Universidad Autónoma de Madrid

Escuela Politécnica Superior

Grado en Ingeniería Informática

TRABAJO DE FIN DE GRADO

GENERACIÓN DE FLUJOS EN REDES MULTIGIGABIT ETHERNET ACELERADA MEDIANTE HARDWARE DEDICADO

Isabel García Contreras
Tutor: Sergio López Buedo

Julio 2015

GENERACIÓN DE FLUJOS EN REDES MULTIGIGABIT ETHERNET ACELERADA MEDIANTE HARDWARE DEDICADO

Autora: Isabel García Contreras
Tutor: Sergio López Buedo

High Performance Computing and Networking
Departamento de Tecnología Electrónica y de las Comunicaciones
Escuela Politécnica Superior
Universidad Autónoma de Madrid

Julio 2015

Agradecimientos

Me gustaría dedicar este trabajo a todas las personas que me han ayudado y apoyado a lo largo de su desarrollo y en los años de carrera.

A José, Rafa y Mario por su ayuda en el trabajo diario.

A mi tutor Sergio, sin el cual este trabajo no habría sido posible y en general a todos miembros del grupo HPCN.

A todos los amigos y compañeros de la carrera con los que a lo largo de los años he compartido risas y quejas.

Por último, a mi familia que siempre se encarga de darme el empujón que necesito a final de curso, especialmente este último año.

Abstract

Abstract — The extended use of the network has made the Internet companies change their strategy about how to analyze their links. On the one hand, the need of more powerful computers and servers arise, which, in turn, increases the economic cost. In order to avoid the use of high-performance devices, arquitectures based on commodity hardware along with FPGAs are sought. On the other hand, they need to characterize the use of their network traffic links to improve the quality of their service and to detect problems or malicious attacks. For that, they analyze the traffic they deal with. Studying it from the point of view of each individual packet results in an infeasible task, for it does not contribute sufficient useful information. The optimum aggregation level for this task on multigigabit links is found to reside in the analysis on the flow level, in which each flow represents a unidirectional connection between two Internet users.

A hybrid system composed of both dedicated hardware and software is presented in this Graduate Project. The developed hardware component is able to capture and analyze the totality of the traffic up to 10 Gbps and 14 Mpps, as well as to classify the packets in subnetworks with no information loss whatsoever. The software subsystem is able to produce its own flows associated with the traffic through the use of the hardware-produced data, with the use of no more than two of the cores of a conventional CPU. These good outcomes are due to the use of DMA transferences. The complete development of this project has been possible thanks to the utilization of HLS tools. Otherwise, it would not have been possible to codify the totality of the project. The hardware development methodology has entailed a realist strategy, technologically and economically oriented, because not only the cost of the FPGA used, but the development time as well have been proportional to the proposed objectives of this project.

Key words — FPGA, Flow Building, Multigigabit Network

Resumen

Resumen — El extendido uso de la red en la actualidad ha significado que las empresas de Internet cambien su estrategia en el análisis de sus enlaces. Por un lado, surge la necesidad de utilizar equipos cada vez más potentes, lo que implica un aumento en el coste. Para evitar el uso de dispositivos de alto rendimiento, se buscan arquitecturas basadas en *commodity hardware* junto con FPGAs. Por otro lado, necesitan caracterizar el uso de sus enlaces para mejorar la calidad de sus servicios y detectar problemas o posibles ataques maliciosos. Para ello realizan un análisis del tráfico que manejan. Estudiar éste a nivel de cada paquete individual es inviable y no aporta la suficiente información relevante. El nivel de agregación óptimo para esta tarea en enlaces multigigabit reside en el análisis a nivel de flujos, donde cada flujo representará una conexión unidireccional entre dos usuarios de Internet.

En este Trabajo de Fin de Grado se presenta un sistema híbrido que integra hardware dedicado con software. El módulo hardware desarrollado es capaz de capturar y analizar el 100 % del tráfico a 10 Gbps y 14 Mpps, así como de clasificar los paquetes en subredes sin que se produzca pérdida de información. El subsistema software es capaz de generar los flujos asociados al tráfico mediante la transferencia de los datos generados en hardware, sin utilizar más de dos núcleos de un procesador convencional, gracias al uso de transferencias DMA. El desarrollo completo de este proyecto ha sido posible, en cuanto al tiempo dedicado a la codificación, gracias al uso de herramientas HLS. La metodología de desarrollo hardware ha supuesto una estrategia realista en términos tecno-económicos ya que tanto el coste de la FPGA utilizada como el tiempo de desarrollo han sido acordes con los objetivos propuestos en este trabajo.

Palabras clave — FPGA, Generación de Flujos, Red Multigigabit

Glosario

AXI-Lite Protocolo de gestión de bus que utiliza la misma mecánica que el AXI-Stream con la diferencia de que soporta escritura en direcciones concretas de memoria. 25, 48

AXI-Stream Protocolo de gestión de bus (de tipo maestro-esclavo) que consta de tres señales principales, el bus de datos, cuyo ancho puede variar, y dos señales de un bit, *valid* y *ready*. Cuando la interfaz esclava está lista para recibir datos activa su bit de *ready* y sabrá que los datos que está recibiendo son válidos cuando la maestra haya activado su bit de *valid*. 14, 17, 22, 24–27, 47, 48

bitstream En este contexto se refiere al fichero binario que configura el Hardware de la FPGA. 19

BRAM Bloque de memoria RAM disponible en una tarjeta FPGA. 11, 34

core IP Módulo hardware privativo configurable mediante parámetros que realiza una función específica. 13, 19, 24, 25, 32, 47

driver Módulo a nivel de núcleo del sistema operativo que se encarga de la comunicación con alguno de los periféricos. 4, 7, 12, 32, 33, 41

huge page Tipo de página de memoria RAM de gran tamaño (1GB). 29

hyperthreading Característica de los procesadores Intel. Si está activada simulará la existencia de más núcleos que los que dispone físicamente. 33, 34

interframe gap Número de ciclos de reloj en los que no se envían datos por la red entre dos paquetes consecutivos. 35

libpcap Tipo de formato de almacenamiento de información de paquetes de red en disco. 12, 31, 32

máscara Combinación de bits que sirve para delimitar el ámbito de una red de ordenadores. 26

multigigabit Referido a una red, aquella que tiene una tasa de transferencia de datos superior a 1 Gbps. 7, 12, 41

template Característica de algunos lenguajes de programación que permite a funciones y clases trabajar con tipos genéricos sin necesidad de implementar un comportamiento específico para ellos. 22

testbed Plataforma en la que se llevarán a cabo pruebas de un determinado sistema. 34

testbench Módulo, normalmente escrito en leguaje HDL, ideado para simular el comportamiento de módulos hardware con la finalidad de comprobar su correcto funcionamiento. 19, 32

traza Fichero en el que se almacenan todos los datos de paquetes capturados de una red. 31, 32

Acrónimos

BAR *Base Address Registers*. 25

CPU *Central Processing Unit*. 2, 18, 36, 37

DMA *Direct Memory Access*. 21, 24, 25, 27–29, 32, 33, 36

DPDK *Data Plane Development Kit*. 13

FIFO *First In First Out*. 25–27

FPGA *Field-Programmable Gate Array*. 1–4, 10–12, 15, 17–19, 21, 25, 28, 33–35, 38, 41, 42, 47, 48

Gbps gigabit por segundo. 2–4, 7, 12, 13, 21, 25, 34, 35, 41

HDL *Hardware Description Languaje*. 3, 4, 13, 19, 22, 25

HLS *High-Level Synthesis*. 18, 21–25, 31

HPCN *High Performance Computing and Networking*. 1–3

I^{2}C *Inter-Integrated Circuit*. 48

IP *Internet Protocol*. 8, 9, 16, 17, 25, 26, 28

IPFIX *Internet Protocol Flow Information Export*. 9, 10, 12

MAC *Media Access Control*. 24, 47

MDIO *Management Data Input/Output*. 47

Mpbs millones de paquetes por segundo. 2, 13, 36, 41

NIC *Network Interface Controller*. 27

PCI Express *Periferal Component Interconnect* de tercera generación. 17–19, 21, 25, 27, 32–34, 47, 48

PCS *Physical Coding Sublayer*. 24, 25, 47

PMA *Physical Medium Attachment*. 24, 25, 47

RAID *Redundant Array of Independent Disks*. 7

RAM *Random Access Memory*. 9, 11, 25, 28, 29

SFP+ *enhanced small form-factor pluggable*. 21, 24, 34

SNMP *Simple Network Management Protocol*. 8, 9

TCP *Transmission Control Protocol*. 8, 9, 17

UDP *User Datagram Protocol*. 8–10, 17

VLAN *Virtual Local Area Network*. 17, 51

XGMII *10 Gigabit Media Independent Interface*. 24, 47

Índice general

Índice de tablas

Índice de figuras

1 Introducción

Este Trabajo de Fin de Grado, realizado en el grupo de investigación *High Performance Computing and Networking* (HPCN), tiene como finalidad la construcción de un sistema de generación de flujos para redes multigigabit utilizando hardware dedicado para un mejor rendimiento. Este sistema se va a desarrollar sobre otro ya existente: *DetectPro*. Para ello, se pretenden realizar las tareas computacionalmente más pesadas en una *Field-Programmable Gate Array (FPGA)* que actuará como coprocesador. Esta estrategia permite que la carga del procesador del ordenador anfitrión disminuya. En los siguientes apartados se describirán los motivos de este proyecto, los objetivos del trabajo y los retos que éstos van a suponer. Por último se explicará brevemente la estructura del documento.

1.1. Motivación del proyecto

Hoy en día casi todo el mundo dispone de uno o varios dispositivos electrónicos (ordenadores, teléfonos móviles, tabletas, televisiones, videoconsolas, *smartwatches*...), y esta tendencia no va sino en aumento. Tal es su crecimiento que ya se habla de *Internet of Things (IoT)* o el Internet de las Cosas, se aventura que hasta los dispositivos más cotidianos estarán conectados a la red. Con este auge en el uso de la tecnología a lo largo de las últimas décadas y el consiguiente incremento del uso de Internet, la implementación de herramientas que analicen enlaces de red de alta velocidad de forma eficiente se ha convertido en una tarea de vital importancia. Las empresas de Internet necesitan estas herramientas para poder mejorar sus servicios o detectar anomalías.

En un principio, se podría pensar que la captura de paquetes de red que atraviesan un enlace para su posterior análisis es suficiente para caracterizar el uso de éste. Debido a las altas tasas de transferencia que se dan en los enlaces multigigabit, es necesario procesar

los datos de esos paquetes a una velocidad determinada. De esta forma, la información que pasa por el enlace no necesita ser almacenada.

Para intentar caracterizar el uso de un enlace de red, el análisis individual de los paquetes que lo atraviesan es inviable, debido a que se pueden alcanzar tasas de hasta 14 millones de paquetes por segundo (Mpbs). Una solución a este problema consistiría en la agrupación de paquetes que compartan ciertas características.

Un flujo de red es un conjunto de paquetes de Internet que representan una comunicación unidireccional entre dos extremos. La información típica que se puede obtener de un flujo es: las direcciones del emisor y el receptor, el tipo de aplicación que se está utilizando, si la información está cifrada o no, si los datos se envían en bloques grandes o pequeños o el tiempo que ha durado la comunicación. Por la información que se obtiene de ella, la generación de flujos ofrece una visión bastante completa del tipo de tráfico que está atravesando un enlace de la red. La existencia de herramientas que sean capaces de abarcar de forma eficiente la mayor cantidad posible de tráfico que atraviesa un enlace es fundamental para la gestión de la propia red.

El desarrollo de estos programas no es una tarea simple, como se expondrá en el análisis del estado del arte. Existen diversas soluciones en el mercado tanto de código libre como privativas, cada una con sus ventajas e inconvenientes. En general, la carencia de estas aplicaciones radica en que no son son capaces de analizar a tasa máxima de red, o, en el mejor de los casos, que solo funcionan cuando el tamaño de los paquetes de red es medio. En los casos extremos en los que los paquetes son de tamaño pequeño o mínimo se produce una pérdida de paquetes por las limitaciones del hardware actualmente utilizado en el procesamiento de los flujos. Estos casos, aunque no sean comunes, son muy importantes a la hora de detectar posibles ataques, pues éstos se suelen realizar mediante el envío de paquetes de tamaño pequeño. Existen, por otro lado, soluciones más eficientes, que conllevan, sin embargo, un elevado coste de adquisición.

DetectPro es un programa de generación de flujos a 10 gigabit por segundo (Gbps), desarrollado por el HPCN, que dispone de muchas funcionalidades implementadas y optimizadas. Adicionalmente a la generación de flujos, dispone de un sistema de alarmas que detecta anomalías en el tráfico tales como ataques o fallos en enlaces cercanos y un sistema de filtrado por el cual se clasifican sólo aquellos que pertenecen subredes concretas que puedan interesar al usuario.

No obstante, el programa tiene algunos inconvenientes, que se introducen ahora y se detallarán más adelante. No es capaz de procesar todo el tráfico cuando las condiciones de un enlace son adversas, como cuando se realizan ataques informáticos donde se utilizan paquetes de red de tamaño pequeño y, además, requiere la reserva exclusiva de 3 núcleos de un procesador para alcanzar su rendimiento máximo. Estos requerimientos de hardware son muy exigentes y las empresas no siempre están dispuestas a ceder el uso completo de sus ordenadores.

En este trabajo de investigación se propone el uso de una FPGA haciendo las funciones de coprocesador. De esta forma se reducirá el uso de *Central Processing Unit* (CPU) de esta clase de programas.

La programación de hardware, en este caso, una FPGA es una tarea ardua que normalmente conlleva tiempos de desarrollo muy largos. Se han estudiado para este proyecto diferentes herramientas que acortan el proceso mediante el uso de lenguajes de más alto nivel. A partir de funciones escritas en lenguajes como C++, estas herramientas generan módulos hardware en diferentes *Hardware Description Languaje*s (HDLs).

1.2. Objetivos del proyecto

El principal propósito de este proyecto es conseguir un sistema que genere flujos con la ayuda de un coprocesador, que en este caso será una FPGA, y que nos ofrezca toda la información necesaria para obtener las características de los paquetes que atraviesan un enlace de red. Para lograrlo se plantean los siguientes objetivos:

En cuanto al hardware:

- Diseñar un módulo hardware implementable en una FPGA que sea capaz de analizar, a tasa de línea de una red multigigabit ethernet (10 Gbps), el tráfico íntegramente y de extraer la información necesaria de cada paquete individual que permita la generación de flujos válidos.
- Realizar una clasificación de los flujos en función de las direcciones del emisor y receptor del tráfico. Para ello se desarrollará en el coprocesador un sistema que sea capaz de detectar la pertenencia de las direcciones IP de los paquetes a las subredes monitorizadas. El tiempo en el que se hará esa clasificación deberá ser suficientemente corto como para que no comprometa el rendimiento del conjunto del proyecto.

En cuanto al software:

- Encontrar un mecanismo de comunicación de la información desde la FPGA al ordenador anfitrión que permita el envío de datos a 10 Gbps (o más) sin que ello ralentice el procesamiento de los datos en software ni requiera más de 3 núcleos de un procesador convencional.
- Utilizar las funcionalidades ya desarrolladas por el HPCN del programa *DetectPro* integrando en él un mecanismo de comunicación con la FPGA.

