco.com/c/en/us/products/ios-nx-os-software/ios-netflow/white-paper-listing.html, [14th April 2015]. 3.5.2

[Sys13] _____, *Cisco network convergence system*, 2013, http://www.cisco.com/en/US/products/ps13132/index.html, [14th April 2015]. 1.1, 5.1.2

[Sys14] _____, *Network Analysis Module (NAM) Products*, 2014, www.cisco.com/go/nam, [14th April 2015]. 3.1.1

[SZTG12] W. Su, L. Zhang, D. Tang, and X. Gao, *Using direct cache access combined with integrated NIC architecture to accelerate network processing*, Proceedings of IEEE Conference on High Performance Computing and IEEE Conference on Embedded Software and Systems, 2012. 7

[UAM] *Campus de excelencia internacional UAM-CSIC*, http://campusexcelencia.uam-csic.es/, [14th April 2015]. 7.2, 8.2

[UMMI13] I. Ucar, D. Morato, E. Magana, and M. Izal, *Duplicate detection methodology for ip network traffic analysis*, IEEE International Workshop on Measurements and Networking, 2013. 2.2.2, 4.4

[VPI11] G. Vasiliadis, M. Polychronakis, and S. Ioannidis, *MIDeA: a multi-parallel intrusion detection architecture*, Proceedings of ACM Conference on Computer and Communications Security, 2011. 3.5.1, 3.5.2, 5.5

[WAcAa] C. Walsworth, E. Aben, k.c. claffy, and D. Andersen, *The CAIDA anonymized 2009, 2012 and 2014 Internet traces*, http://www.caida.org/data/passive/passive_2009_dataset.xml; http://www.caida.org/data/passive/passive_2012_dataset.xml;http:

//www.caida.org/data/passive/passive_2014_dataset.xml [14th April 2015]. 3.4.2, 4.2.1, 5.3, 5.4.1, 6.1

[WACAb] C. Walsworth, E. Aben, k.c. Claffy, and D. Andersen, *The CAIDA anonymized 2009 Internet traces*, http://www.caida.org/data/passive/passive_2009_dataset.xml, [14th April 2015]. 4.3.1

[WDC11] W. Wu, P. DeMar, and M. Crawford, *Why can some advanced Ethernet NICs cause packet reordering?*, IEEE Communications Letters **15** (2011), no. 2, 253–255. 1.1, 2, 3.4.2, 4.2.1

[WOL02] J. Wolkerstorfer, E. Oswald, and M. Lamberger, *An asic implementation of the aes sboxes*, Topics in Cryptology — CT-RSA 2002, Lecture Notes in Computer Science, Springer Berlin Heidelberg, 2002. 2.1.1

[WP12] S. Woo and K. Park, *Scalable TCP session monitoring with Symmetric Receive-Side Scaling*, Tech. report, KAIST, 2012. 3.1

[Xtr] XtremeData Inc., *XtremeData XD2000i in-socket accelerator*, http://www.altima.co.jp/products/xtremeData/download/XD2000i_PF_WEB.pdf, [14th April 2015]. 2.1.2, 2.2.2

[YHB+11] AJ. Younge, R. Henschel, J.T. Brown, G. von Laszewski, J. Qiu, and G.C. Fox, *Analysis of virtualization technologies for high performance computing environments*, Cloud Computing (CLOUD), 2011 IEEE International Conference on, July 2011, pp. 9–16. 1.1, 6

[YKL04] F. Yu, R.H. Katz, and T.V. Lakshman, *Gigabit rate packet pattern-matching using TCAM*, Proceedings of IEEE Conference on Network Protocols, 2004. 1.1, 6

[YLWH14] C.T. Yang, J.C. Liu, H.Y. Wang, and C.H. Hsu, *Implementation of gpu virtualization using pci pass-through mechanism*, The Journal of Supercomputing **68** (2014), no. 1, 183–213 (English). 6.2.3

[YZ10] J. Yu and X. Zhou, *Ultra-high-capacity DWDM transmission system for 100G and beyond*, IEEE Communications Magazine **48** (2010), no. 4, S56–S64. 1.1

[ZFP12] L. Zabala, A. Ferro, and A. Pineda, *Modelling packet capturing in*

a traffic monitoring system based on Linux, Proceedings of Performance Evaluation of Computer and Telecommunication Systems, 2012. 3.1.2

GLOSSARY

— A —
accuracy, 86, 114
affinity, 39, 54, 83, 134
ASIC, 10
atomic operations, 81

— B —
batch, 38, 87
buffer, 80

— C —
cache, 34, 40, 102, 103

— D —
duplicate, 112
duplicates, 18

— F —
flow record, 23, 123, 125, 130, 131, 134, 137
FPGA, 11
FPR, 115
full virtualization, 155

— G —
GPGPU, 12

— H —
hash, 23, 112
HLL, 12
hypervisor, 153

— K —
KPT, 78, 91

— L —
listener, 42, 81, 130, 132

— M —
MRTG, 124, 131, 132, 136

— N —
network stack, 28
NTSS, 109
NUMA, 169

— P —
passthrough, 159
polling, 78, 91

— R —
RAID, 101, 157, 165

— S —
SSD, 101, 106, 108
stream, 38, 80

— T —
tcpdump, 100
timestamp, 15, 86
TPR, 115

— U —
UDTS, 88

— V —
virtio, 155
virtual function, 160
VNMA, 168
VNP, 164
VoIP, 139

— W —
WDTS, 89

A

Capture engines' usage examples

A.1 Getting started

Prior to setting up capture engines, some knowledge of the system architecture must be obtained in advance to exploit NUMA capabilities and perform and optimize scheduling.

The first step consists in getting an overview of the NUMA architecture and the devices attached to each node. To this end, the `lstopo` command should be used.

As can be observed in Listing A.1 the command returns a text output with a tree scheme describing each NUMA node and the devices attached to it. Note that each output line containing L# references a processing core. In the example shown it can be observed that two processors (sockets) are present. Each processor is assigned to a NUMA node with 6 processing cores and 64 GB of memory. Focusing on the NICs, two different NICs with two interfaces each are attached to the NUMA node 0. In this case, the system presents an architecture similar to Fig. 3.2(b) where PCIe lines are directly connected to one processor. In this scenario, our 10 GbE NIC corresponds to the interfaces `eth2` and `eth3` which are assigned to NUMA node 0. Conversely, packet capture and processing tasks must be assigned to cores 0 to 5 in order to exploit memory locality.

