UNIVERSIDAD AUTÓNOMA DE MADRID

ESCUELA POLITÉCNICA SUPERIOR

TRABAJO DE FIN DE GRADO

CLASIFICACIÓN DE FLUJOS EN 10 GBPS ETHERNET MEDIANTE INTEL DPDK Y GPUS

Rafael Leira Osuna
(rafael.leira@estudiante.uam.es)

JUN 2013

CLASIFICACIÓN DE FLUJOS EN 10 GBPS ETHERNET MEDIANTE INTEL DPDK Y GPUS

AUTOR: Rafael Leira Osuna
TUTOR: Dr. Iván González Martínez

High Performance Computing and Networking
Dpto. de Ingeniería Informática
Escuela Politécnica Superior
Universidad Autónoma de Madrid
JUN 2013

Resumen

Resumen — Internet es una red en constante crecimiento. Para poder lidiar con ese crecimiento y su tráfico asociado surge una creciente necesidad de equipos cada vez más potentes, y por ende, más caros. Los proveedores de Internet (ISPs) al igual que diferentes empresas u organizaciones necesitan clasificar y controlar el tráfico de su red. Esto puede deberse a diferentes motivos, como aplicar diferentes políticas de calidad de servicio (QoS), filtrar el contenido de un enlace o prevenir ataques maliciosos contra una red y sus equipos. Por tanto la clasificación de tráfico se vuelve un tema crítico para algunas redes que no puede resolverse por métodos convencionales como la clasificación por puerto sino que se vuelve necesario un proceso mas complejo. No obstante, estas técnicas de clasificación deben ser suficientemente rápidas como para funcionar a tasa de línea. En este trabajo fin de grado se presenta un sistema que unifica todo el proceso de clasificación a nivel de flujo a alta velocidad. Dicho sistema es capaz de capturar tráfico a alta velocidad, construir flujos a tiempo real y finalmente realizar una clasificación de dichos flujos dentro de una GPU. Todo esto es posible a una tasa de 10 Gbps utilizando hardware convencional reduciendo así los costes a los que se ven sometidos las sondas de Deep Packet Inspection (DPI) comerciales. Los resultados obtenidos muestran un rendimiento influido por el número de flujos concurrentes y el número de protocolos o firmas buscados. A pesar de ello, el sistema de captura es capaz de soportar 10 Gbps en condiciones normales, de la misma forma que la GPU es capaz de clasificar, en el mejor de los casos, hasta 12 Mega Flujos por segundo.

Palabras Clave — Intel DPDK, DPI, GPU, CUDA, Procesamiento de Flujos, Construcción de Flujos, Quality of Service, Sonda de red

Abstract — Internet is an ever-growing network. The network equipment has to be improved and cheaper to cope with this growth and its associated traffic. Internet service providers (ISPs), companies, and different organizations require to control and classify traffic inside its network in order to apply different Quality of Service (QoS) policies for specific protocols, to filter network traffic inside a link or to prevent different kind of attacks. Then, such classifying systems are critical. However, classification by port does not provide good results, and it is necessary to apply other more complex techniques. Nevertheless these classification techniques have to be fast enough to work at line rates. This end-of-degree project presents a system that unifies the entire process involved in flow classification at high speed. It captures the traffic, builds flows from the received packets on the fly and finally classifies them inside a GPU. All the process is possible at 10 Gbps using commodity hardware reducing costs against commercial DPI probes.. The results show that the achieved performance is influenced by two factors: the number of protocols to find inside the GPU and the number of concurrent network flows in a link. Nevertheless, the probe can easily handle 10 Gbps in common situations and classify up to 12 Mega Flows per second on the fly using commodity hardware.

Index Terms — Intel DPDK, DPI, GPU, CUDA, Flow processing, Flow building, Quality of Service, Network Probe

Agradecimientos

Son muchas a las personas a las que debería agradecer y poco el espacio en donde escribir.
A Diego Hernando, por hacer esta plantilla en LaTeX, por su apoyo y su amistad a lo largo de los años en esta carrera.
A Iván González, por haber sido un magnífico y comprensivo tutor y por ayudarme siempre en todo lo que ha podido. Gracias a él, este Trabajo fin de grado ha podido completarse a tiempo.
A Jorge López, por apoyarme en la realización del paper y haber sido comprensivo con mi pobre inglés.
A Pedro Gómez y a Pedro María Santiago del Río, por el trabajo previo realizado y su ayuda a lo largo del trabajo.
Y en general al grupo de investigación HPCN y a todos sus miembros por haberme dejado participar como uno más y por el apoyo que en general he recibido por su parte a lo largo de este año y medio.
De la misma manera agradezco el apoyo a todas y cada una de las personas que he conocido a lo largo de la carrera, a mis amigos y a familiares y a Bea que de una u otra forma han estado conmigo a lo largo de esta carrera haciéndola más amena y finalmente llevándome a escribir este TFG y con él, terminar mi grado.