1.3. Retos

Durante la elaboración de este proyecto, para conseguir los objetivos mencionados, han surgido diferentes retos que superar tanto en el aspecto del software como en el del hardware. Los retos más difíciles han sido los relacionados con los requisitos de rendimiento del sistema desarrollado. Algunos de los retos encontrados han sido:

- En primer lugar la elección de la herramienta apropiada. Debido al limitado número de horas de las que se dispone para realizar el Trabajo de Fin de Grado, el tiempo disponible para el desarrollo del sistema no es muy extenso. Por ello se ha puesto especial interés en la elección del entorno de desarrollo y lenguaje de programación.

- Explorar todas sus características y conocerla de forma que se obtengan de ella los mejores resultados, una vez elegida la herramienta.

- Es necesario, por otra parte, que la herramienta seleccionada sea capaz interpretar el código de más alto nivel y traducir su comportamiento al correspondiente en lenguaje HDL. Las tareas de este hardware deben realizarse en un tiempo acorde al rendimiento deseado en todo el proyecto, es decir, en el mismo tiempo en el que se reciben los paquetes.

- Una vez desarrollado el hardware, será necesario optimizarlo para que funcione a la tasa deseada (10 Gbps). Además, al estar sincronizado con software, tanto la transferencia del coprocesador al ordenador anfitrión como el procesamiento en sí deberán respetar esta tasa.

- Este proyecto utilizará código previamente desarrollado. El reto consistirá en integrar la funcionalidad existente con el *driver* encargado de la comunicación entre hardware y software.

1.4. Estructura del documento

Este documento está dividido en cinco secciones.

En el capítulo 2 se realizará un estudio del estado del arte del análisis de redes, poniendo especial atención en la generación de flujos. Se analizarán los diferentes formatos de flujos así como las soluciones ya existentes en el mercado, agrupadas en tres formatos distintos según la infraestructura utilizada. Por último, dado que el desarrollo hardware no es una tarea sencilla, también se estudiarán posibles entornos de programación para el proyecto.

En el capítulo 3 se definirá el proyecto, su alcance y sus funcionalidades, y se detallarán los requisitos de los diferentes módulos que lo forman. También se enunciarán los entornos de programación elegidos y la motivación del uso de éstos. Para concluir este capítulo, se describirá el modelo de ciclo de vida del proyecto que se llevará a cabo.

En el capítulo 4 se describirá la arquitectura general del sistema, incluyendo los subsistemas hardware y software. A continuación se realizará una breve descripción de la forma de uso del entorno de programación elegido. En el tercer punto de este capítulo se presentará la arquitectura del módulo hardware desarrollado. Posteriormente se explicará el funcionamiento del software reutilizado. Para finalizar el capítulo, se hará una descripción del modo de comunicación entre el programa software y la FPGA.

En el capítulo 5 se especificarán las formas de validación del hardware (diferentes tipos de simulaciones), el entorno en el que se ha simulado un enlace real y los resultados de rendimiento obtenidos. Se realizará, así mismo, una comparativa del sistema desarrollado híbrido (hardware-software) respecto del original basado únicamente en software.

En el capítulo 6 se expondrán las conclusiones de este Trabajo de Fin de Grado y las posibles líneas de trabajo futuro.

2

Estado del Arte

En esta sección se va a realizar un estudio del estado del arte de los principales puntos de este Trabajo de Fin de Grado. Por un lado se desarrollará el análisis de redes, en concreto en redes multigigabit ethernet, y, por otro lado, las principales herramientas de desarrollo de hardware existentes en el mercado.

2.1. Introducción al análisis de redes

Los operadores de Internet necesitan hacer análisis rutinarios del tráfico con el fin de conocer el uso real de sus enlaces, encontrar posibles fallos en los enlaces vecinos o detectar ataques maliciosos. Típicamente, estos análisis se realizan mediante técnicas de monitorización pasiva, para que el impacto sobre la red sea nulo.

En las primeras décadas de Internet, se llevaba a cabo la captura y volcado de paquetes a disco con herramientas como tcpdump. La idea era almacenar todos los paquetes que atravesaban un enlace de red con un formato estándar que posteriormente se pudiera leer y analizar. Esta tarea se podía llevar a cabo mediante una tarjeta de red convencional que transmita los datos a un software encargado de escribir los paquetes en el disco.

Este sencillo método es suficiente en enlaces lentos (hasta 1 Gbps) pero se queda corto para enlaces multigigabit (10 Gbps) por varias razones. En primer lugar, las gestiones que debe realizar el sistema operativo al recibir paquetes ralentiza el proceso. Para resolver este problema, diversas empresas y universidades han diseñado *drivers* específicos que optimizan la transferencia de paquetes consiguiendo soluciones buenas únicamente basadas software. En segundo lugar, por la velocidad de transferencia de los datos: La tasa de escritura en discos convencionales no es suficiente; se requeriría un *Redundant Array*

of Independent Disks (RAID) para conseguir un rendimiento aceptable, lo que supone un incremento considerable del coste. Una de las soluciones a este segundo problema es el almacenamiento parcial de la información relevante transmitida por la red. Existen varias técnicas que se describen a continuación:

Generación de flujos mediante puertos.

Consiste en la reconstrucción de las comunicaciones unidireccionales entre emisores y receptores mediante la información obtenida a partir de los campos de las cabeceras de los paquetes que se han enviado. Para ello se suelen estudiar sólo los paquetes que implementan *Internet Protocol* (IP). Los campos que se utilizan no están estandarizados y pueden variar de unas aplicaciones a otras. Un ejemplo de quíntupla de caracterización de flujos muy utilizada es: dirección IP origen, dirección IP de destino, número de protocolo de nivel de transporte, y, en caso de que se utilicen *Transmission Control Protocol* (TCP) o *User Datagram Protocol* (UDP), número puerto origen y número puerto destino.

Muestreo estadístico.

Consiste en la captura de paquetes completos durante intervalos cortos de tiempo. La desventaja de este método es que, para que realmente sea representativo, el enlace se tiene que encontrar en un estado estable, es decir, el tráfico que fluye a través de él no debería sufrir grandes oscilaciones a lo largo del tiempo. Además, podría no ser eficaz a la hora de detectar ataques de red ya que éstos se pueden producir en de tiempos inferiores al intervalo de muestreo. A pesar de esto, existen métodos estadísticos bastante eficaces a la hora de clasificar tráfico, como se muestra en [1], donde utilizan técnicas de análisis bayesianas.

Protocolos de gestión de red.

Son protocolos de la capa de aplicación que facilitan el intercambio de información de administración entre dispositivos de red como por ejemplo *Simple Network Management Protocol* (SNMP). Aportan ciertas estadísticas sobre la red pero no permiten el cálculo de las matrices de tráfico. Realizan un conteo de paquetes pero no almacena información a cerca de las conexiones.

Este trabajo está centrado en la primera opción.

Dando una definición más formal, un flujo es una secuencia de paquetes unidireccional con ciertas propiedades comunes que discurren a través de un dispositivo de red. Primero se van a estudiar los distintos formatos de exportación de flujos.

2.2. Formatos de exportación de flujos

En primer lugar hablaremos de NetFlow [2]. Es un protocolo desarrollado por Cisco Systems [3] para recolectar información sobre tráfico IP. El concepto de flujo, al no ser estándar, no tiene definidas unas características fijas que lo determinen. En el caso de NetFlow versión 9 [4], se define un flujo como una secuencia unidireccional de paquetes que comparten 7 características: Interfaz de ingreso (índice del protocolo SNMP); dirección IP origen; dirección IP destino; protocolo de transporte; puerto origen para tráfico UDP o TCP (0 en caso de otros protocolos); puerto destino para tráfico TCP o UDP, tipo y código para paquetes ICMP (0 en caso de otros protocolos); y Type o Service de la cabecera IP.

Un flujo se considera inactivo si no se observan paquetes que le pertenezcan en un tiempo prefijado. Se exportará si cumple alguna de las siguientes condiciones:

- El exportador detecta el final del flujo. Por ejemplo si la bandera de FIN o RST se detecta en el caso de conexiones TCP.
- El flujo ha estado inactivo durante un cierto periodo de tiempo.
- Para flujos de larga duración, el exportador comunica la existencia del flujo de forma regular (tras un cierto tiempo).
- Cuando el exportador tiene problemas internos, un flujo se podría ver obligado a expirar prematuramente; por ejemplo por falta de memoria.

Internet Protocol Flow Information Export (IPFIX) [5] es un protocolo abierto, que se creó como respuesta a la necesidad de un estándar universal de exportación de flujos desde routers, sondas y otros dispositivos de análisis. El protocolo IPFIX es un superconjunto del protocolo NetFlow v9, después de haber evolucionado a través de un proceso de adición de funcionalidades para cumplir con los requisitos y las necesidades de las partes interesadas dentro del grupo de trabajo de IPFIX.

En la Tabla 2.1 se puede ver algunos ejemplos de registros de flujos exportados en formato IPFIX [6].

2.3. Soluciones disponibles

La construcción de flujos a alta velocidad no es un problema fácil de resolver. Se requieren tanto velocidad de cálculo como una cantidad memoria *Random Access Memory* (RAM) importante para poder mantener la información de todos posibles flujos concurrentes durante el procesamiento. No obstante, es de interés para los proveedores de Internet contar con esta información, y diferentes fabricantes e investigadores han implementado algunas soluciones. A continuación se detallan algunas opciones existentes en el mercado que atacan el problema de maneras diferentes.

Tabla 2.1: Ejemplos de tuplas de IPFIX

Version	Export time	SrcIP	DstIP	ProtocolId
10	2007-10-09 00:01:57 UTC	192.0.2.2	192.0.2.3	6
10	2007-10-09 00:01:58 UTC	192.168.2.2	192.0.1.2	6
10	2007-10-09 00:02:03 UTC	192.0.36.2	192.0.6.12	6

Octets	FlowStartSeconds	SrcPort	DstPort	TotalPackets
18000	2007-10-09 00:01:13 UTC	80	32770	65
19010	2007-10-09 00:01:13 UTC	223	35471	76
20087	2007-10-09 00:00:02 UTC	55684	84749	65

2.3.1. Generación de flujos vía routers y switches

Estos dispositivos suelen utilizar protocolos como NetFlow o IPFIX. Los routers y switches de una red recogen estadísticas en todas sus interfaces y envían los datos a un servidor central donde serán almacenados y procesados.

Los registros de NetFlow se exportan tradicionalmente mediante tráfico UDP. Por razones de eficiencia, el router no guarda los flujos una vez que han sido exportados, de forma que si un paquete de NetFlow se pierde por la congestión de la red o problemas del medio, todos los datos se pierden. Otra desventaja de este sistema es que cuando los dispositivos que monitorizan reciben una gran cantidad de paquetes dedican todos sus recursos al correcto reenvío de los datos, por lo que dejan de generar estadísticas.

2.3.2. Generación de flujos vía hardware

NetFPGA [7] es un proyecto de código software y hardware libre iniciado por Stanford University y Xilinx [8]. Consiste en una plataforma reconfigurable basada en una FPGA Virtex. Una de las primeras implementaciones se hizo en una NetFPGA-1G (Virtex-2) [9] desarrollado por Martin Žádník. Esta solución exportaba flujos mediante el protocolo NetFlow v5 y era capaz de medir de las cuatro interfaces de 1 Gbps de la tarjeta a tasa de línea, teniendo en cuenta sólo tráfico IPv4. El inconveniente de esta solución es que la memoria sólo permitía mantener 4.000 flujos concurrentes.

Otra de las soluciones iniciales fue la de Yusuf [10] en 2.008. Propuso una arquitectura para análisis de flujos que utilizaba un dispositivo basado en Virtex-2 capaz de almacenar hasta 65.536 flujos concurrentes a una tasa máxima de 3 Mpps.

NetFPGA-10G es la evolución del proyecto NetFPGA comentado anteriormente. Para él se construyó una nueva placa integra una FPGA Virtex-5 dotada de cuatro interfaces Ethernet de 10 Gbps. Hace dos años, se diseñó para esta plataforma una aplicación que reconstruía y exportaba los flujos según el protocolo Netflow [11]. El sistema sigue el esquema de la Figura 2.1, obtenida del mismo artículo.

En paralelo se realizan dos procesos, por un lado la creación y actualización de flujos

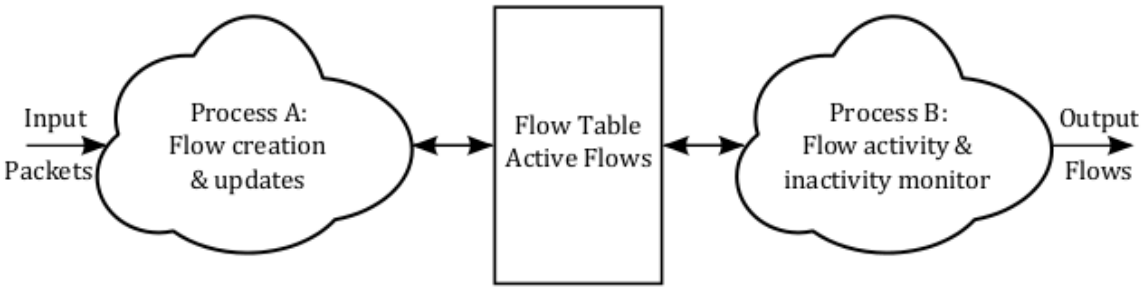

Figura 2.1: Proceso de generación de flujos [11].

en la tabla y en paralelo por otro lado se exportan los flujos que han expirado.

Para ello el sistema contaba con varios módulos que se presentan en la Figura 2.2. Los módulos que componen el sistema son:

- *Packet Parser*, que extrae la información de los paquetes
- *Hashing Module*, que calcula la función *hash* que se utilizará para insertar o actualizar los flujos en la tabla.
- *Export module*, que comprueba qué flujos han expirado de la tabla con unas condiciones de *timeout*.

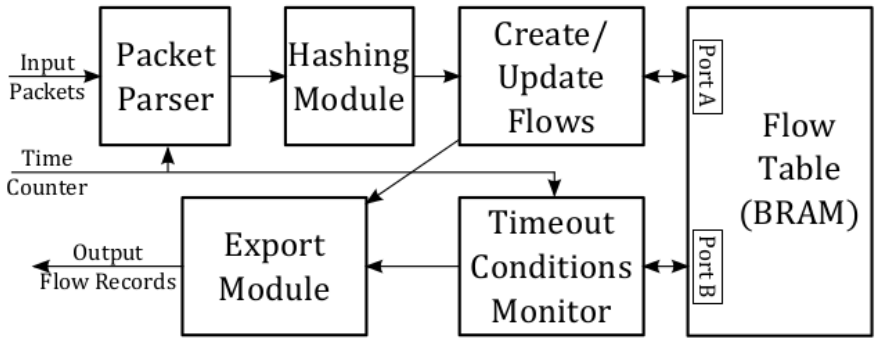

Figura 2.2: Esquema de módulos de NF_QDR [11].

La limitación de esta aplicación está en el número de flujos concurrentes ya que, por el número de BRAMs disponibles en la tarjeta, podía ser, como máximo, 786432.

Las Block RAM son un tipo de memoria disponible en las FPGA con dos puertos que tienen una capacidad del orden de kilobits. La latencia de acceso a estas memorias suele ser de 2 ciclos de reloj, que es una latencia muy baja. Por tanto, son muy útiles

en este tipo de aplicaciones. No obstante, al ir integradas en el chip programable, no son muy extensas, es decir, no tienen mucha capacidad. Por ello, son el factor limitante de la mayoría de las soluciones hardware.