It is worth pointing out that storage hardware is attached to NUMA node 1 (a RAID-0 volume recognized as `sda`). If packet storage is needed. a processing overhead may exist as data has to be transferred from processor 0 where traffic is received to processor 1 where traffic is stored. The `numactl` command can be used to get an idea of how expensive this data transfer is in computational terms. Code A.2 shows the execution and output of this command. The output shows the available NUMA nodes specifying the processing cores and memory assigned as well as the distance matrix. Note that the figures shown in the distance matrix do not correspond to CPU cycles nor time measurements. Such numbers only provide a priority relationship where the higher value the slower the processor's access to that memory chunk.

```
> lstopo
  Machine (128GB)
    NUMANode L#0 (P#0 64GB)
    Socket L#0 + L3 L#0 (15MB)
      L2 L#0 (256KB) + L1 L#0 (32KB) + Core L#0 + PU L#0 (P#0)
      L2 L#1 (256KB) + L1 L#1 (32KB) + Core L#1 + PU L#1 (P#1)
      L2 L#2 (256KB) + L1 L#2 (32KB) + Core L#2 + PU L#2 (P#2)
      L2 L#3 (256KB) + L1 L#3 (32KB) + Core L#3 + PU L#3 (P#3)
      L2 L#4 (256KB) + L1 L#4 (32KB) + Core L#4 + PU L#4 (P#4)
      L2 L#5 (256KB) + L1 L#5 (32KB) + Core L#5 + PU L#5 (P#5)
      HostBridge L#0
        PCIBridge
          PCI 8086:1521
            Net L#0 "eth0"
        PCI 8086:1521
            Net L#1 "eth1"
        PCIBridge
          PCI 8086:10fb
            Net L#2 "eth2"
          PCI 8086:10fb
            Net L#3 "eth3"
        PCIBridge
          PCI 8086:1d6b
        PCIBridge
          PCI 102b:0532
          PCI 8086:1d02
    NUMANode L#1 (P#1 64GB)
    Socket L#1 + L3 L#1 (15MB)
      L2 L#6 (256KB) + L1 L#6 (32KB) + Core L#6 + PU L#6 (P#6)
      L2 L#7 (256KB) + L1 L#7 (32KB) + Core L#7 + PU L#7 (P#7)
      L2 L#8 (256KB) + L1 L#8 (32KB) + Core L#8 + PU L#8 (P#8)
      L2 L#9 (256KB) + L1 L#9 (32KB) + Core L#9 + PU L#9 (P#9)
      L2 L#10 (256KB) + L1 L#10 (32KB) + Core L#10 + PU L#10
          (P#10)
      L2 L#11 (256KB) + L1 L#11 (32KB) + Core L#11 + PU L#11
          (P#11)
      HostBridge L#5
        PCIBridge
          PCI 1000:005b
            Block L#4 "sda"
```

Code A.1: Usage and output examples of the `lstopo` command

```
>numactl --hardware

available: 2 nodes (0-1)
node 0 cpus: 0 1 2 3 4 5
node 0 size: 65503 MB
node 0 free: 61844 MB
node 1 cpus: 6 7 8 9 10 11
node 1 size: 65536 MB
node 1 free: 60506 MB
node distances:
node 0 1
  0: 10 21
  1: 21 10
```

Code A.2: Usage and output examples of the `numactl` command

Taking this memory access latency matrix into account, the memory allocation of the processes can be adjusted. Focusing on this example, if a capture or a processing task is assigned to cores 0 to 5, memory should be allocated in NUMA node 0. This configuration can be achieved by either using the `numactl` command or using the `libnuma` C API. Code A.3 gives an example of the former. In this case using the option `--membind` followed by a list of NUMA nodes, the memory used by the `my_program` application will be allocated on NUMA node 0 until no more memory is available and then subsequent allocations will use NUMA node 1. If no memory is available in neither node 0 nor 1 the application will terminate. Note that the execution of a program with `numactl --membind` does not guarantee CPU affinity.

```
numactl --membind=0,1 ./my_program
```

Code A.3: Usage example of the `numactl` command to allocate memory

Using `libnuma`[1], memory allocation can be assigned to a specific NUMA node via programing by using the C API shown in Code A.4

```
void *numa_alloc_local(size_t size);
void *numa_alloc_on_node(size_t size,int node);
```

Code A.4: Libnuma memory allocation API

[1] http://linux.die.net/man/3/numa

Another task of paramount importance when running captures or processing applications is assigning tasks to processing cores. These can be assigned with the `taskset` command. As can be seen in Code A.5, using the parameter `-c` followed by the number of a processing core assigns the execution of `my_program` to the core indicated. If multiple processing cores are needed, a comma separated list can be defined. Additionally, the assignment process can be done programmatically by means of the `pthread` library as shown in Code A.6.

```
1  taskset -c=1 ./my_program
2  taskset -c=1,3,5 ./my_program
```

Code A.5: Usage example of the `taskset` command

```
1  int pthread_setaffinity_np(pthread_t thread, size_t cpusetsize, const
       cpu_set_t *cpuset);
```

Code A.6: `pthread` processes assignment API

The `isolcpus` kernel option may be used to maximize the efficiency of the assignment of tasks to processing cores. This option allows a set of processing cores to be isolated from the general kernel SMP balancing and scheduler algorithms. Thus, the only way a process can be assigned to such a set of cores is by explicitly attaching it through the `taskset` command or similar. With this approach, a capture or processing application can be exclusively assigned to a processing core. Code A.7 shows a sample kernel boot configuration in which processing cores 0,1,2,3,4,5 are isolated.

```
1  linux /boot/vmlinuz-3.8.0-29-generic
       root=UUID=b11691e7-f968-4023-aa28-3f5a4d831fa5
       isolcpus=0,1,2,3,4,5 ro
```

Code A.7: Usage example of the `isolcpu` option

A.2 Setting up capture engines

The goal of this subsection is to provide a quick reference guide for the commands and applications used to configure the driver and receive traffic for each capture engine. All commands shown in this subsection except compilation-related ones must be executed with superuser privileges.