Existen sistemas de altas prestaciones como, por ejemplo, *FlowMon* desarrollado por INVEA-TECH [12]. Este sistema fue analizado por M. Žádník [13], mencionado anteriormente. En su informe se describe que las sondas basadas en FPGA son capaces de monitorizar enlaces de 10 Gbps a tasa de línea. Se analizan dos versiones de FlowMon: una capaz de monitorizar a tasa de línea pero con escasa flexibilidad y otra flexible y lo suficientemente potente como para monitorizar enlaces de 10 Gbps en condiciones normales. Esta solución supone para las empresas un elevado coste económico.

2.3.3. Generación de flujos vía software

Existe una amplia gama de implementaciones software en función de las prestaciones deseadas. Algunas de las herramientas más populares *open source* disponibles son fProbe, nProbe y softlowd.

fProbe [14] captura tráfico y genera los flujos de Netflow asociados. Está basada en *libpcap* y funciona a 1 Gbps. *nProbe* [15] fue desarrollada por Cisco. Exporta flujos tanto hacia el ordenador como a la red y en diferentes formatos. El funcionamiento se puede ver en la Figura 2.3. Este programa recibe los flujos generados por una sonda en el ordenador y los almacena en una base de datos del ordenador si se desea. A su vez, estos datos se exportan a través de la red con NetFlow o IPFIX hacia un colector más grande.

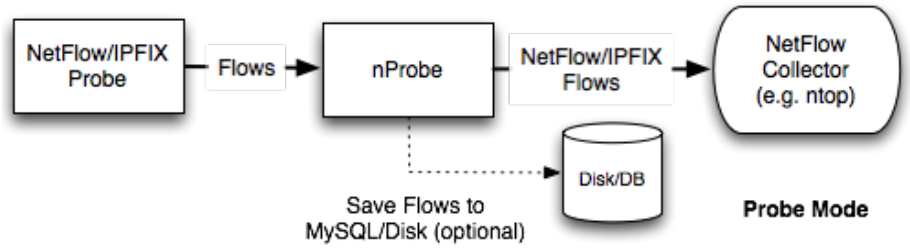

Figura 2.3: Funcionamiento de nProbe [15].

nProbe está pensado para funcionar a 1 Gbps, pero también es posible su ejecución en una red de alta velocidad. En la actualidad, el *commodity hardware* no puede conseguir el análisis a 10 Gbps como se comentó anteriormente, pero es posible alcanzar esta tasa utilizando *drivers* específicos.

Por último, softlowd analiza flujos de tráfico obtenidos mediante la escucha de una interfaz de red o un fichero en el que se hayan guardado paquetes capturados con herramientas como tcpdump. Al igual que los dos programas anteriores, este programa no soporta la tasa de redes multigigabit.

En cuanto a soluciones de más altas prestaciones, ffProbe es una herramienta que utiliza técnicas avanzadas de programación paralela. En el artículo escrito por Danelutto

[16], se compara ffProbe con nProbe y softflowd. Consiguió una tasa cercana a los 10 Mpbs para ffProbe y unos resultados peores para las demás herramientas.

También podemos encontrar *DetectPro* [17], que utiliza un driver específico, HPCAP [18]. Éste programa genera flujos a 10 Gbps en tiempo real, almacena parte del *payload* de los paquetes, tiene alarmas que detectan flujos sospechosos y analiza las subredes a las que pertenecen los paquetes. Procesa a tasa de red de un enlace en condiciones normales pero pierde paquetes cuando de media tienen tamaño pequeño.

Por último, Intel *Data Plane Development Kit* (DPDK) [19] que es un conjunto de librerías y drivers pensado para el procesado de paquetes. Fue creado con el propósito de obtener el máximo rendimiento por tarjeta (10 Gbps), realizar una implementación sencilla y rápida, explotar al máximo la arquitectura multicore y crear programas flexibles y escalables sin trabajo adicional. Está desarrollado para procesadores Intel y en Linux, pero la licencia es propietaria. El inconveniente es que requiere el uso de tarjetas de red específicas. Además, está orientado a captura de *batches* de paquetes, no a generación de flujos y supone una limitación a nivel de software.

2.4. Herramientas y lenguajes de desarrollo hardware

A la hora de acelerar un sistema con un coprocesador hardware nos encontramos con algunos problemas en torno a los lenguajes HDL como, por ejemplo, la dificultad a la hora de de programar y de *debuguear* o la falta de abstracción de éstos. Estos inconvenientes han propiciado el desarrollo de diferentes herramientas que facilitan la programación de hardware mediante el uso de lenguajes de más alto nivel. En [20] se realiza un estudio de algunas de las herramientas, cuyas características se resumen a continuación.

- **SystemVerilog.** Es un lenguaje de tipo HDL en el que los diseños se describen como redes de flujo de datos con reglas atómicas. El uso de este lenguaje se ha descartado por la dificultad en su curva de aprendizaje.

- **OpenCL.** Es un lenguaje utilizado para computación paralela. Empresas como Altera y Xilinx han desarrollado programas que sintetizan hardware a partir de funciones codificadas en este lenguaje. El inconveniente de esta técnica es su joven desarrollo; sólo se puede acceder a las herramientas mediante *early access* y no tienen soporte completo.

- **LegUp.** Es una herramienta *open source* que genera código HDL a partir de código escrito en C o C++. Esto facilita el aprendizaje ya que la mayoría de los programadores conocen estos lenguajes previamente. Su principal inconveniente está en que no ofrece el rendimiento requerido por el proyecto.

- **Vivado HLS.** Es una herramienta de Xilinx que permite la generación de cores IP a partir de funciones descritas en C++. Esta herramienta no aparece en el artículo pero ha sido probada y utilizada en el HPCN previamente. El funcionamiento de

estos cores está optimizado y facilita la gestión de interfaces de buses de tipo AXI-Stream.

Tras el análisis de estas herramientas, y teniendo en cuenta los objetivos de este proyecto, se ha considerado que el entorno que más ventajas ofrece para el desarrollo es Vivado HLS, por su rápido aprendizaje, el nivel de optimización y las facilidades que ofrece a la hora de comprobar el comportamiento de los módulos generados, entre otras características.

3

Análisis y diseño

En este capítulo se describe de forma más detallada el Trabajo de Fin de Grado y su alcance. A continuación se explican los requisitos de los diferentes módulos del proyecto y se exponen las herramientas que se van a utilizar para el desarrollo del mismo. Por último se explica el modelo de ciclo de vida utilizado.

3.1. Definición del proyecto

El proyecto consistirá en construir un sistema de generación de flujos acelerado mediante una tarjeta FPGA. Para ello se implementarán en hardware la captura de paquetes de la red, los cálculos básicos y el filtrado. Se pretende idear un mecanismo de transmisión que respete la tasa de red para comunicar los datos de la FPGA al ordenador anfitrión. Para las tareas que se desean realizar en software se reutilizará un programa ya probado, *DetectPro*, realizando las modificaciones acorde a las funcionalidades que se realizan en el coprocesador.

3.2. Alcance

El proyecto se compone de dos partes claramente diferenciadas. Por un lado, se diseñará una aplicación hardware implementada en una tarjeta FPGA que será la encargada del cálculo. Por otro lado, se desarrollará una funcionalidad básica de recepción software de los datos enviados desde la tarjeta y se modificará el programa *DetectPro* para que funcione de acuerdo al hardware dedicado.

Como se estudió en el estado del arte (sección 2.3.2), los diseños implementados puramente en hardware tienen limitaciones a la hora de mantener la información necesaria para todos los flujos concurrentes esperables de una red multigigabit. Por ello, las tareas que se realizarán en hardware serán las de la extracción de los valores de los campos de las cabeceras de los paquetes relevantes para el cálculo de flujos; la clasificación de subredes; y el marcado de tiempo.

En cuanto a la parte software del proyecto, es importante destacar que no se pretende modificar ni añadir funcionalidades a *DetectPro*. La tarea consistirá en sustituir los módulos de comunicación con la tarjeta de red y funciones de parseo de paquetes por una nueva funcionalidad que se encargará de recibir e interpretar los datos del hardware.

3.3. Catálogo inicial de requisitos

A continuación se detallan los requisitos funcionales y no funcionales tanto del subsistema hardware como el software.

3.3.1. Requisitos del subsistema hardware

Requisitos funcionales

RFH 1. El subsistema deberá disponer de una señal para reiniciar todo el diseño hardware.

RFH 2. El subsistema deberá transmitir al anfitrión únicamente los paquetes que utilicen protocolo IP.

RFH 3. El subsistema dispondrá de un mecanismo de configuración con el que se especificarán las subredes que se van a monitorizar.

- **RFH 3.1.** Las subredes monitorizadas y sus identificadores se leerán de un fichero de configuración.

RFH 4. El subsistema deberá descartar los paquetes que no pertenezcan a las subredes monitorizadas.

RFH 5. El subsistema deberá extraer información de los paquetes que cumplan los requisitos funcionales 2 y 4.

- **RFH 5.1.** El subsistema deberá extraer de la cabecera de nivel de enlace del paquete las direcciones físicas de origen y destino.
- **RFH 5.2.** El subsistema deberá extraer de la cabecera de nivel de red las direcciones de origen y destino, el protocolo de transporte, el número identificador y la bandera de paquete fragmentado.

RFH 5.3. El subsistema deberá extraer de la cabecera de nivel de transporte en caso de que el paquete utilice protocolo UDP los números de puerto origen y destino.

RFH 5.4. El subsistema deberá extraer de la cabecera de nivel de transporte en caso de que el paquete utilice protocolo TCP los números de puerto origen y destino así como las principales banderas TCP.

RFH 6. El subsistema deberá mostrar a qué subredes monitorizadas pertenece la dirección IP de origen de un paquete.

RFH 7. El subsistema deberá mostrar a qué subredes monitorizadas pertenece la dirección IP de destino de un paquete.

RFH 8. El subsistema deberá soportar la interpretación del protocolo *Virtual Local Area Network* (VLAN) de forma anidada.

RFH 9. El subsistema deberá realizar un marcado temporal de los paquetes.

RFH 10. El subsistema ofrecerá un mecanismo para conocer la cantidad de paquetes que se han recibido y procesado de la red.

Requisitos no funcionales

RNFH 1. El subsistema hardware deberá comunicarse con el ordenador mediante el bus *Periferal Component Interconnect* de tercera generación (PCI Express).

RNFH 2. El subsistema debe procesar los paquetes a tasa máxima de red, 10 Gbps y 14 Mpps.

RNFH 3. Se utilizarán interfaces AXI-Stream para la comunicación entre todos los módulos.

RNFH 4. El subsistema debe ser implementable en una FPGA de Xilinx Kintex-7 KC705 xc7k325tffg900-2 [21].

RNFH 5. El subsistema se conectará a la red multigigabit mediante un conector SFP+.

3.3.2. Requisitos del subsistema software

Requisitos funcionales

RFS 1. El subsistema será capaz de construir flujos a partir de la información de paquetes independientes.

RFS 2. La información de los flujos generados deberá almacenarse en disco para su posterior análisis.

RFS 3. Las condiciones de expiración de un flujo serán las mismas que las definidas en [4], exceptuando la condición número 4.

RFS 4. El subsistema dispondrá de una generación de estadísticas para unas subredes determinadas.

- **RFS 4.1.** En las estadísticas se contabilizarán: el número de flujos por segundo, la tasa en bytes por segundo y en paquetes por segundo.
- **RFS 4.2.** Las subredes monitorizadas y sus identificadores se leerán de un fichero de configuración.
- **RFS 4.3.** Las estadísticas se almacenarán en ficheros independientes para cada subred monitorizada.

RFS 5. El subsistema dispondrá de funciones para iniciar y terminar la comunicación con la FPGA.

RFS 6. El subsistema traducirá los datos recibidos de la FPGA al tipo de estructura que utilice.

RFS 7. El subsistema convertirá el *timestamp* calculado en hardware a la escala del ordenador anfitrión.

Requisitos no funcionales

RNFS 1. La aplicación debe funcionar en Linux con un kernel de 2.6.32.

RNFS 2. El subsistema traducirá los datos recibidos respetando la velocidad de envío del PCI Express.

RNFS 3. La espera de nuevos datos de la tarjeta no debe consumir tiempo computacional de la CPU.

RNFS 4. El subsistema total, no ocupará más de 2 núcleos de la CPU al 100 %.

3.4. Entorno de desarrollo y lenguajes de programación

En esta sección se describen las tecnologías actuales de desarrollo que se van a utilizar para la implementación de los dos subsistemas del proyecto.

3.4.1. Entornos de desarrollo del subsistema hardware

Analizando los posibles entornos de programación, se ha decidido que se desarrollará la funcionalidad principal de la aplicación hardware (el parseo y filtrado de los paquetes) con Vivado *High-Level Synthesis* (HLS) en su versión 2014.2 de Xilinx [22]. Este

programa ofrece unas ventajas significativas frente a una implementación en VHDL o verilog directa, porque facilita tareas como la gestión de interfaces de comunicación y la implementación de máquinas de estados. También ofrece mejoras en cuanto a la simulación del comportamiento hardware, porque permite la programación de los *testbench* en C++, facilitando el proceso de desarrollo. Como característica adicional interesante, se puede apuntar que este entorno permite la exportación del diseño en forma de core IP. Gracias a esto, es muy sencillo integrar con los módulos que se desarrollarán en HDL. Además, tiene la particularidad de acelerar el proceso de síntesis, puesto que sólo debe realizarse una vez: al generarse, y no en cada implementación como ocurriría si se exportara en lenguaje HDL. La programación se realizará en C++, ya que es el lenguaje requerido por el entorno.

La arquitectura de la comunicación de la FPGA con el ordenador, así como los módulos comparadores se realizarán en verilog, puesto que requiere una mejor optimización. Se utilizará el programa Vivado versión 2014.2 [23] para la síntesis, implementación y generación del *bitstream* del diseño completo. Se ha elegido esta herramienta ya que incluye métodos de simulación de hardware mediante *testbench*. También permite *debuguear* el diseño una vez programado el *bitstream* en la FPGA mediante un sistema para la monitorización de los registros o buses.

3.4.2. Entorno de desarrollo del subsistema software

En cuanto al desarrollo software, se llevará a cabo en Scientific Linux 6.2 en lenguaje C las dos tareas. Por un lado la adaptación de *DetectPro* y por otro lado la interfaz que se desarrollará sobre el driver para la comunicación por PCI Express. Para la detección y corrección de errores en la programación se utilizará GDB [24].

3.5. Modelo de ciclo de vida

Se ha dividido el desarrollo de todo el proyecto en diferentes fases. En primer lugar se llevará a cabo el desarrollo del subsistema hardware. Una vez este diseño haya sido completado y se hayan verificado y validado sus requisitos, se procederá a la adaptación de los datos recibidos a las estructuras del programa original. Por último, se adaptará el sistema de captura integrándolo con *DetectPro*.

3.5.1. Subsistema hardware

Se va a utilizar un modelo incremental iterativo, ya que pueden aparecer nuevos requisitos durante el desarrollo del trabajo. Este sistema permite comprobar incrementalmente que los requisitos no funcionales hardware se satisfacen. A continuación se detalla la actividad aproximada que se realizará en cada iteración:

Iteración 1. El diseño comunicará los datos recibidos por la red sin realizar ningún tipo de operación.

Iteración 2. El diseño cumplirá RFH 1 y RFH 5.