A.2.1 Default driver

First, we describe how the affinity-aware tests have been used in the default mechanism, i.e. the `ixgbe` driver plus the PCAP library. The `ixgbe` driver version used is 3.11.33-k, and the version of the PCAP library is 1.1.1-10. With regard to the number of queues to be used, this value can be modified by means of the `RSS` parameter at driver load time (`insmod` command), as shown in Code A.8. The next step is to wake up the network interface and using the `ethtool` utility to disable pause frames (we do not want the network probe to stop the other side's transmission) and the offload options (in order to prevent the NIC from merging incoming packets into bigger ones and polluting our sampling of the network). Once the number of desired queues has been set, a different core must be assigned to fetch packets from each queue in order to obtain maximum performance. An example of how this can be done using the system `/proc` interface is shown in the usage example. In this scenario, we have set 5 RSS queues to be used, assigning them to cores 0 to 4. Finally, we have developed a simple application that fetches incoming packets and counts them using the PCAP library and we have called it `test`. This application has been scheduled to run on core 5 (still in the same NUMA node as the 5 cores fetching packets from the different queues) via the `taskset` Linux command.

```
cd ixgbe-3.11.33-k/src/
make
insmod ixgbe.ko MQ=1,1 RSS=5,5
#wake up interface in promisc mode
#disable pause negotiation and offload settings
ifconfig eth1 up promisc
ethtool -A eth1 rx off tx off
ethtool -K eth1 tso off gso off gro off lro off
#configure IRQ affinity
core=0
cat /proc/interrupts | grep eth1 |
    awk split($1,a,":");print a[1] |
        while read irq
        do
            echo $core > /proc/${irq}/smp_affinity_list
            core=$(( $core + 1))
        done
taskset -c 5 ./test eth1
```

Code A.8: Usage example of the `ixgbe` driver in an affinity-aware scenario

A.2.2 PF_RING DNA

In the case of PF_RING, we used version 6.0.0.1. Once we have entered the downloaded folder, we must compile both the `pf_ring` and the PF_RING-aware version of our NIC driver (see Code A.9). When both drivers have been compiled, they can be installed using the script provided: `load_dna_driver.sh`. Changing the number of receive queues must be done by editing the driver load script, changing the RSS parameter in the line inserting the `ixgbe` driver into the system. The `load_dna_driver.sh` script also adjusts all interrupt affinity issues for each receive queue. To receive traffic using PF_RING, we executed the `pfcount_multichannel` application. The arguments of this program are as follows: `-i` indicates the device name, `-a` enables active packet waiting, `-e` sets reception only mode and `-g` specifies the thread affinity for the different queues. In the example, only one receive thread mapped to core 0 is used; if more threads are to be used, the core affinity for each one must be separated using ':'.

```
cd PF_RING-6.0.0.1/
cd kernel/
make
cd ../drivers/DNA/ixgbe-*/src/
make
./load_dna_driver.sh
cd ../../../../userland/examples/
pfcount_multichannel -i dna0 -a -e 1 -g 0
```

Code A.9: PF_RING usage example

A.2.3 PacketShader

With respect to PacketShader, its version 0.2 was used. We installed the driver using the `install.py` script provided, whose arguments are the number of RX and TX queues to be used by each NIC controlled by PacketShader. As Code A.10 shows, the engine provides an installation script to decide the number of receive queues, and it configures the interrupt affinity schedule. The bundle downloaded includes a sample application named `rxdump`, designed to dump incoming packet information through the standard output just as `tcpdump` would do. We have slightly modified this sample program so it only receives and counts incoming packets and launched it, with the desired network device as its argument. The execution of this sample program was attached to core 0 via the `taskset` utility as the installation script set the receive queue management to this core.

```
cd io-engine-0.2/
cd driver/
make
./install.py 1 1
cd ../samples/rxdump/
taskset -c 0 ./rxdump <args>
```

Code A.10: PacketShader usage example

A.2.4 netmap

We downloaded the latest version of netmap from the author's github repository, specifically, we downloaded the version committed on April 1, 2014. In order to use it, first both the netmap kernel module and the netmap-aware `ixgbe`

driver must be compiled. Before inserting any of those modules, the user must disable or enable CPUs in the system to accommodate the number of receive queues desired to be used. Note that netmap's default behavior is to use all available CPUs. The sample code shown in Code A.11 shows a way of doing that for one CPU (`num_cpus=1`). Once the corresponding CPUs have been disabled/enabled, the `netmap.ko` and `ixgbe.ko` drivers must be inserted in that order. Now it is time to wake up our interface, disable pause frames and offload settings, and configure interrupt affinity. Finally, the `pkt-gen` sample application can be used to receive network traffic. The `-i` parameter tells the program which device to receive traffic from, and the `-f rx` parameter indicates that the program is to work in rx-only mode. When using this application, the core affinity must be set via the `taskset` utility. Note that the program should be scheduled on the cores the queues' interrupts were previously mapped to. It is important to remember to re-enable all CPUs once you have finished using netmap.

```
1   cd netmap/
2   cd LINUX/
3   make KSRC=/usr/src/linux-kernel-headers-dir
4   #Set the number of active CPUs
5   totall_cpus=12
6   num_cpus=1
7   for i in $(seq 0 $(( $total_cpus - 1 )) )
8   do
9       if [ $i -ge $num_cpus ]
10      then
11              echo 0 | tee /sys/devices/system/cpu/cpu${i}/online
12      else
13              echo 1 | tee /sys/devices/system/cpu/cpu${i}/online
14      fi
15  done
16  insmod netmap_lin.ko
17  cd ixgbe/
18  make
19  insmod ixgbe.ko
20  #wake up interface in promisc mode
21  #disable pause negotiation and offload settings
22  ...
23  #configure IRQ affinity as with plain ixgbe
24  ...
25  cd ../../examples/
26  taskset -c 0 ./pkt-gen -i eth1 -f rx
```

Code A.11: Netmap usage example

A.2.5 PFQ

The version of PFQ, used is the 3.7. First we install the `pfq` driver and then the custom version of `ixgbe` setting the desired amount of queues, which is set to 2 in the example shown in Code A.12. We must then wake up the interface, disable pause frame and offload setting and set interrupt affinity, just as shown before in Code A.8. To receive packets from `eth1` using two queues with the right CPU affinity, we run the `pfq-counters` sample application. This application allows to instantiate different socket groups, each receiving all or a fraction of the traffic assigned to a certain interface. Those groups must be defined with their CPU binding via the `-t` parameter with the following syntax: `sock_id.core.iface[.queue.queue...]`. Where `core` is the CPU in which the thread receiving this socket's traffic will be executed, this core should be mapped not to collide with those configured to run the interface's interrupt code. Note that if no queues are specified, the traffic from all queues belonging to the specified interface will be processed.

```
1   modprobe ioatdma
2   modprobe dca
3   nqueues=2
4   pfq-load -q $nqueues -c pfq.conf
5   #wake up interface in promisc mode
6   #disable pause negotiation and offload settings
7   ...
8   #configure IRQ affinity as with plain ixgbe
9   ...
10  core=$nqueues
11  pfq-counters -c 1514 -t 0.${core}.eth1
```