Iteración 3. El diseño cumplirá los requisitos del prototipo anterior y RFH 2, RFH 3, RFH 4, RFH 6, RFH 7 y RFH 8.

Iteración 4. El diseño cumplirá todos los RFH y RNFH.

3.5.2. Subsistema software

El desarrollo de este subsistema se llevará a cabo con un modelo en cascada iterativo, que permite volver a fases anteriores si se produjera un cambio en los objetivos o los requisitos. Esta característica es muy importante debido a que se trata de un proyecto de investigación.

4

Desarrollo

En este capítulo se va a describir la arquitectura del sistema completo. En primer lugar se ofrecerá una visión general del sistema, incluyendo hardware y software. A continuación se mostrará un sencillo ejemplo de uso del entorno de programación escogido (Vivado HLS). En la siguiente sección, se expondrá la arquitectura del diseño hardware tal y como se ha implementado en la FPGA. En el punto siguiente se detallará el funcionamiento de los módulos hardware desarrollados en este trabajo. En la sección 5, se describirá el programa software en el que se ha basado este sistema (*DetectPro*), y se indicarán los cambios que se han realizado. Por último se mostrará el algoritmo de comunicación utilizado para la transferencia y adaptación de los datos entre los diferentes subsistemas del proyecto.

4.1. Arquitectura general del sistema

El sistema está formado por un ordenador anfitrión y una FPGA conectada mediante PCI Express. En el ordenador se llevará a cabo la ejecución del programa principal y la FPGA actuará de coprocesador. En la Figura 4.1 se muestra un esbozo aproximado del sistema completo.

El flujo de trabajo del sistema es el siguiente: Los datos llegan a través de un conector *enhanced small form-factor pluggable* (SFP+) que soporta 10 Gbps. En la FPGA se extrae la información relevante. Una vez procesados los datos, se envían a través del bus PCI Express al anfitrión mediante transferencias *Direct Memory Access* (DMA). Los datos almacenados en memoria RAM se encuentran en páginas de tamaño más grande de lo común (1 GByte), que son las llamadas *Huge Pages* en la Figura 4.1. El subsistema sofware accederá estas páginas especiales y convertirá los datos leídos al tipo de estructura de datos

que utiliza. A partir de estas estructuras, el programa generará o actualizará los flujos que ha ido almacenando en su memoria de flujos. A medida los flujos vayan expirando se exportarán al disco duro.

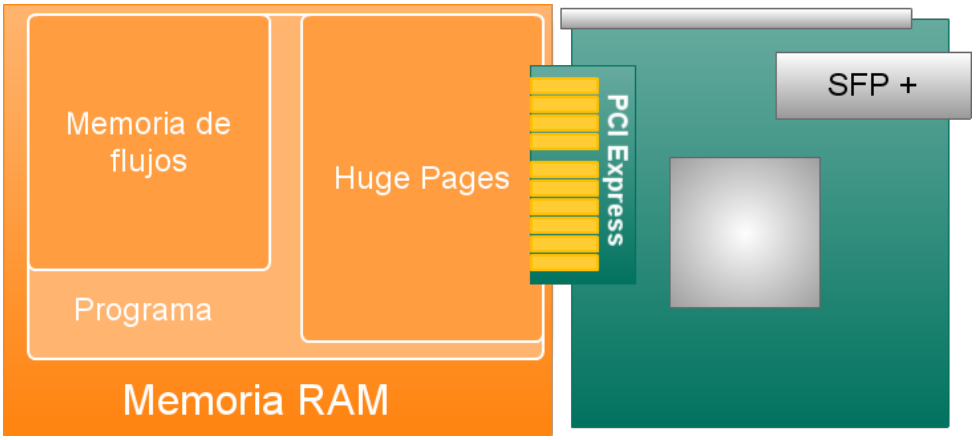

Figura 4.1: Arquitectura general del sistema.

4.2. Programación con Vivado HLS

Es importante destacar las facilidades que han supuesto el uso de un lenguaje HLS en lugar de un lenguaje HDL, sobre todo a la hora de la gestión de interfaces AXI-Stream. Para describir un módulo hardware con esta herramienta se utiliza una función típica escrita en C++ donde los parámetros serán las entradas y salidas del módulo. En nuestro caso, para la escritura y lectura de estas interfaces se ha utilizado la clase hls::stream<T>, ya que tiene definidas funciones de lectura, escritura y acceso a datos. Para indicarle al compilador el tipo gestión de buses que se desea se escriben directivas tipo *pragma*, como se puede ver en la Figura 4.2.

```
1  void data_manager_v2(stream<mac_user_interface_type> &input, stream<output_interface_type>
       ↪ &output, stream<mac_user_interface_type> &comp_input,
       ↪ stream<mac_user_interface_type> &timestamp) {
2
3      #pragma HLS RESOURCE variable=input core=AXI4Stream metadata="-bus_bundle␣mac_i"
4      #pragma HLS RESOURCE variable=output core=AXI4Stream metadata="-bus_bundle␣c2s_i"
```

Figura 4.2: Declaración de entrada/salida de un módulo HLS

El *template* de la clase hls::stream<T>se utiliza para indicar el ancho de bus de datos u otras señales adicionales de la interfaz. En el caso de este diseño, se define con la estructura *output_interface_type* que se muestra en la Figura 4.3. A parte de las señales típicas de AXI-Stream, se está utilizando también la señal de last, que se activará cuando se envíen los últimos bytes de cada paquete.

La escritura y lectura de estas interfaces es muy sencilla, como se puede ver en el ejemplo de la Figura 4.4. No es necesaria ninguna clase de máquina de estados para

```
1  template<int D>
2    struct my_axis2{
3      ap_uint<D> data;
4      ap_uint<1> last;
5    };
6
7  typedef my_axis2<128> output_interface_type;
```

Figura 4.3: Estructura del bus

su gestión. Mediante llamadas a funciones de la clase stream, el programa codificará la gestión del bus. Esta característica ha sido muy beneficiosa a la hora de reducir el tiempo de programación. En la línea 7 de la Figura 4.4 se distingue cómo se lee de la interfaz del comparador y se almacena el valor en la variable *ip_net_in*. Esta clase de llamadas es bloqueante y, para ello, internamente, Vivado HLS creará una máquina de estados que permanecerá a la espera hasta que esté listo el dato que se lee de esa interfaz. A pesar de esto, el módulo no quedará bloqueado. En la siguiente sentencia se especifica que se desea leer de la interfaz *input*, así que el módulo intentará leer los datos de ella en paralelo mientras espera los datos de la otra y viceversa.

```
1     mac_user_interface_type ip_net_in, ip_net_out, maco, macd, ts;
2     output_interface_type axi_w, subredes;
3
4
5  principal: while(1){
6
7     comp_input.read(ip_net_in);
8     input.read(maco);
9     comp_input.read(ip_net_out);
10    input.read(macd);
11    timestamp.read(ts);
12    axi_w.data.range(127, 64) = macd.data;
13    axi_w.data.range(63, 0) = maco.data;
14
15    if(ip_net_in.data != 0 && ip_net_out.data!=0){
16
17        output.write(axi_w);
18        input.read(axi_rd);
19
20        axi_w.data.range(63, 0) = axi_rd.data;
21        size = axi_rd.data.range(63, 32);
22
23        input.read(axi_rd);
24        input.read(axi_rd2);
25
26        subredes.data.range(127, 64) = ip_net_out.data;
27        subredes.data.range(63, 0) = ip_net_in.data;
28
29        axi_w.data.range(127, 64) = axi_rd.data;
30        output.write(axi_w);
31
32        axi_w.data.range(63, 0) = axi_rd2.data;
33        axi_w.data.range(127, 64) = ts.data;
34
35        output.write(axi_w);
36        output.write(subredes);
```

Figura 4.4: Ejemplo de escritura y lectura de bus en Vivado HLS

El código completo realizado en lenguaje HLS se muestra en el Anexo B.

4.3. Arquitectura general de la FPGA

El diseño hardware se ha implementado sobre otra arquitectura que fue previamente codificada por el HPCN. En dicha arquitectura se utilizan cores IP ofrecidos por Xilinx y un core de DMA de NorthWest Logic que fue adquirido y probado para proyectos anteriores [25].

En la Figura 4.5 se muestra de forma esquemática la disposición e interconexión de los cores IP. Su funcionamiento es el siguiente: Los datos llegan por la red a través de un conector SFP+. Estos datos se procesan mediante el core IP de Xilinx que gestiona la *Physical Coding Sublayer* (PCS) y el *Physical Medium Attachment* (PMA). Su función consiste en realizar una conversión, independiente del medio, de la capa física a formato *10 Gigabit Media Independent Interface* (XGMII).

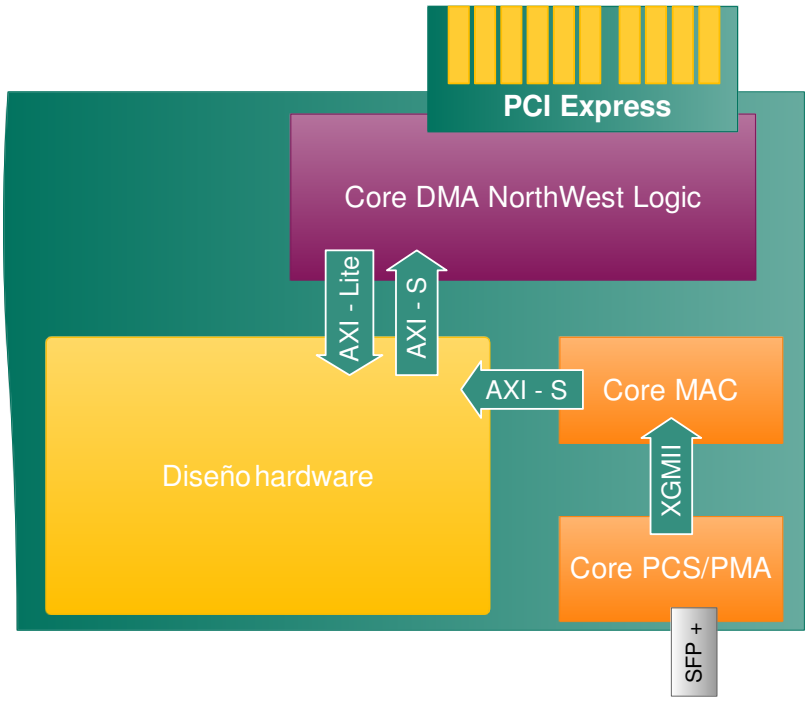

Figura 4.5: Arquitectura completa del diseño hardware.

A continuación los datos son procesados por un core de *Media Access Control* (MAC) que envía los paquetes en formato AXI-Stream eliminando los preámbulos del protocolo *802.3-2012 - IEEE Standard for Ethernet* [26]. Tanto este módulo como el core que

gestiona PCS/PMA funcionan a una frecuencia de reloj de 156,25 MHz, que es la requerida para recibir y enviar tráfico a 10 Gbps.

Los datos llegan al módulo diseñado que los procesa y genera las estructuras sintéticas para el sistema operativo. Una vez realizados todos los cálculos, los resultados se van almacenando en una cola que implementa el protocolo AXI-Stream. Es importante destacar que esta cola es asíncrona debido a la diferencia de dominio de reloj entre los diferentes módulos.

Por último, el core DMA (que utiliza a su vez un core de PCI Express de Xilinx) lee los datos de la FIFO a 125 MHz y transfiere los datos a la memoria RAM del ordenador en bloques de 16 MB. Este core IP también es capaz de leer y escribir en registros de la FPGA mediante los *Base Address Registers* (BAR) que ofrece el PCI Express. Esta característica se ha aprovechado para realizar la transferencia de la información de las subredes que se quieren monitorizar. Para escribir en la memoria donde se almacenan esos datos se ha utilizado una estructura de buses AXI-Lite ya que soporta escritura en direcciones. También se ha utilizado para implementar un *reset* del diseño.

4.4. Arquitectura del diseño realizado

A continuación se describe la funcionalidad de cada uno de los módulos que se han implementado o utilizado en el diseño hardware. Para tener una visión completa de la aplicación se presenta la Figura 4.6, que es un esquema ampliado del "Diseño hardware"de la Figura 4.5. Los bloques que aparecen en verde fueron codificados mediante el entorno de desarrollo Vivado HLS, los naranjas son cores IP que ofrece Xilinx y los bloques azules se escribieron en verilog. La interconexión de todos estos módulos se realizó también en este último lenguaje HDL.

De forma más detallada, las funcionalidades que realiza cada módulo son las siguientes:

- **Módulo ethernet-IP manager**

  Este módulo es el encargado de realizar toda la extracción de información de los paquetes. Lee los paquetes de una cola *First In First Out* (FIFO) donde previamente se han ido almacenando los datos. La interfaz de esta cola es AXI-Stream. A continuación extrae toda la información expuesta en la sección de diseño(3) y la envía a una FIFO donde se almacenarán temporalmente hasta que el resto de cálculos hayan sido realizados. Paralelamente al procesamiento, el módulo envía las direcciones IP origen y destino a otra cola.

  Para que sea capaz de procesar a tasa de línea los datos procedentes de la red, este módulo utiliza un reloj de 200 MHz en vez de 156,25 MHz, que es la frecuencia del resto de la aplicación por lo que todas las colas FIFO conectadas a él serán asíncronas.

- **Módulo data fifo**

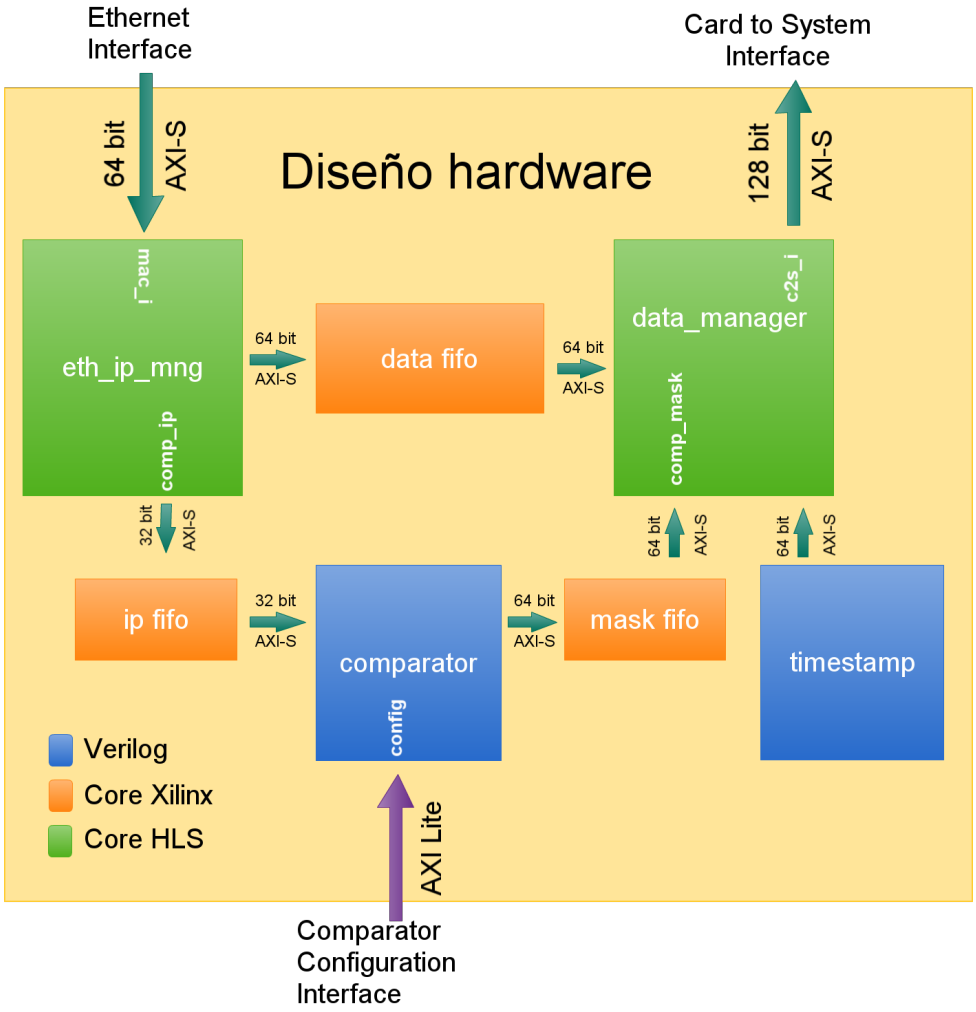

Figura 4.6: Arquitectura de la aplicación hardware diseñada.