Code A.12: PFQ usage example

A.2.6 Intel DPDK

We used version 1.6.0r2 of Intel DPDK. As mentioned in the previous section, DPDK uses hugepages in order to gain performance, so the user must boot their system with the proper hugepages options. The example shown in Code A.13 allocates 4 hugepages each of 1 GB. Note that hugepages are evenly distributed between the different NUMA nodes of your system, which in our case means two hugepages per node. After booting the system, a `hugetlbfs` must be mounted for use by DPDK-based applications. Once the user has compiled DPDK's driver, both the system's `uio` and the compiled `igb_uio.ko` drivers must be loaded in that order. Intel's documentation encourages DPDK users to disable CPU frequency scaling governor in order to avoid performance losses

due to power saving adjustments. Code A.13 shows a way of disabling it. Finally, the `testpmd` application is executed in interactive mode. Its invocation requires a large set of parameters which includes CPU affinity masks, queue configuration, the number of hugepages and mount point, number of memory channels, ... After properly launching `testpmd`, we must set the rx-only mode and give the capture start order.

```
1  # add to the grub boot line of your kernel
2  # ... default_hugepagesz=1G hugepagesz=1G hugepages=4
3  mount -t hugetlbfs -o pagesize=1G,size=4G none /mnt/huge
4
5  cd dpdk-1.6.0r2
6  cd build/kmod/
7  make
8  modprobe uio
9  insmod igb_uio.ko
10 #properly set CPUs scaling governor
11 for g in /sys/devices/system/cpu/*/cpufreq/scaling_governor
12 do
13     echo performance > $g
14 done
15 cd ../app/
16 #launch testpmd application
17 ./testpmd <... parameter list ...>
18 testpmd> set fwd rxonly
19 testpmd> start
```

Code A.13: Intel DPDK usage example

A.2.7 HPCAP

Finally, we have used the version 4 of the HPCAP capture engine. This solution comes with a configuration file that the user may edit to change the engine's settings. A complete documentation of this file can be found in HPCAP's github repository. Once the parameters have been properly set, the user launches the `install_hpcap.bash` script which compiles the code, installs the driver, and configures the interface settings (interrupt affinity included). After running the installation script, users can use the `hpcapdd` sample application to receive traffic from the network, as shown in Code A.14. This application receives both the interface and queue indexes to receive traffic from (in the example, the application will receive traffic from interface `hpcap0`'s queue 0). The third argument is a directory path the program will write the incoming traffic to, but a null value means that nothing will be written (packets will only be captured from the network). CPU-affinty scheduling must be done with the `taskset` command. In

our example, the `hpcapdd` application was scheduled in core 1 because the kernel-level thread was being executed in core 0 (which was set in the configuration file).

```
1  cd HPCAP4
2  #change configuration parameters
3  vi params.cfg
4  ./install_hpcap.bash
5  cd samples/hpcapdd
6  make
7  taskset -c 1 ./hpcapdd 1 0 null
```

Code A.14: HPCAP usage example

B

HPCAP MANUAL

B.1 Using the HPCAP driver

The first step you must follow to use HPCAP in your system is obtain the HPCAP`X` (where `X` = release number) containing all the files related to the corresponding release. Inside this folder you will find:

B.1.1 Installing all the required packages

Inside the `deps` folder you will a `install.bash` script that will install the required packages in a Debian-based distro. It is possible to use both the HPCAP driver and the detect-Pro10G in a different distro, but you will have to manually install the required dependencies.

B.1.2 Configure your installation

All the scripts used for the installation and management of the HPCAP driver make use of the information specified in the `params.cfg` file that is located in the root level of the HPCAP`X` folder. Consequently, this file has to be properly modified so the HPCAP driver and dependant applications can properly run in your system.

Here you can find a list with parameters you must make sure to have properly configured before you run HPCAP:

- `basedir`: this parameter contains the path to your installation. For example, if you install HPCAP in the `home` folder of user `foo` the value that must be written in this file is: `/home/foo/HPCAPX`.
- `nif`: this parameter must contain the number of network interfaces available in your system (only the ones that would be managed by Intel's `ixgbe` driver). This number is ussually two times the number of PCI network

cards plugged in your system (assuming you plug only 2-port cards), but this could vary if you use, for example, 1-port, 4-port network cards.

- `nrxq,ntxq`: number of RSS/Flow-Director queues that you want to use in your 10Gb/s network interfaces for both RX and TX purposes. The default value for this parameter is 1, which is recommended to be kept unless you know what changing this value implies.
- `ifs`: this is the list of interfaces that you want to be automatically woken up once the HPCAP driver has been installed. For each of those interfaces a monitoring script will be launched and will keep record of the received and lost packets and bytes (inside the `data` subfolder, see B.1.4). Check B.1.3 for more information regarding the interface naming and numbering policy. **Warning:** only a subset those interfaces configured to work in HPCAP mode should appear on this list (no standard-working interfaces).
- Interface-related parameters: those parameters must be configured for each one of the interfaces listed by the `ifs` parameter. All those parameters follow the format `<param name><itf index>` where `<itf index>` is the number identifying the index regardless the prefix to that number in the system's interface name (see B.1.3). Those parameters are:
  - `mode<itf index>`: this parameter changes the working mode of the interface between HPCAP mode (when the value is 2) and standard mode (if the value is 1). Note that an interface working in standard mode will not be able to fetch packets as interface in HPCAP mode would be able to, but the standard mode allows users to use for TX purposes (E.g.: launching a `scp` through this interface). An interface working in standard mode will no be able to be benefited byt the usage of the `detect-Pro10G` versions that can be found in the `samples` subfolder.
  - `core<itf index>`: this parameter fixes the processor core of the machine that will poll the interface for new packets. As this poll process will use the 100% of this CPU, affinity issues must be taken into account when executing more applications, such as detect-Pro10G. For further information see B.4.
  - `vel<itf index>`: this parameter allows the user to force the link speed that will be negotiated for each interface. Allowed values are 1000 and 10000.
  - `caplen<itf index>`: this parameter sets the maximum amount of bytes that the driver will fetch from the NIC for each incoming packet.
  - `pages<itf index>`: amount of kernel pages to be assigned to this interface's kernel-level buffer. The installation script will check

the amount of pages to be used and make sure the sum of the pages used by all interfaces in HPCAP mode is the total. If this condition is not met, the installation script will issue an error message with useful information for changing this configuration.