Este módulo consiste en una cola FIFO asíncrona que se encarga de almacenar temporalmente los datos extraídos de las cabeceras a la espera de que todos los cálculos asociados a un paquete hayan sido realizados.

- **Módulo comparador**

  Este módulo recibe direcciones IP de 32 bit de la *ip fifo* con una interfaz AXI-Stream. Internamente posee una memoria con 64 subredes y sus máscaras asociadas. Para cada subred almacenada, calcula su pertenencia y genera un mapa de bits, donde cada bit a 1 representa la pertenencia a la subred que esté en esa posición. Este dat se enviará por otra interfaz AXI-Stream de 64 bit que pertenece a otra FIFO. La funcionalidad pensada para este módulo es la búsqueda de forma paralela al parseo de paquetes.

- **Módulo ip fifo**

  Este módulo consiste en una cola FIFO asíncrona que almacena las direcciones IP cuya pertenencia a subredes se quiere obtener. La existencia de esta cola se debe a

dos razones. En primer lugar al cambio de dominio de reloj explicado anteriormente. En segundo lugar, al retardo que introduce el comparador. Gracias a este módulo, el *ethernet_IP_manager* puede enviar varias direcciones en ciclos consecutivos sin tener en cuenta esta latencia.

- **Módulo mask fifo**

  Este módulo almacena la información producida por el comparador en espera de ser leída por el *data manager*.

- **Módulo de timestamp**

  Consiste en un contador que se va incrementando en 1 cada ciclo de reloj de 156,25 MHz. La conversión al formato de tiempo utilizado en software se llevará a cabo durante la transferencia de datos.

- **Módulo data manager**

  Este módulo se encarga de recolectar todos los datos que han sido calculados por separado. Lee de las FIFO con interfaces AXI-Stream de 64 bit pero envía con 128 bit (respetando la tasa de PCI Express, ya que escribe a 156,25 MHz pero el PCI Express lee a 125 MHz). En primer lugar lee de la cola FIFO donde el comparador fue almacenando su salida. Si un paquete pertenece a las subredes que se están analizando, es decir si las máscaras de bits son distintas de 0, procederá a leer los datos obtenidos de la FIFO de datos y a enviarlos. A continuación, envía el *timestamp* calculado por el módulo correspondiente. Por último, escribe las máscaras de subredes detrás de la información de cada paquete. En el caso de que un paquete no perteneciera a ninguna de las subredes que se están monitorizando, no se envía nada al anfitrión, pero sí es necesario extraer los datos de la FIFO de datos del paquete para que quede sincronizada con el comienzo del siguiente.

4.5. Arquitectura de *DetectPro*

Para el cumplimiento de los requisitos de software 1 - 5 definidos en la sección 3.3.2, se ha reutilizado un programa ya codificado del que se dispone, *DetectPro*. La Figura 4.7, obtenida de [17], muestra la arquitectura de éste. Consiste en un módulo a nivel de kernel que captura los paquetes de la red, *Traffic Sniffer*, y tres módulos a nivel de usuario, *Packet Dumper*, *Flow Manager* y *Flow Exporter*.

El flujo de trabajo del sistema es el siguiente: Cada paquete que llega a la interfaz Ethernet, se transfiere mediante el *Network Interface Controller* (NIC) vía DMA a un *buffer* en anillo a nivel de núcleo. El *packet sniffer* comprueba el *buffer* en busca de nuevos paquetes continuamente. Si hay uno disponible, lo copia en un *buffer* más grande que es accesible desde el nivel de usuario. El módulo *Packet Dumper* lee bloques de tamaño fijo del *buffer* del paquete y los escribe en el disco para su posterior análisis (este módulo no se está utilizando en este trabajo). El módulo *Flow Manager* lee los paquetes de uno en uno del *buffer*, y los procesa actualizando la tabla de flujos en memoria (volátil) a la vez que

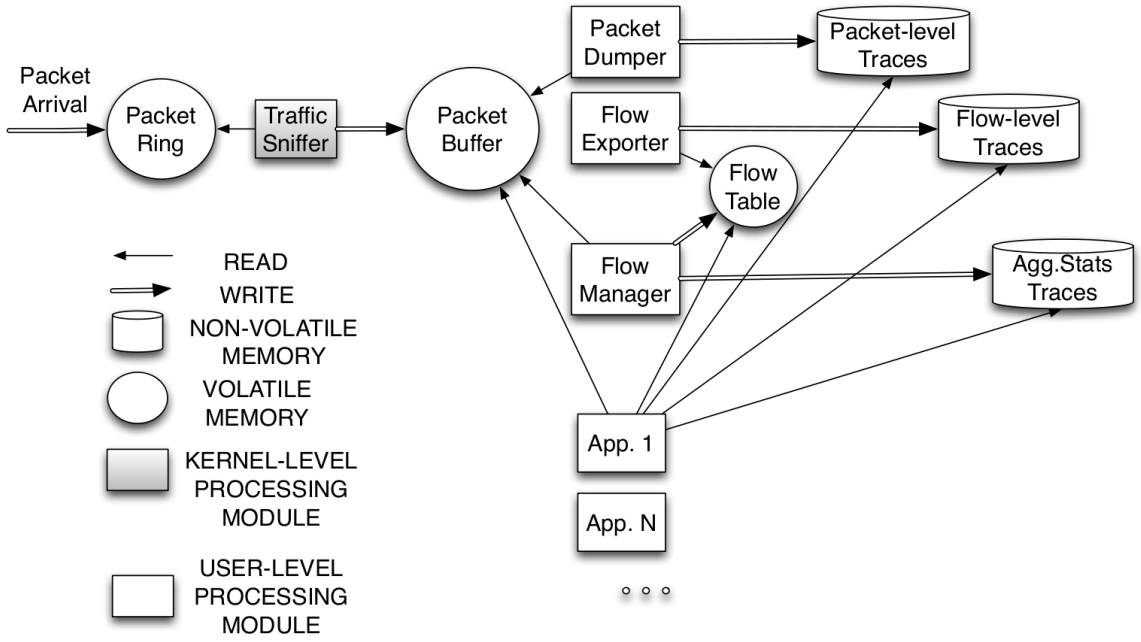

Figura 4.7: Arquitectura de *DetectPro* [17].

recolecta estadísticas de grano fino en el disco. El módulo *Flow Exporter* comprueba la tabla de flujos, exportando los registros de flujos expirados (incluyendo los primeros bytes del payload) al disco. Cabe destacar que estas tareas se ejecutan de forma simultánea en diferentes cores, aplicando afinidad (la ejecución de cada hilo del programa está fijada en un núcleo del procesador en concreto), para aprovechar al máximo el paralelismo de las arquitecturas multi-core o multi-procesador.

La tarea realizada consistió en sustituir el módulo *Packet Sniffer* y el módulo a nivel de núcleo con su correspondiente *driver* por otro módulo que transfiera los datos desde la FPGA al *DetectPro*, también mediante transferencias DMA. De esta forma, todo el parseo y extracción de los datos de los protocolos IP y transporte se realizará en la FPGA, dejando al *DetectPro* con la única tarea de actualizar la tabla de flujos y exportar al disco los flujos expirados, así como otras estadísticas que interesen.

4.6. Comunicación entre arquitecturas

Para que toda la arquitectura descrita en las secciones previas pueda funcionar correctamente y a la tasa deseada, es necesario un *driver* de altas prestaciones que realice las transferencias de DMA. Se ha utilizado uno previamente diseñado por el HPCN [25] modificado para este proyecto. El funcionamiento de este driver fue probado en un diseño previo que requería la misma tasa de transferencia entre FPGA y memoria RAM que este diseño.

Las transferencias se realizan en bloques de 16 MB. Esta cantidad de datos es lo

suficientemente grande como para que los descriptores que se usan para las transferencias no añadan mucho *overhead* y lo suficientemente pequeña como para que no se tenga que producir mucho tiempo entre transferencias sin procesamiento. Estas transferencias se copian páginas de memoria RAM sin que el procesador intervenga en el proceso ya que son DMA. Los datos se almacenan temporalmente en páginas de memoria especiales (*huge pages*) de tamaño 1 GB. En este sistema se utilizan 8 páginas de este tamaño. Los datos que se reciben en bloques y se copian en las estructuras siguiendo este Algoritmo 1.

Algorithm 1 Transmisión de datos

```
1:  procedure PROCESAR
2:      if Quedan datos transmitidos por DMA then
3:          if Los datos disponibles son suficientes para un paquete then
4:              Copiar los datos de la estructura
5:          else
6:              Copiar los datos disponibles a un buffer
7:              Esperar la transmisión del siguiente bloque
8:          Copiar subredes a las que pertenece
9:          Ajustar timestamp
10:     else
11:         Pedir datos
12:         Esperar transmisión
```

Esta función se invoca desde el programa principal cada vez que está listo para clasificar un paquete.

5 Pruebas y resultados

En este capítulo se detallan las diferentes simulaciones y pruebas que se han realizado para verificar y validar el diseño.

5.1. Simulaciones

Como se mencionó en la sección 3.4, el desarrollo se ha llevado a cabo mediante diferentes herramientas y cada una de ellas posee mecanismos de prueba diferentes. Estas simulaciones se corresponden con el subsistema hardware de este proyecto. A continuación se describen brevemente las pruebas que se han realizado en estos entornos.

5.1.1. Simulación software

El entorno Vivado HLS, como se explicó previamente, se generan módulos hardware a partir de funciones en C++.

La simulación de estos módulos consiste en la ejecución de un programa en C++ que contiene una llamada al módulo que se ha programado. Para simular la interfaz de red, se utiliza una clase específica de Xilinx llamada hls::stream. Antes de realizar la llamada al módulo, se guardan varios paquetes leídos de una traza en formato *libpcap*. Estas ejecuciones se pueden *debuguear* con una herramienta incluida en el entorno.

5.1.2. Simulación hardware

El entorno Vivado proporciona una herramienta de simulación de hardware. Dado que la realizar un *testbench* de todo el mecanismo del PCI Express es un trabajo inabarcable para este proyecto y ya ha sido probado por la empresa desarrolladora del core IP que realiza las transferencias DMA, se ha simulado únicamente la salida del módulo de Ethernet. Esta simulación ha consistido en generar el *stream* de datos a partir de una traza en formato *libpcap* con un parseador de esta clase de ficheros escrito en verilog. La salida de este módulo se conecta al diseño realizado como se muestra en la Figura 5.1. La salida del módulo diseñado se escribe en un fichero. Para comprobar la validez de esta salida se han desarrollado en este trabajo programas auxiliares.

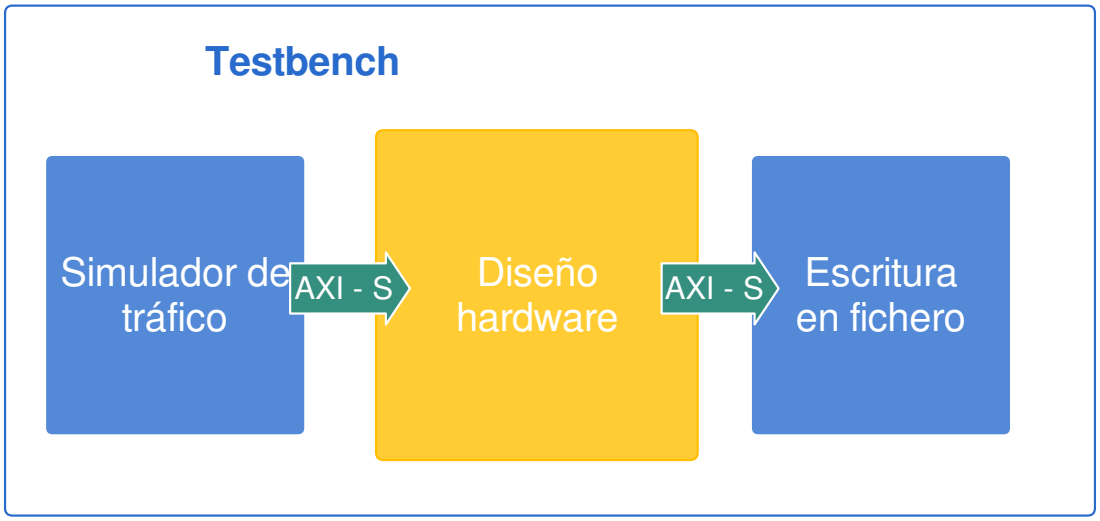

Figura 5.1: Esquema de la simulación verilog.

5.1.3. Simulación de *DetectPro*

Para poder probar de forma independiente la parte del programa hardware y software, se han implementado las funciones del *driver* de forma que simulan las transacciones DMA leyendo de un fichero normal. Este fichero se genera mediante la simulación del hardware que se describió en el apartado anterior.

5.2. Pruebas con entorno real

5.2.1. Configuración del sistema

Configuración de arranque

Para la transferencia correcta y eficiente de datos mediante DMA, el *driver* necesita páginas de memoria de tamaño no estándar. Por lo tanto en la entrada del inicio de Linux habrá que incluir los siguientes argumentos:

- *default_ hugepagesz=1G*: Establece el tamaño por defecto de las páginas de memoria no convecionales a 1 GB.
- *hugepagesz=1G*: Indica que se usará el tamaño de 1 GB.
- *hugepages=8*: Indica el número de páginas que reservará inicialmente el sistema operativo y que serán las que solicite el *driver* para las transferencias DMA.

Configuración de la BIOS

Por defecto, está activada la opción de *hyperthreading* del sistema operativo. Sin embargo, al interesarnos la capacidad de cálculo a velocidad rápida, es mejor desactivarlo ya que el *scheduler* del sistema operativo podría cambiar de núcleo la ejecución del hilo a otros cores simulados, lo que provocaría la recarga de las cachés aunque no fuera necesario. En el caso de la máquina sobre la que se realizarán los experimentos, se pasará de 8 cores virtuales a 4 reales.

También se debe especificar que la conexión de PCI Express que se usa es de segunda generación con 4 líneas.

5.3. Estadísticas de la tarjeta

El diseño hardware ocupa los recursos de la tarjeta que se muestran en la Tabla 5.1 y en la Figura 5.2. Como se puede observar, quedan bastantes recursos libres en la FPGA por lo que podría considerarse la posibilidad de utilizar un encapsulado de menores prestaciones con la finalidad de reducir el precio.