Warning: changing any of the above mentioned parameters will take no effect until the driver is re-installed.

Once all the configuration parameters have been properly set, the execution of the script `install_hpcap.bash` will take charge of all the installation steps that need to be made in order to install the HPCAP driver.

B.1.3 Interface naming and numbering

This version of the driver allows choosing whether each interface is to work in HPCAP or standard (traditional, `ixgbe`-alike) modes. Consequently, a decision regarding the naming policy for those interfaces was made.

The target of this policy was always being able to identify each of the interfaces of our system regardless the mode it is working on (so you will be able to know which `<param name><itf index>` of the `params.cfg` file maps to each interface). This leaded to each interface being named as `<mode identifyer><interface index>`, where:

- `<mode identifyer>`: can be `hpcap` when the interface is working in the HPCAP mode, or `xgb` if the interface works in standard mode.
- `<interface index>`: this number will always identify the same interface regardless its working mode. For example, if the first interface found in the system is told to work in standard mode and second interface in the HPCAP mode, you will see that your system has the `xgb0` and `hpcap1` interfaces. If you revert the working mode for such interfaces you will then find interfaces named `hpcap0` and `xgb1`.

B.1.4 Per-interface monitored data

Once the driver has been properly installed, a monitoring script will be launched for each of the interfaces specified in the `ifs` configuration parameter. The data generated by those monitoring scripts can be found in the `data` subfolder. In order to avoid the generation of single huge files, the data is stored in subfolders whose names follow the format `<year>-<week number of year>`.

Inside each of those `<year>-<week number of year>` subfolders, you

will find a file for each of the active interface plus a file with CPU and memory consumption related information.

On the one hand, the fields appearing in each of the `hpcapX` files are:

```
<timestamp> <RX bps> <RX pps> <lost bps estimate> <lost pps>
```

On the other hand, the fields of the `cpus` file are:

```
<timestamp> <%-of-used-CPU(one for each CPU)> <total memory> <used memory> <free memory> <cached memory>
```

B.1.5 Waking up an interface in *standard* mode

If a interface configured to work in standard mode wants to be configured, the `wake_standard_iface.bash` script is provided. Its usage is the following:

```
./wake_standard_iface.bash <iface> <ip addr> <netmask> <core> [<speed, default 10000>]
```

Where:

- `iface` is the interface you want to wake up. As this is thought for interfaces working in standard mode, the interface should be some `xgbN`.
- `<ip addr> <netmask>` are the parameters needed to assign an IP address and a network mask for this interface.
- `core` is the processor where all the interrupts regarding this will be sent, so we can make sure that such interrupts do not affect the capture performance of an HPCAP interface.
- `speed` is an optional parameter that allows you to force the link speed of this interface. Its values can be 1000 or 10000.

B.1.6 Sample applications

hpcapdd

`hpcapdd` is a sample program that maps HPCAP's kernel packet buffer for a certain interface and write its contents into the desired destination directory. Note that, in order to obtain maximum performance, the data access is made in a byte-block basis. This byte-block access policy has consequences regarding

its usability, as it must be assured that the application starts running in a correct state. `hpcapdd` will generate data files following the RAW format (see B.2).

`hpcapdd` has been programmed so it preformas an orderly close when receiving a `SIGINT` signal, so it must be ended with:

```
kill -s SIGINT ...
or
killall -s SIGINT ...
```

Importantly, **the HPCAP driver must be re-installed** before launching a new instance of `hpcapdd`.

hpcapdd_p

`hpcapdd_p` is a sample program that maps HPCAP's kernel packet buffer for a certain interface and write its contents into the desired destination directory. Note that `hpcapdd_p` accesses the data in a per-packet rather than in a byte-block basis. This have an effect damaging the write throughput performance but may result of interest as it has some interesting usability effects. `hpcapdd_p` will generate data files following the RAW format (see B.2).

`hpcapdd_p` has been programmed so it preformas an orderly close when receiving a `SIGINT` signal, so it must be ended with:

```
kill -s SIGINT ...
or
killall -s SIGINT ...
```

Differently from `hpcapdd`, `hpcapdd_p` does not require the HPCAP driver to be reinstalled before launching a new instance of the program.

Importantly, if an instance of `hpcapdd` is to be run after closing an instance of `hpcapdd_p`, the HPCAP drriver must be re-intalled. In the opposite case (launching an `hpcapdd` instance and then an `hpcapdd_p` one) no driver re-installation is needed.

B.2 Working with the RAW file format

This section describes the structure of the "raw" format files generated by the usage of the HPCAP driver.

RAW files are generated by the programs that fetch traffic from the network using the `HPCAP` driver (see B.1.6).

B.2.1 File data structures

A raw file is composed by a set of consecutive packets. Each packet is preceded by its corresponding header which contains information related to the packet just as shown in Fig.B.1:

- **Seconds** 4 bytes containing the seconds field of the packet timestamp.
- **Nanoseconds** 4 bytes containing the nanoseconds field of the packet timestamp.
- **Caplen** 2 bytes containing the amount of bytes of the packet included in the file.
- **Len** 2 bytes containing the real size of the packet.