En cuanto a las restricciones de tiempo se cumplen satisfactoriamente. Se ha obtenido un *Worst Negative Slack* de 0,214 ns y un *Worst Hold Slack* de 0,030 ns.

Tabla 5.1: Estadísticas de ocupación de la tarjeta

Recurso	Utilización	Disponible	Porcentaje Utilizado
Flip-Flops	41166	407600	10,10 %
Look-Up Tables	29817	203800	14,63 %
Memory LUT	2699	64000	4,22 %
Entrada/Salida	10	500	2,00 %
BRAM	165	445	37,08 %
BUFG	9	32	28,12 %
MMCM	1	10	20,00 %
Gigabit Transceivers	5	20	25,00 %

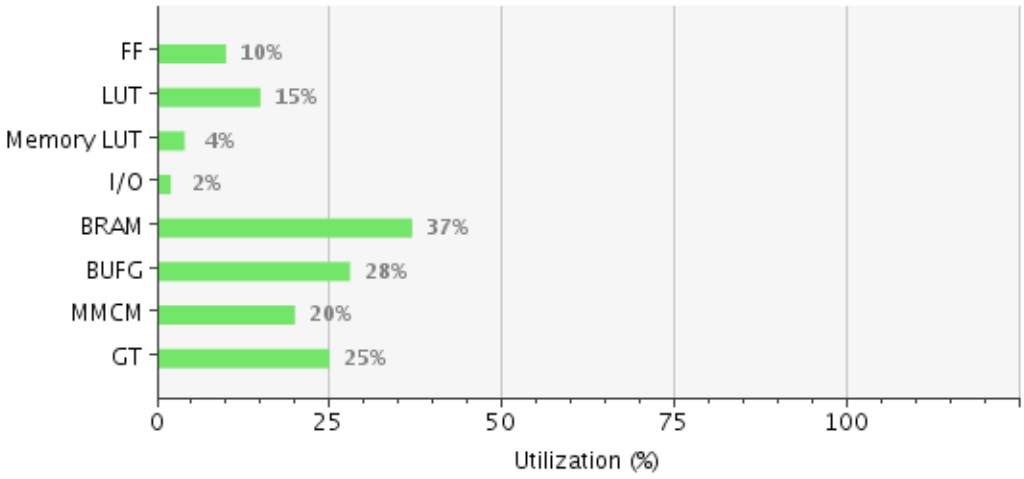

Figura 5.2: Utilización de la FPGA.

5.4. Entorno del experimento

En esta sección se describe el *testbed* experimental, especificando los detalles hardware, software y de tráfico.

Hardware

Se dispone de dos ordenadores, uno que recibe y otro que genera tráfico. El ordenador encargado de recibir tiene un procesador de 4 núcleos Intel Xeon E5-1620 a 3,60 GHz con 16 GB de memoria RAM Quad Channel a 1600 MHz DDR3 y una placa modelo kc705 de Xilinx que contiene una FPGA Kintex-5 con encapsulado xc7k325tffg900-2, conexión de PCI Express y una interfaz SFP+ [21].

El ordenador encargado de enviar está dotado de un Intel(R) Xeon(R) E5-1620 a 3,6 GHz con 4 núcleos e *hyperthreading* habilitado. Para generar tráfico se ha utilizado un diseño [25] programado en una NetFPGA, una placa que contiene una FPGA Virtex-5 de Xilinx y 4 conectores SFP+ por los que se pueden enviar datos a 10 Gbps. Este

diseño permite generar paquetes con datos aleatorios de tamaño fijo con un *interframe gap* configurable por el usuario. También existe la posibilidad de enviar paquetes que fueron previamente almacenados en una traza.

Ambos equipos están conectados con un cable de fibra óptica.

Software

En el sistema receptor Scientific Linux 6.2 con versión de 64-bit y kernel de Linux 2.6.32.

El sistema emisor funciona con Scientific Linux 6.4 de 64-bit y el software utilizado para generar y enviar tráfico fue previamente programado en el HPCN [25].

Tráfico

El tráfico que se ha reproducido es parte de una traza anonimizada del enlace CAIDA, almacenada en un centro de datos de Equinix en Chicago [27].

5.5. Pruebas de rendimiento del módulo hardware

Para las pruebas individuales del módulo hardware se han utilizado dos programas diferentes.

5.5.1. Pruebas de procesamiento hardware y transferencia

Para estos experimentos se ha utilizado un programa que sólo se encarga de realizar las transferencias de datos. Se colocaron dos contadores de paquetes mediante registros de la FPGA, uno a la entrada del módulo diseñado, encargado de contar los paquetes recibidos y otro a la salida de los datos producidos que contabiliza el número de paquetes procesados. Se han recibido el 100 % de los paquetes enviados a 10 Gbps en las pruebas que se muestran en la Tabla 5.2.

Tabla 5.2: Rendimiento individual del módulo hardware (sólo transferencia)

Tamaño medio de paquete	Nº de paquetes recibidos
976 Bytes	1.000.000
976 Bytes	3.000.000
64 Bytes	50.000.000
64 Bytes	100.000.000
64 Bytes	150.000.000

5.5.2. Pruebas de procesamiento en software

Se ha codificado un programa sencillo que pide los datos de cada paquete de uno en uno, sin realizar ninguna clase de procesamiento. Se han realizado pruebas de la misma forma que en la sección anterior. Se han procesado el 100 % de los paquetes en los casos que aparecen en la Tabla 5.3.

Tabla 5.3: Rendimiento con procesamiento en software

Tamaño medio de paquete	Nº de paquetes recibidos
976 Bytes	1.000.000
976 Bytes	3.000.000
64 Bytes	50.000.000
64 Bytes	100.000.000
64 Bytes	150.000.000
64 Bytes	300.000.000

5.6. Uso de CPU

En cuanto al uso del procesador, si se ejecuta el programa descrito en la sección 5.5.2, se comprueba que no consume recursos de la CPU si no se recibe tráfico. Esto se debe a que la espera de las transferencias de datos es bloqueante. Además, estas transferencias, realizadas mediante DMA hacen que no sea necesario el uso del procesador. Este comportamiento se puede apreciar en la Figura 5.3. El pico de la CPU1 (línea naranja) se corresponde con el inicio de la aplicación.

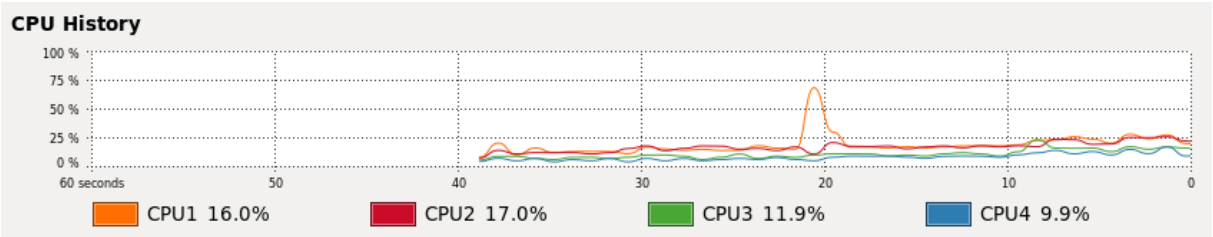

Figura 5.3: Uso de CPU sin recepción de paquetes.

En la Figura 5.4 se muestra el uso de CPU de dicho programa en el caso de recepción a 14 Mpbs. Se puede observar que, de media, no se utilizan más de 2 núcleos del procesador.

En cuanto al uso de CPU del sistema completo, es decir, incluyendo la integración del desarrollo hardware con el software, se puede observar en la Figura 5.5 el rendimiento del programa sin recepción de tráfico. El pico de uso de la CPU2 (en rojo) se corresponde con la inicialización del programa. Como se puede observar, tampoco consume recursos si no recibe tráfico, al contrario que *DetectPro*, que, como se puede ver en la Figura 5.6 consume dos núcleos.

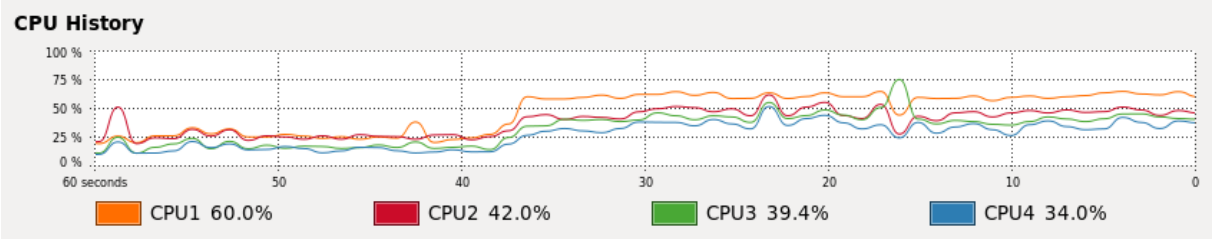

Figura 5.4: Uso de CPU con recepción de 14 Mpps.

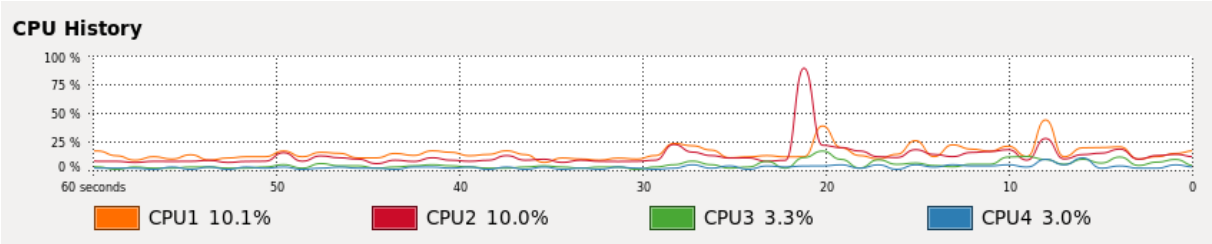

Figura 5.5: Uso de CPU del sistema sin procesamiento.

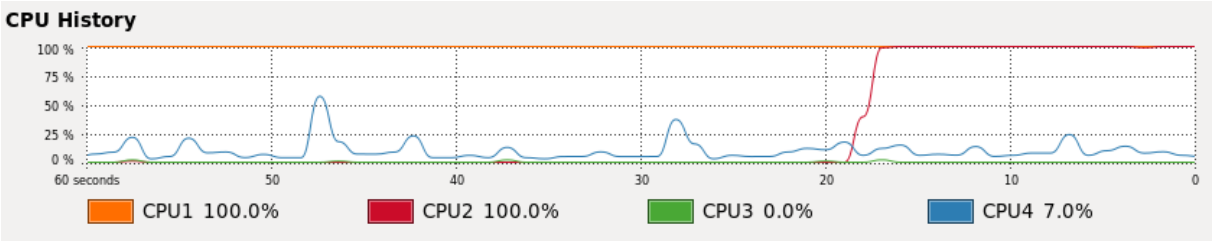

Figura 5.6: Uso de CPU de *DetectPro* sin procesamiento.

Por último, la utilización del procesador del sistema desarrollado con recepción a máxima tasa se muestra en la Figura 5.7. Se puede observar que no se utiliza en ningún caso más de dos núcleos y no supera a *DetectPro* en consumo de CPU.

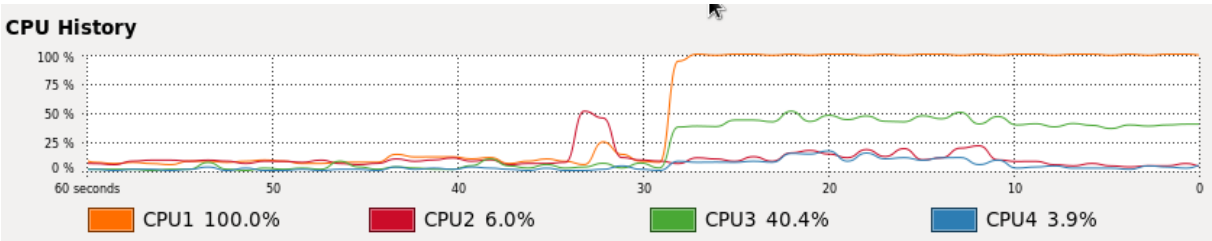

Figura 5.7: Uso de CPU del sistema con recepción de 14 Mpps.

5.7. Comparativa con *DetectPro* original

En la Figura 5.8 se muestran los experimentos realizados con *DetectPro* y el sistema acelerado en el caso en el que se generan flujos y se monitoriza una única subred.

La línea azul se corresponde con el procesamiento de *DetectPro* original. La línea naranja representa el rendimiento del sistema desarrollado en este trabajo. En este caso es peor debido a que no se ha aprovechado al máximo paralelismo ofrecido por la FPGA a la hora de clasificar, ya que sólo se monitoriza una única subred.

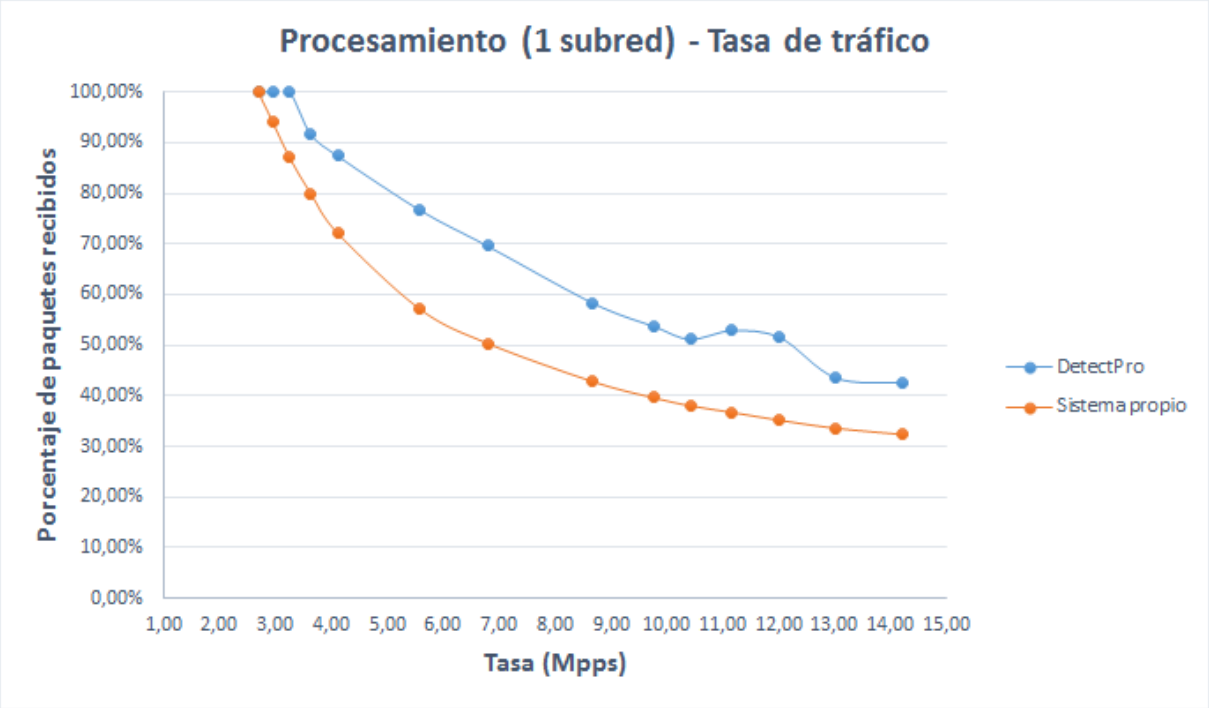

Figura 5.8: Procesamiento de paquetes con monitorización de 1 subred.

En la Figura 5.9 se muestran los experimentos realizados con *DetectPro* y el sistema acelerado. Las línea azul representa la cantidad de paquetes procesados por *DetectPro* y la naranja el rendimiento del sistema construido. Se corresponde con la ejecución de los programas realizando una monitorización de 64 subredes. Cabe destacar que la pérdida de paquetes en el caso de este sistema se produce a nivel de software. Como se vio en las secciones 5.5.1 y 5.5.2 la FPGA sí es capaz de procesar todo el tráfico.

El sistema desarrollado en este trabajo presenta una mejora si la tasa de recepción de paquetes es menor que aproximadamente 7 Mpps. Esta tasa es aceptable ya que, para el caso en el que esta información es relevante (si se realizan ataques informáticos), es muy difícil que se llegue a la tasa máxima de paquetes. En estas circunstancias los paquetes se verían mezclados con el tráfico normal, que reduce, con mucho, la cantidad media de paquetes por segundo recibidos.

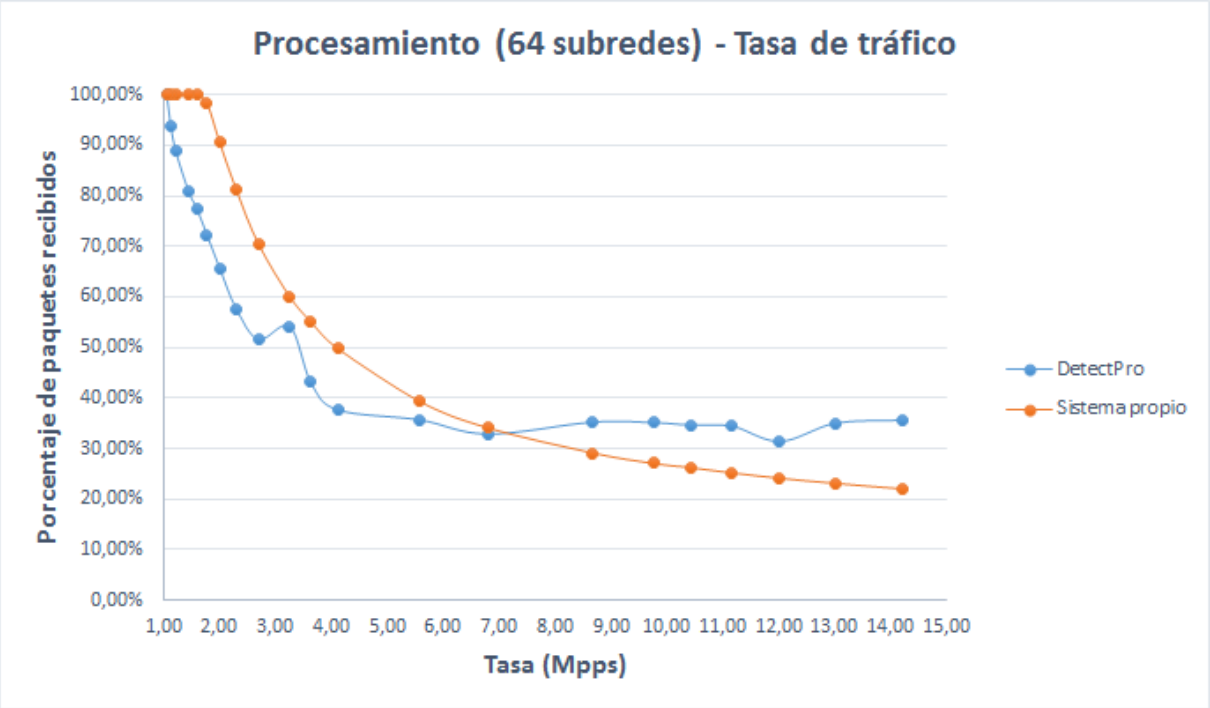

Figura 5.9: Procesamiento de paquetes con monitorización de 64 subredes.

6

Conclusiones

Las conclusiones que se expondrán en este capítulo relacionarán los objetivos y retos propuestos al inicio de este documento con los logros que se han ido obteniendo a lo largo del tiempo dedicado a la realización de este proyecto.

Los pasos seguidos para la construcción de un sistema de generación de flujos a alta velocidad han permitido conseguir el objetivo propuesto de diseñar un módulo que fuera capaz de analizar a tasa de línea de una red multigigabit ethernet (10 Gbps) el tráfico íntegramente, así como de extraer la información necesaria de cada paquete individual que permitiera la generación de flujos válidos. Se ha conseguido el rendimiento deseado gracias a una plataforma hardware basado en FPGA que soporta el procesado del tráfico de red a una tasa de 10 Gbps. Este sistema hardware permite también realizar los filtrados que se buscaban en los requisitos iniciales del proyecto.

En cuanto a los objetivos software, se ha conseguido encontrar un mecanismo de comunicación de la información desde la FPGA al ordenador anfitrión que permite el envío de datos a 10 Gbps (o más) sin que ello ralentice el procesamiento de los mismo en software ni requiera más de 3 núcleos de un procesador convencional, gracias a la adaptación específica un *driver* previamente desarrollado.

La carga de trabajo del desarrollo de este Trabajo de Fin de Grado se ha visto reducida gracias al aprovechamiento de un programa ya desarrollado, *DetectPro*, aunque la tarea de realizar una integración con el módulo hardware no haya sido en absoluto trivial.

El rendimiento obtenido del procesamiento en FPGA ha sido máximo, ya que consigue capturar, procesar y enviar al ordenador anfitrión todo el tráfico a 10 Gbps y 14 Mpbs.

Los resultados obtenidos del sistema completo, en cuanto al uso de CPU, son mejores, ya que reduce el uso de tres núcleos del procesador a aproximadamente dos. Esto

representa una notoria mejora.

En los resultados de análisis de tráfico se observa una mejora considerable cuando se analizan 64 subredes en paralelo con una tasa de recepción menor o igual a 7 Mpps. Cabe destacar que la pérdida de estos paquetes se produce por el cuello de botella que supone la construcción de flujos a nivel de software.

El uso de una FPGA ha mejorado el rendimiento a la hora de capturar paquetes sin implicar un tiempo de desarrollo elevado gracias a la herramienta elegida. Ésta ha sido muy beneficiosa, a pesar del tiempo dedicado para el aprendizaje de su uso. La metodología utilizada es una estrategia realista en términos tecno-económicos, ya que la FPGA requerida para este proyecto no tiene un coste elevado y la herramienta seleccionada no requiere unos tiempos de desarrollo extensos.

En resumen, los objetivos propuestos inicialmente se han alcanzado satisfactoriamente gracias a la construcción de un sistema híbrido compuesto por hardware dedicado y un programa software. En cuanto a los retos propuestos, se considera que la herramienta elegida ha sido apropiada, ya que ha sido tan eficiente como requería el proyecto y ha permitido que las tareas se desarrollen en el tiempo deseado.

7

Trabajo Futuro

Los resultados ha sido positivos, pero el tiempo que se puede dedicar a un Trabajo de Fin de Grado no es suficiente para conseguir todas las funcionalidades deseables, así como sus correspondientes optimizaciones.

Aún quedan muchas tareas de mejora de este sistema que no estaban dentro del alcance inicial del proyecto, como, por ejemplo, el etiquetado de flujos mediante su *payload* o la generalización en la monitorización de cualquier número de subredes. Actualmente, el sistema está limitado a la monitorización de 64 subredes, un número insuficiente para las necesidades empresariales. Esto se debe a que la clasificación se realiza en hardware con comparadores en paralelo. Un trabajo futuro interesante será la sustitución de estos módulos por memorias de tipo TCAM, de forma que permitiría una ampliación a un número mucho mayor de subredes.

En cuanto a la integración del software y hardware, las líneas de investigación futuras se pueden centrar tanto en la reducción de uso de recursos del procesador (manteniendo el nivel de procesamiento actual), como en el aumento de la capacidad de procesamiento de paquetes por segundo.

Bibliografía

[1] A. W. Moore and D. Zuev, "Internet traffic classification using bayesian analysis techniques," in *ACM SIGMETRICS Performance Evaluation Review*, vol. 33, no. 1. ACM, 2005, pp. 50–60.

[2] "Netflow services and solution guide," http://www.cisco.com/c/en/us/td/docs/ios/solutions_docs/netflow/nfwhite.html, accessed: 2015-06-17.

[3] "Cisco systems, inc," http://www.cisco.com/, accessed: 2015-06-17.

[4] J. Flick and J. Johnson, "Cisco Systems NetFlow Services Export Version 9," Internet Requests for Comments, RFC 3954, October 2004. [Online]. Available: https://tools.ietf.org/html/rfc3954

[5] E. Boschi, H. Europe, L. Mark, F. FOKUS, J. Quittek, M. Stiemerling, NEC, P. Aitken, and I. Cisco Systems, "IP Flow Information Export (IPFIX) Implementation Guidelines," Internet Requests for Comments, RFC 5153, April 2008. [Online]. Available: https://tools.ietf.org/html/rfc5153

[6] B. Trammell, E. Boschi, H. Europe, L. Mark, , F. IFAM, T. Zseby, F. FOKUS, A. Wagner, and E. Zurich, "Specification for the IPFIX File Format," Internet Requests for Comments, RFC 5655, October 2009. [Online]. Available: https://tools.ietf.org/html/rfc5655

[7] "Netfpga 10g," http://netfpga.org/site/#/systems/3netfpga-10g/details/, accessed: 2015-05-03.

[8] "Xilinx inc," http://www.xilinx.com/, accessed: 2015-02-16.

[9] "Netflow probe in netfpga-1g," https://github.com/NetFPGA/netfpga/wiki/NetFlowProbe#NetFlowProbe, accessed: 2015-05-16.

[10] S. Yusuf, W. Luk, M. Sloman, N. Dulay, E. C. Lupu, and G. Brown, "Reconfigurable architecture for network flow analysis," *Very Large Scale Integration (VLSI) Systems, IEEE Transactions on*, vol. 16, no. 1, pp. 57–65, January 2008.

[11] M. Forconesi, G. Sutter, S. Lopez-Buedo, and J. Aracil, "Accurate and flexible flow-based monitoring for high-speed networks," in *Field Programmable Logic and Applications (FPL), 2013 23rd International Conference on*. IEEE, September 2013.

[12] "Flowmon probe," https://www.invea.com/en/products-and-services/flowmon/flowmon-probes, accessed: 2015-05-16.

[13] M. Žádník, L. Polčák, O. Lengál, M. Elich, and P. Kramoliš, "Flowmon for network monitoring," in *Networking Studies V : Selected Technical Reports.* Prague: CESNET, z.s.p.o., 2011, pp. 135–153.

[14] "Ubuntu manpage: fprobe-ulog - a newflow probe," http://manpages.ubuntu.com/manpages/precise/man8/fprobe-ulog.8.html, accessed: 2015-05-17.

[15] "nprobe v7," http://www.ntop.org/products/netflow/nprobe/, accessed: 2015-05-16.

[16] M. Danelutto, L. Deri, and D. De Sensi, "Network monitoring on multicores with algorithmic skeletons." in *PARCO*, 2011, pp. 519–526.

[17] P. M. S. del Río and J. A. Rico, "Internet traffic classification for high-performance and off-the-shelf systems," 2013.

[18] 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," in *High Performance Computing and Communications, 2014 IEEE 6th Intl Symp on Cyberspace Safety and Security, 2014 IEEE 11th Intl Conf on Embedded Software and Syst (HPCC, CSS, ICESS), 2014 IEEE Intl Conf on.* IEEE, August 2014, pp. 486–489.

[19] "Data plane development kit," http://dpdk.org/, accessed: 2015-06-17.

[20] O. Arcas-Abella, G. Ndu, N. Sonmez, M. Ghasempour, A. Armejach, J. Navaridas, W. Song, J. Mawer, A. Cristal, and M. Lujan, "An empirical evaluation of high-level synthesis languages and tools for database acceleration," in *Field Programmable Logic and Applications (FPL), 2014 24th International Conference on.* IEEE, September 2014, pp. 1–8.

[21] "Xilinx kintex-7 fpga kc705 evaluation kit," http://www.xilinx.com/products/boards-and-kits/ek-k7-kc705-g.html, accessed: 2015-06-17.

[22] "Vivado hls," http://www.xilinx.com/products/design-tools/vivado/integration/esl-design.html, accessed: 2015-02-16.

[23] "Vivado," http://www.xilinx.com/products/design-tools/vivado.html, accessed: 2015-02-16.

[24] "Gdb: The gnu project debugger," http://www.gnu.org/software/gdb/, accessed: 2015-02-23.

[25] J. F. Zazo Rollón *et al.*, "Sistema basado en fpga para la captura de tráfico en redes multigigabit ethernet," September 2014.

[26] "802.3-2012 - ieee standard for ethernet - section 4," *IEEE Std. 1516-2000*, vol. 4, December 2012.

[27] "Passive monitor equinix-chicago," http://www.caida.org/data/monitors/passive-equinix-chicago.xml, accessed: 2015-04-05.

Apéndices

A

Cores IP utilizados

Cores utilizados en la FPGA.

- **Core DMA**

  Se ha decidido reutilizar este core desarrollado por Northwest Logic ya que facilita la comunicación mediante el bus PCI Express (hasta tercera generación), posee múltiples motores DMA de alto rendimiento y soporta la interfaz AXI4-Stream (utilizada en todo el diseño).

- **Ten Gigabit Ethernet PCS/PMA (10GBASE-R/KR)**

  La interfaz 10GBASE-R/KR es una interfaz serie. Está pensada para que provea funcionalidad de la ”Physical Coding Sublayer"PCS y la ”Physical Medium Attachment"PMA entre la interfaz de 10 Gigabit independiente del medio XGMII en un controlador de acceso al medio de 10 Gigabit MAC y el lado físico de la interfaz de red (PHY).

  Este core IP tiene una interfaz XGMII para una comunicación simple con el MAC de 10 Gigabit y un sistema de configuración mediante una interfaz MDIO.

- **Ten Gigabit Ethernet MAC**

  Es el core que se encarga de convertir los datos del formato XGMII (independiente del medio) a un formato AXI-Stream, con el que trabajará la aplicación. Ofrece también la opción de ser configurado mediante una interfaz *Management Data Input/Output* (MDIO). Este core también ofrece estadísticas de los paquetes recibidos

- **AXI Procotol Converter**

En el bloque de control, es necesario cambiar la interfaz que se recibe del PCI Express, AXI Full, AXI-Lite, que es el tipo de interfaz que se utiliza en todo el diseño, por ello, la entrada este bloque se conecta a la interfaz que proviene de la comunicación con el anfitrión y la salida un *crossbar* que separa los datos.

- **AXI Crossbar**

  Para configurar o leer los registros de estado los diferentes elementos de la arquitectura, el sistema operativo se comunica mediante una única interfaz. Gracias al acceso a diferentes posiciones de memoria se indica al diseño a qué configuración se quiere acceder. Este bloque se encarga de esta tarea, separa las instrucciones de configuración recibidas, en función de a qué core vaya dirigido. En este caso, los datos recibidos se dividen entre el bloque de control y el bloque del comparador de la siguiente forma:

Interfaz maestra	**Dirección base**	**Ancho de la dirección**
M00_AXI (mgt_block)	0x00000	16
M01_AXI (comparador)	0x10000	13

  A su vez, el bloque de control también debe repartir la escritura y lectura en los registros de configuración entre el bus *Inter-Integrated Circuit* (I^2C), la interfaz de configuración del core Ethernet, y de la aplicación.