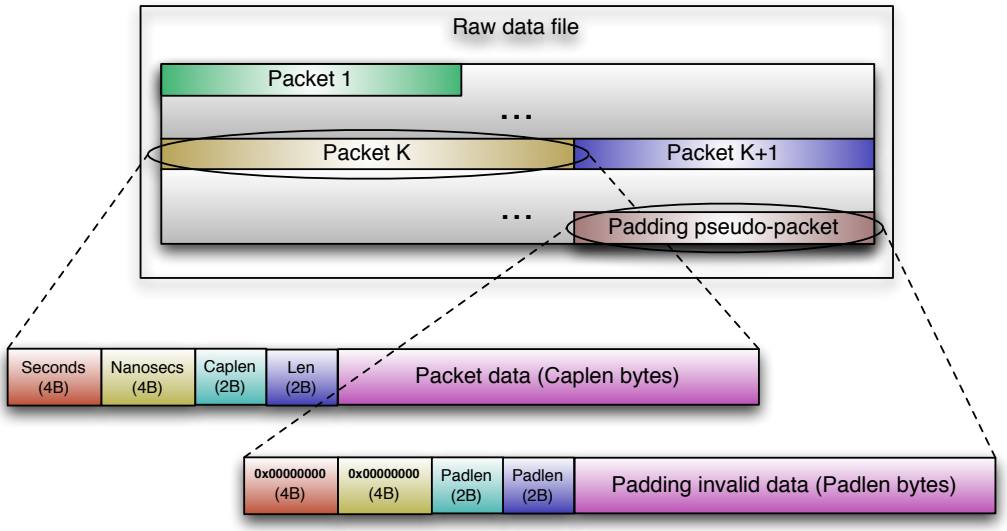


Figure B.1: Raw file format

The end of the file is denoted by the appearance of a pseudo packet showing the amount of padding bytes added at the end of the file (in order to generate files of the same size). The padding pseudo-packet has a similar header than any other packet in the file with the difference that both the "Seconds" and the "Nanoseconds" fields are set to zeros. Once the padding pseudo-packet has been located, the padding size can be read from any of the "Len" or "Caplen" fields. Note that the padding length could be zero.

B.2.2 Example code

The next pages show an example code for a programs that reads a raw file and generates a new pcap file with the contents of the first one.

```
1  while(1)
2  {
3      /* Read packet header */
4      if( fread(&secs,1,sizeof(u_int32_t),fraw)!=sizeof(u_int32_t) )
5      {
6          printf("Segundos\n");
7          break;
8      }
9      if( fread(&nsecs,1,sizeof(u_int32_t),fraw)!=sizeof(u_int32_t) )
10     {
11         printf("Nanosegundos\n");
12         break;
13     }
14     if( (secs==0) && (nsecs==0) )
15     {
16         fread(&caplen,1,sizeof(u_int16_t),fraw);
17         fread(&len,1,sizeof(u_int16_t),fraw);
18         if( len != caplen )
19             printf("Wrong padding format [len=%d,caplen=%d]\n", len,
                   caplen);
20         else
21             printf("Padding de %d bytes\n", caplen);
22         break;
23     }
24
25     if( fread(&caplen,1,sizeof(u_int16_t),fraw)!=sizeof(u_int16_t) )
26     {
27         printf("Caplen\n");
28         break;
29     }
30     if( fread(&len,1,sizeof(u_int16_t),fraw)!=sizeof(u_int16_t) )
31     {
32         printf("Longitud\n");
33         break;
34     }
35     /* Fill header */
36     h.ts.tv_sec=secs;
37     h.ts.tv_usec=nsecs/1000;
38     h.caplen=caplen;
39     h.len=len;
40
41     /* Read packet data*/
42     if( caplen > 0 )
43     {
44         ret = fread(buf,1,caplen,fraw);
45         /* whatever process required*/
46     }
47 }
```

Code B.1: `raw2pcap` code example

B.3 Quick start guide

This section shows how to properly install HPCAP in your system. We will assume you already have a HPCAPX.tar.gz file in your system, and we will show how to continue from there.

1. Check that your kernel is compatible with HPCAP. Nowadays HPCAP has been tested with 2.6.32, 3.2, 3.5 and 3.8 kernels.

2. Save your current network interface mapping so you can easily identify (using the MAC address) which interface is connected to which link:

```
ifconfig -a > old_interfaces.txt
```

3. Decompress the contents of the data file.

```
tar xzvf HPCAPX.tar.gz
```

   This command will create the HPCAPX directory in your system.

4. Enter the HPCAPX directory.

```
cd HPCAPX
```

5. Edit the params.cfg file according to your current configuration[1] (see B.1.2).

6. Install the driver using the installation script (you will need superuser privileges). The script will compile both the driver ans the user-level library if they have not been compiled before.

```
./install_hpcap.bash
```

7. Check that the monitoring script is on by checking the contents of the data files:

```
tail -f data/<year>-<week/hpcapX
```

   Note that traffic counters will not be valid until there is at least one application fetching data from the corresponding (ifterface,queue) pair.

[1] In order to identify the NICs you may need to launch the installation script once in order to use the MAC to identify which interfaces are to work on HPCAP or standard mode, the re-edit the params.cfg file and install the driver using the script again.

B.3.1 Launching hpcapdd

Once you have properly installed HPCAP on your system, you can use hpcapdd (or similar programs) to store the traffic from the network into your system.

1. Make sure you have enough space for traffic storage. it is recommended to use a different volume than the used for your operating system. You can check by executing

```
df -h
```

2. Go to the hpcapdd directory (assuming we are already at the HPCAPX directory).

```
cd samples/hpcapdd
```

3. Launch the application (you will need superuser privileges)

```
taskset -c 1 ./hpcapdd 3 0 /storage
```
[2]

B.3.2 Checking traffic storage

1. Check the counter files in the data subdirectory.
2. In your storage target directory, list the files that have already been written

```
ls -l /storage/*
```

   All of the generated files should have a size of $2\text{GB} = 2147483648$bytes.

3. Convert one file from raw to pcap

```
cd HPCAPX/samples/raw2
./raw2 /storage/<dir>/<raw_file> <pcap_file>
```

   If the capture is being properly made, the program should end showing the message:

```
Padding de XX bytes
```

   with XX being the amount bytes added at the end of the file for obaining a file size multiple of the filesystem's block size.

[2] In this case the application will fetch the traffic arriving to the queue 0 on hpcap3. The core 1 has been chosen in order to not interfere with the kernel-level capture thread and to avoid interception from different NICs, as in this example it is assumed that core 0 has been assigned to the kernel receive thread)

B.4 Frequently asked questions

- **Which Linux kernel versions does HPCAP support?**
  HPCAP has been written for working under diverse Linux kernel/distribution combinations. Specifically, it has been tested for the following versions:
  - **OpenSuse:** 2.6.32 .
  - **Ubuntu/Debian:** 2.6.32, 3.2.0, 3.13.0 .
  - **Fedora:** 3.5.3, 3.14.7 .
- **Should I always use all the available interfaces in `hpcap` mode?**
  No. The reason for this is that the total amount of memory that this driver can use as internal buffer is 1GB, and this amount is divided between the `hpcapX` interfaces present in your system. Thus, if you are not going to capture packets from one interface configure it to work in standard mode and you will have bigger buffers for your capturing interfaces. The amount of memory assigned to each interface out of this 1GB is configured via the `pages` parameter (see B.1.2).
- **Can I use more than one RX queue?**
  If you are going to use Detect-Pro the answer is no. The current version of Detect-Pro support packet capture for just one RX queue. Otherwise you can use more than one RX queue by changing `nrxq` parameter in the `params.cfg` file. Notice that if you use more than one queue per interface you will need to instantiate several packet-consuming applications (at least one per queue) in order to fetch all the data).
- **Can I simultaneously capture data from an interface with `dd` and `hpcapdd`?**
  Yes. In fact, you can use any amount of `dd` or `hpcapdd` instances over the same pair (interface,queue). The limit on the amount of simultaneous applications fetching data from the same pair (interface,queue) is defined in the `include/hpcap.h` file with the `MAX_LISTENERS` constant. Note that a change in this value will take no effect if the driver is not recompiled and re-installed in your system.
- **Is there a way to make sure that my system and user processes will not interfere in the capture process?**
  Yes. First of all, you must make sure of which CPUs you are going to use for the HPCAP driver and Detect-Pro. At this point we have a list of CPUs that we want to isolate from the system scheduler. Now, depending on the distribution we are using:
  - **OpenSuse:** you have to edit the `/boot/grub/menu.lst` file and add into the boot desired command line the following:

```
isolcpus=0,1,2,..,k
```

(with `0,1,2,..,k` are the CPUs to be isolated). The next time you boot your system your changes will have taken effect.

- **Ubuntu/Debian:** you have to edit the `/boot/grub/menu.lst` file, and in the `GRUB_CMDLINE_LINUX_ DEFAULT` parameter add the following:

```
isolcpus=0,1,2,..,k
```

Then, execute `update-grub` and the next time you boot your system your changes will have taken effect.

- **I am not obtaining the expected network capture performance, what is happening**
  Make sure that you have properly set the processor affinity for your storage programs (`dd`, `hpcapdd`) via the `taskset` command. You should check that the assigned processor is not used by any kernel capture thread (the ones specifies by the `coreX` parameters in the `params.cfg` file, see B.1.2).
  Furthermore, you might be obtaining a poor performance because the kernel capture threads are not assigned to the same NUMA node where NIC is connected. In order to check that you should see the content of the file `/sys/bus/pci/devices/<pci_identifier>/numa_node` where `<pci_identifier>` can be obtained searching for your NIC in the `lspci` command's output. This file will tell you the NUMA node[3] you should schedule the capture threads for your NIC (a value of -1 means that there will be no performance difference regardless the NUMA node you use).
  If you're still obtaining a poor performance this may be due to the kernel-level memory buffer being stored in the opposite NUMA node than where the NIC is placed. In such cases you should keep the kernel capture thread in the NUMA node attached to your NIC (via the `coreX` parameter in `params.cfg` file), but should schedule your storage processes in the opposite NUMA node (via the `taskset` command). This way the inter-node memory transfers are minimized and maximum performance is met.
- **I see no traffic from the counters that appear on the interface's counter data files located on `data/<year>-<week>`**
  This is normal behaviour. Until there is at least one application listening, the kernel driver will not fetch packets and thus all the counters will be zero. Nevertheless, if the traffic intensity is very high, it is possible that the lost counter will be incremented but the amount is not to be trusted. This is a known issue that has not been solved yet.

[3] Using the `numactl --hardware` command you can check which processors belong to each NUMA node.

- **Is it possible the size of the RAW files generated by `hpcapdd` or Detect-Pro?**
  Yes, you can. Remember the default file size is 2GB, which was chosen as a tradeoff between write performance ans data accesibility. However, if this file size does not fit your requirements you can change it (altough it is not reccommended due to the delicate operations involved) by editing the following parameter in the `HPCAPX/include/hpcap.h` file:

```
#define HPCAP_BS ( 1048576ul )
#define HPCAP_COUNT ( 2048ul )
#define HPCAP_FILESIZE (HPCAP_BS*HPCAP_COUNT)
```

  Changing the block size (`HPCAP_BS` parameter) is discouraged, so the right way of changing the generated files' size is changing the `HPCAP_COUNT` parameter.
  Important: after changing these parameters (or anyone in the `hpcap.h` header file, it is required to re-compile the driver and any of the sample applications based on HPCAP used.

- **Is there a way to know which NUMA node is my NIC connected to?**
  Yes. The easiest way of obtaining this information is via the `lstopo` Linux command (which requires the installation of the `hwloc` package. An example of this command is Code A.1.
  In this example both NICs are connected to NUMA node 0.
  Other way of getting this information is by means of the `lspci` command and the `/sys/` system's interface. An example of this is the following:

```
1  > lspci
2  ...
3  05:00.0 Ethernet controller: Intel Corporation 82599EB
       10-Gigabit SFI/SFP+ Network Connection (rev 01)
4  05:00.1 Ethernet controller: Intel Corporation 82599EB
       10-Gigabit SFI/SFP+ Network Connection (rev 01)
5  ...
6
7  > cat /sys/bus/pci/devices/0000\:05\:00.0/numa_node
8  1
9  > cat /sys/bus/pci/devices/0000\:05\:00.1/numa_node
10 1
```

  Code B.2: Obtaining NUMA information using `lscpi` and `sysfs`

  In this exampleboth NICs are connected to NUMA node 1.
  Nota that this procedures are also valid for obtaining NUMA-related information for other PCI devices (i.e., an RAIC controller card, etc.).