Interfaz maestra	**Dirección base**	**Ancho de la dirección**
M00_AXI (bus iic)	0x02000	11
M01_AXI (core ethernet)	0x02800	11
M02_AXI (app)	0x03000	11

- **AXI Clock Converter**

  Como se observó en la Figura 4.5, no todo el diseño se rige por la misma señal de reloj. La comunicación con el ordenador se realiza a 125 MHz y ancho de bus de datos de 128 bit y los datos procedentes de la red se envían y reciben a 156,25 MHz con ancho de bus de 64 bit. Por ello son necesarios conversores de frecuencia de acceso a los datos que se comunican mediante AXI-Stream.

- **AXI4-Stream Data Width Converter**

  Como se explica en el core anterior, también es necesario realizar una conversión en el ancho del bus de datos a la hora de las transferencias.

- **AXI IIC**

  El diseño utiliza el reloj proporcionado por el ordenador mediante el bus PCI Express pero, como se comentó anteriormente, se necesita un reloj diferente para el uso de la interfaz Ethernet. Para ello se hace uso de un oscilador incluído en la FPGA (Si5326) que genera una señal de 156,25 MHz. La configuración de este oscilador se realiza mediante la escritura en sus registros de la información que va atravesando los diferentes *crossbar*.

- **FIFO**

  Se utilizan varias FIFOs en el diseño para evitar la pérdida de datos así como facilitar la sincronización entre los diferentes módulos.

B

Código HLS de la aplicación

El módulo *data manager* se muestra en la Figura B.1.

El la función principal del módulo *eth_ip_manager* se muestra en la Figura B.2. En la Figura B.3 se muestra el código en caso de que se haya recibido un paquete sin etiqueta de VLAN o número de etiquetas par. Y en la Figura B.4 se muestra el código de parseo de información en el caso de que se haya recibido un paquete con un número impar de etiquetas VLAN.

```
1
2  void data_manager_v2(stream<mac_user_interface_type> &input, stream<output_interface_type>
       ↪ &output, stream<mac_user_interface_type> &comp_input,
       ↪ stream<mac_user_interface_type> &timestamp) {
3
4      #pragma HLS RESOURCE variable=input core=AXI4Stream metadata="-bus_bundle␣mac_i"
5      #pragma HLS RESOURCE variable=output core=AXI4Stream metadata="-bus_bundle␣c2s_i"
6      #pragma HLS RESOURCE variable=comp_input core=AXI4Stream metadata="-bus_bundle␣
           ↪ comp_mask"
7      #pragma HLS RESOURCE variable=timestamp core=AXI4Stream metadata="-bus_bundle␣ts"
8
9
10     ap_uint<32> size;
11
12     int cont;
13
14     subredes.last = 1;
15     cont_w.last = 0;
16
17     mac_user_interface_type ip_net_in, ip_net_out, maco, macd, ts;
18     output_interface_type axi_w, subredes;
19
20
21 principal: while(1){
22
23     comp_input.read(ip_net_in);
24     input.read(maco);
25     comp_input.read(ip_net_out);
26     input.read(macd);
27     timestamp.read(ts);
28     axi_w.data.range(127, 64) = macd.data;
29     axi_w.data.range(63, 0) = maco.data;
30
31     if(ip_net_in.data != 0 && ip_net_out.data!=0){
32
33         output.write(axi_w);
34         input.read(axi_rd);
35
36         axi_w.data.range(63, 0) = axi_rd.data;
37         size = axi_rd.data.range(63, 32);
38
39         input.read(axi_rd);
40         input.read(axi_rd2);
41
42         subredes.data.range(127, 64) = ip_net_out.data;
43         subredes.data.range(63, 0) = ip_net_in.data;
44
45         axi_w.data.range(127, 64) = axi_rd.data;
46         output.write(axi_w);
47
48         axi_w.data.range(63, 0) = axi_rd2.data;
49         axi_w.data.range(127, 64) = ts.data;
50
51         output.write(axi_w);
52         output.write(subredes);
53         output.write(cont_w);
54     }
55     else{
56         input.read(axi_rd);
57         size = axi_rd.data.range(63, 32);
58         input.read(axi_rd);
59         input.read(axi_rd);
60     }
61 }
62 }
```

Figura B.1: Código de *data_manager*

```
1  void eth_ip_lvl4_mng(stream<mac_user_interface_type> &input,
       ↪ stream<mac_user_interface_type> &output, stream<uint32_t> &comp_output) {
2
3      #pragma HLS RESOURCE variable=input core=AXI4Stream metadata="-bus_bundle␣mac_i"
4      #pragma HLS RESOURCE variable=output core=AXI4Stream metadata="-bus_bundle␣c2s_i"
5      #pragma HLS RESOURCE variable=comp_output core=AXI4Stream metadata="-bus_bundle␣comp_ip"
6
7      ap_uint<64> maco, macd;
8      uint64_t cont_ip = 0, cont_cap = 0;
9      uint8_t transport_prot, frag_flag=0, vlan_flag = 0;
10     uint32_t comp_src = 0x0, comp_dst = 0x0, vlan_tag = 0;
11     mac_user_interface_type axi_rd, axi_w, axi_rd2;
12     bool isIp = false;
13     axi_w.last = 0;
14     axi_w.strb = 0xFF;
15
16     principal: while(1){
17     #pragma HLS LOOP_TRIPCOUNT min=1 max=1 avg=1
18
19         input.read(axi_rd2);
20         input.read(axi_rd);
21         vlan_tag = 0;
22         maco.range(63, 48) = 0;/*mac destino*/
23         maco.range(0,47) = axi_rd2.data.range(0,47);
24         macd.range(63, 48) = 0;/*mac origen*/
25         macd.range(0,15) = axi_rd2.data.range(48,63);
26         macd.range(47, 16) = axi_rd.data.range(31, 0);
27
28         if(axi_rd.data.range(39, 32) == 0x81 && axi_rd.data.range(47, 40) == 0x00){
29             input.read(axi_rd); // tipo | IHL | tam | id ( o tipo de nuevo)
30             while(axi_rd.data.range(7,0) == 0x81 && axi_rd.data.range(15, 8) == 0x00 &&
                   ↪ axi_rd.data.range(39, 32) == 0x81 && axi_rd.data.range(47, 40) == 0x00){
31             #pragma HLS LOOP_TRIPCOUNT min=0 max=1 avg=1
32                 input.read(axi_rd);
33             }
34             if(axi_rd.data.range(7,0) == 0x81 && axi_rd.data.range(15, 8) == 0x00)
35                 vlan_tag = 0;
36             else
37                 vlan_tag = 1;
38         }
39         if(vlan_tag == 0)
40             isIp = axi_rd.data.range(39, 32) == 0x08 && axi_rd.data.range(47,40) ==0x00;
41         else
42             isIp = axi_rd.data.range(7,0) == 0x08 && axi_rd.data.range(15, 8) == 0x00;
43         if(isIp){
44             cont_ip++;
45             axi_w.data= maco; //enviado -> source_mac
46             output.write(axi_w);
47             axi_w.data= macd; //enviado -> destination_mac
48             output.write(axi_w);
49             input.read(axi_rd2);
50
51             if(vlan_flag & 1){ //mirar si se desalinearon los campos por el tag de vlan
52                 parse_unaligned_data(input, output, comp_output, axi_rd, axi_w, axi_rd2);
53             }
54             else{
55                 parse_aligned_data(input, output, comp_output, axi_rd, axi_w, axi_rd2);
56             }
57         }
58         while (axi_rd.last==0){ //seguir leyendo de la interfaz si no ha llegado la senal
               ↪ de last
59             #pragma HLS LOOP_TRIPCOUNT min=1 max=1 avg=1
60             input.read(axi_rd);
61         }
62     }
63 }
```

Figura B.2: Código de la función principal *eth_ip_manager*

```
inline void parse_aligned_data(stream<mac_user_interface_type>&input,
    ↪ stream<mac_user_interface_type> &output, stream<uint32_t> &comp_output,
    ↪ mac_user_interface_type &axi_rd, mac_user_interface_type &axi_w,
      mac_user_interface_type &axi_rd2){
   uint8_t transport_prot;
   b0:{
   #pragma HLS PROTOCOL fixed
       size.range(7, 0) = axi_rd2.data.range(15, 8);
       size.range(15, 8) = axi_rd2.data.range(7, 0);
       size.range(31, 16) = 0;
       axi_w.data.range(63, 48) = 0;//tamano ip
       axi_w.data.range(47, 40) = axi_rd2.data.range(7, 0);
       axi_w.data.range(39, 32) = axi_rd2.data.range(15, 8);
       axi_w.data.range(31, 24) = axi_rd2.data.range(23, 16);//identificador ip
       axi_w.data.range(23, 16) = axi_rd2.data.range(31, 24);
       axi_w.data.range(15, 8) = axi_rd2.data.range(63, 56); //protocolo transporte
       axi_w.data.range(7,1) = 0;
       axi_w.data.bit(0) = axi_rd2.data.bit(37);
       transport_prot = axi_rd2.data.range(63, 56);
       output.write(axi_w);
       input.read(axi_rd); // checksum | ip o | ip d (mitad)
       axi_w.data.range(7,0) = axi_rd.data.range(47,40);
       axi_w.data.range(15, 8) = axi_rd.data.range(39, 32);
       axi_w.data.range(23, 16) = axi_rd.data.range(31, 24);
       axi_w.data.range(31, 24) = axi_rd.data.range(23, 16);
       comp_output.write(axi_w.data.range(31, 0));
       axi_w.data.range(63, 56) = axi_rd.data.range(55, 48);//guardar ip destino
       axi_w.data.range(55, 48) = axi_rd.data.range(63, 56);
   }
   b1:{
   #pragma HLS PROTOCOL fixed
       input.read(axi_rd); // ip d | opt ... o transport header
       axi_w.data.range(47, 40) = axi_rd.data.range(7, 0);
       axi_w.data.range(39, 32) = axi_rd.data.range(15, 8);
       output.write(axi_w); //enviar ips
   }
   b2:{
   #pragma HLS PROTOCOL fixed
   if(transport_prot == TCP_PROTO){
           axi_w.data.range(39, 32) = axi_rd.data.range(47, 40); //puerto destino
           axi_w.data.range(47, 40) = axi_rd.data.range(39, 32);
           axi_w.data.range(55, 48) = axi_rd.data.range(31, 24); //puerto origen
           axi_w.data.range(63, 56) = axi_rd.data.range(23, 16);
           input.read(axi_rd); // seq number | ack number | h_len | flags
           axi_w.data.range(31, 24) = axi_rd.data.range(63, 56); //flags
           axi_w.data.range(23, 17) = 0;//flag_FIN
           axi_w.data.bit(16) = axi_rd.data.bit(56);
           axi_w.data.range(15, 9) = 0;
           axi_w.data.bit(8) = axi_rd.data.bit(60);//flag_ACK_nulo
           axi_w.data.range(7, 0) = 0; //expired by flags
       }
       else if(transport_prot == UDP_PROTO){
           axi_w.data.range(39, 32) = axi_rd.data.range(47, 40);
           axi_w.data.range(47, 40) = axi_rd.data.range(39, 32);
           axi_w.data.range(55, 48) = axi_rd.data.range(31, 24);
           axi_w.data.range(63, 56) = axi_rd.data.range(23, 16);
           input.read(axi_rd);
       }
       else{
           input.read(axi_rd);
           axi_w.data = 0;
       }
       output.write(axi_w);
       comp_output.write(axi_w.data.range(31, 0));
   }
}
```

Figura B.3: Código de extracción de los datos alineados de *eth_ip_manager*

```
1  inline void parse_unaligned_data(hls::stream<mac_user_interface_type>&input,
       ↪ hls::stream<mac_user_interface_type> &output,
2      hls::stream<uint32_t> &comp_output, mac_user_interface_type &axi_rd,
           ↪ mac_user_interface_type &axi_w, mac_user_interface_type &axi_rd2){
3
4      uint8_t transport_prot;
5      ap_uint<32> ip_dest, size;
6      b5:{
7      #pragma HLS PROTOCOL fixed
8          input.read(axi_rd);//ip o | ip dest | source port
9          size.range(31, 16) = 0;
10         size.range(15, 8) = axi_rd2.data.range(39, 31);
11         size.range(7, 0) = axi_rd2.data.range(47, 40);
12         axi_w.data.range(63, 48) = 0;//tamano ip
13         axi_w.data.range(47, 40) = axi_rd2.data.range(39, 31);
14         axi_w.data.range(39, 32) = axi_rd2.data.range(47, 40);
15         axi_w.data.range(31, 24) = axi_rd2.data.range(55, 48); //identificador ip
16         axi_w.data.range(23, 16) = axi_rd2.data.range(63, 56);
17         axi_w.data.range(15, 8) = axi_rd2.data.range(23, 16); //protocolo transporte
18         transport_prot = axi_rd2.data.range(23, 16);
19         axi_w.data.range(7,1) = 0; //bandera
20         axi_w.data.bit(0) = axi_rd2.data.bit(29);
21         output.write(axi_w);
22         axi_w.data.range(31, 24) = axi_rd.data.range(15, 8);
23         axi_w.data.range(23, 16) = axi_rd.data.range(7, 0);
24         axi_w.data.range(7, 0) = axi_rd2.data.range(55,48);
25         axi_w.data.range(15, 8) = axi_rd2.data.range(63,56);
26         comp_output.write(axi_w.data.range(31, 0));//mandar al comparador la ip origen
27         axi_w.data.range(63, 56) = axi_rd.data.range(47, 40);//ip destino
28         axi_w.data.range(55, 48) = axi_rd.data.range(39, 32);
29         axi_w.data.range(47, 40) = axi_rd.data.range(31, 24);
30         axi_w.data.range(39, 32) = axi_rd.data.range(23, 16);
31         ip_dest = axi_w.data.range(63, 32);
32     }
33     output.write(axi_w); //escribir IPs
34     if(transport_prot == TCP_PROTO){
35         //puerto origen
36         axi_w.data.range(63, 47) = axi_rd.data.range(63, 47);
37         input.read(axi_rd);// dest port | seq number | ack
38         axi_w.data.range(47, 32) = axi_rd.data.range(15, 0); //puerto destino
39         input.read(axi_rd); // ack | | flags | window
40         axi_w.data.range(31, 24) = axi_rd.data.range(39, 32); //flags
41         axi_w.data.range(23, 17) = 0;
42         axi_w.data.bit(16) = axi_rd.data.bit(24); //FIN flag;
43         axi_w.data.range(15, 9) = 0;
44         axi_w.data.bit(8) = axi_rd.data.bit(28); //flag_ACK_nulo
45         axi_w.data.range(7, 0) = 0;//expired by flags
46     }
47     else if(transport_prot == UDP_PROTO){
48         axi_w.data.range(63, 47) = axi_rd.data.range(63, 47); //puerto destino
49         input.read(axi_rd);
50         axi_w.data.range(47, 32) = axi_rd.data.range(15, 0); //puerto origen
51         input.read(axi_rd);
52     }
53     else{
54         input.read(axi_rd);
55         input.read(axi_rd);
56         axi_w.data = 0;
57     }
58     comp_output.write(axi_w.data.range(31, 0));
59 }
```

Figura B.4: Código de extracción de los datos desalineados de *eth_ip_manager*