LISTS

List of codes

4.1 Contents of the `/dev/` directory in a system running HPCAP 82

A.1 `lstopo` command example 210
A.2 `numactl` command example 211
A.3 `numactl` membind option 211
A.4 Libnuma API 211
A.5 `taskset` command example 212
A.6 `pthread` API 212
A.7 `isolcpu` option 212
A.8 `ixgbe` example 214
A.9 PF_RING example 215
A.10 PacketShader example 215
A.11 Netmap example 216
A.12 PFQ example 217
A.13 DPDK example 218
A.14 HPCAP example 219

B.1 `raw2pcap` code example 228
B.2 Obtaining NUMA information using `lscpi` and `sysfs` 233

List of equations

4.1 Packet transfer time for in a 10 Gb/s network 86
4.2 Packet transfer time for 60 byte packets in a 10 Gb/s network 86
4.3 Uniform Distribution of TimeStamp 89
4.4 Weighted Distribution of TimeStamp 91

List of figures

2.1 Argos' block scheme 16
2.2 Packet burst train structure 17
2.3 One-way delay measurements 17
2.4 Twin^{-1}'s architecture 18
2.5 Throughput obtained by different hardware alternatives for duplicate removal over real traffic 19

3.1 RSS architecture 24
3.2 NUMA architectures topology examples 27
3.2 NUMA architectures topology examples 28
3.3 Legacy Linux RX scheme 30
3.4 Linux NAPI RX scheme 32
3.5 Legacy Linux Network Stack (serialized paths) 35
3.6 Optimized Linux Network Stack (independant parallel paths) 37
3.7 PF_RING DNA's RX scheme 44
3.8 PacketShader's RX scheme 46
3.9 PFQ's RX scheme 50
3.10 Intel DPDK's architecture 51
3.11 Engines' performance for worst and average scenarios 57
3.12 Engine's performance comparison 60
3.12 Engine's performance comparison 61

4.1 HPCAP kernel packet buffer 79
4.2 HPCAP kernel packet buffer 80
4.3 HPCAP packet reception scheme 85
4.4 Batch timestamping effect on inter-arrival times 88
4.5 Degradation on timestamping accuracy with batch size 89
4.6 Inter-packet gap distribution 90
4.7 Packet capture performance for different timestamping policies 96
4.8 Effect of packet timestamping on performance 97
4.8 Effect of packet timestamping on performance 98
4.9 tcpdump's packet storage performance 100
4.10 Write throughput scalability for mechanical and solid-state drives 107
4.11 Effect of the FS on write throughput (mechanical drives) 108
4.12 Effect of the FS on write throughput (SSD) 109
4.13 Percentage of stored packets by high-performance NTSS . . . 111
4.14 Duplicate packet removal hash table structure 113
4.15 Duplicate removal performance evaluation 118
4.15 Duplicate removal performance evaluation 119

5.1 Contrast between our approach and a conventional one 126
5.2 M³Omon's Architecture 129
5.3 Network time series during an anomalous event 138
5.4 Flow concurrence and throughput for both directions 139
5.5 SIP VoIP network and VoIPCallMon architectures 140
5.6 Active calls and new calls managed by VoIPCallMon..... 142

6.1 Percentage of packets captured by different solutions 150
6.2 Full-virtualization and paravirtualization 154
6.3 Virtualized I/O device 158
6.4 Packet capture performance for different VF alternatives 162
6.5 Virtual network probe 167
6.6 Network monitoring agent 169

B.1 Raw file format 226

List of tables

2.1 Pros and cons summary for different hardware alternatives . 10

3.1 Comparison of the diverse capture engines 42
3.2 Maximum throughput in a 10GbE link according to packet size 54
3.3 Memory and CPU usage in a 10 Gb/s average scenario 62
3.4 Network applications over novel capture engines 67

4.1 Experimental timestamp error: synthetic traffic 92
4.2 Experimental timestamp error: real traffic 93
4.3 Packet header effect in effective throughput 102
4.4 Write throughput summary results for mechanical drives 104
4.5 Write throughput summary results for solid-state drives 106
4.6 Characteristics of the traces used for duplicate removal testing 114
4.7 Duplicate classification accuracy for different real traces varying the hash table configuration 115

5.1 Packet sniffer and dumper modules performance 134
5.2 Throughput and packet loss of the different modules 135

6.1 Write throughput summary results for a virtualized RAID 0 . . 157
6.2 Bare-metal and passthrough packet capture performance . . . 159
6.3 Packet capture performance using different VF-generators . . 161
6.4 Percentage of packets processed for a VNP 165
6.5 VNMA packet capture performance 169
6.6 VNMA capture and storage performance 170

ACRONYMS

API	Application Program Interface
ASIC	Application-Specific Integrated Circuit
CAPEX	CAPital EXpenditures
CPU	Central Processing Unit
CUDA	Compute Unified Device Architecture
DMA	Direct Memory Access
DNA	Direct NIC Access
DNS	Domain Name System
FPGA	Field Programmable Gate Array
GPGU	General-Purpose Graphic Processing Unit
HDL	Hardware Description Language
HLL	High-Level Language
I/O	Input/Output
IP	Internet Protocol
IPSEC	Internet Protocol security
ISP	Internet Service Provider
IXGBE	Intel's 10 Gigabit Ethernet Linux driver
KPT	Kernel-level Polling Thread
MRTG	Multi-Router Traffic Grapher
NAPI	New API
NIC	Network Interface Card
NTP	Network Time Protocol
NTSS	Network Traffic Storage Solution

NUMA ..... Non Uniform Memory Access

OPEX ..... OPErational EXpenditures

P2P ..... Peer-to-peer

PCAP ..... Packet Capture API

PCI ..... Peripheral Component Interconnect

PCIE ..... Peripheral Component Interconnect Express

PF ..... Physical Function

POSIX ..... Portable Operating System Interface

PSTN ..... Public Switched Telephone Network

PTP ..... Precision Time Protocol

QoE ..... Quality of Experience

QoS ..... Quality of Service

RSS ..... Receive Side Scaling

RTP ..... Real-time Transport Protocol

RX ..... Reception

SIP ..... Session Initiation Protocol

SLA ..... Service Level Agreement

SMP ..... Symmetric Multi Processor

SPMC ..... Single Producer, Multiple Consumer

SSD ..... Solid-Sate Drive

TCP ..... Transmission Control Protocol

TS ..... Timestamp

TX ..... Transmission

UDP ..... User Datagram Protocol

UDTS ..... Uniform Distribution of TimeStamp

VF ..... Virtual Function

VM . . . . . Virtual Machine

VNMA . . . . . Virtual Network Monitoring Agent

VNP . . . . . Virtual Network Probe

VoIP . . . . . Voice over IP

WBC . . . . . Write-Back Cache

WDTS . . . . . Weighted Distribution of TimeStamp

WTC . . . . . Write-Through Cache