Índice general

Índice de cuadros

Índice de figuras

Glosario

DFA Deterministic finite automaton. 6, 30, 39

DPDK Data Plane Development Kit. 2, 3, 5, 13–17, 19, 21, 22, 28, 43

DPI Deep Packet Inspection. 2–4, 6, 7, 30, 36, 39

FPGA Field Programmable Gate Array. 5, 20, 36

GPU Unidad de Procesamiento Gráfico. 2–4, 6, 11, 18, 26, 28, 30–34, 43, 44

HPET High Precision Event Timer. 13, 14, 19, 21, 22

LCORE Núcleo lógico o Logical CORE. 10, 17, 40

NIC Network Interface Card. 17–19, 40

QoS Quality of Service. 1, 6, 30, 33, 38, 39

RTP Real Time Protocol. 36, 38, 39

1 Introducción

Internet está creciendo y cambiando día a día, con más y más sitios web, nuevos protocolos, usuarios y terminales [1]. Este aumento en la actividad supone un mayor número de conexiones entre distintos terminales. Cada una de estas conexiones está representada como un flujo de datos. Cada flujo tiene asociado un protocolo en particular por el cual dos terminales distintos son capaces de establecer una conexión y comunicarse.

No obstante, clasificar estos protocolos y flujos no es una tarea ni mucho menos trivial. Los proveedores de Internet (ISPs) así como diferentes empresas y organizaciones necesitan clasificar estos flujos a muy altas velocidades, ya sea para poder proporcionar Quality of Service (QoS) a sus usuarios, para prevenir ataques externos, o simplemente para prohibir ciertos protocolos dentro de una subred (Por ejemplo protocolos P2P).

Como es bien conocido, Internet está construido sobre los sólidos conceptos de IP (Internet Protocol). Sin embargo, la comunicación entre dos terminales suele venir dada por un flujo encapsulado en TCP o UDP. Dicha comunicación debe ser reensamblada a partir de paquetes independientes para que exista una comunicación real. A partir de este hecho surgen los primeros problemas a resolver: 1) la captura de paquetes a alta velocidad y 2) el reensamblado o construcción de flujos al vuelo.

Para realizar esto de forma eficiente, es necesario un sistema capaz de mantener un throughput elevado, constante y preferiblemente, aunque no indispensable, con una latencia reducida. Dado que es un problema común en las redes de hoy en día, ya existen diferentes dispositivos con la capacidad de realizar estos cometidos (por ejemplo [2–5]). En cambio, estos dispositivos tienen el problema de tratarse de productos, en general, excesivamente caros. Por ello y para poder competir contra ellos, el sistema propuesto funciona sobre un hardware de bajo coste.

A lo largo de este trabajo fin de grado se propone una solución para lidiar con cada uno de los problemas mencionados anteriormente. Finalmente se presentará un sistema capaz de realizar clasificación de flujos a tiempo real, integrando todo el proceso en una única sonda de bajo coste.

1.1 Motivación

El procesamiento de red en equipamiento de bajo coste es una necesidad real en el mercado [6]. Bajo esta premisa se inició el desarrollo de este trabajo fin de grado. Para lograr este objetivo, se planteó integrar diferentes módulos ya existentes en una o dos sondas de bajo coste. Cada uno de estos módulos independientes tenía la capacidad de procesar el tráfico a alta velocidad. Sin embargo, estos módulos se veían sometidos a algún tipo de limitación. Los módulos mencionados son:

1. Motor para la construcción de flujos: *FlowProcess*
2. Clasificador de flujos mediante Deep Packet Inspection (DPI) en una Unidad de Procesamiento Gráfico (GPU) [7]
3. Recepción de paquetes en GPU mediante zero-copy [6]
4. Recepción de paquetes a alta velocidad mediante Intel Data Plane Development Kit (DPDK) [8]

El módulo 1 y 2 disponían de un buen rendimiento. En cambio este par de módulos estaba diseñado para trabajar offline mediante el uso de un fichero pcap. Esto implica la necesidad de almacenar previamente una serie de datos y paquetes de grandes dimensiones. Por tanto, la integración con un módulo online permite mejorar el potencial de estos módulos. Además, el módulo numero dos, dado que trabaja a nivel de flujo, depende de los resultados del primer módulo. Esto, en condiciones normales, implicaría un segundo preprocesado de los datos de manera offline.

El tercer módulo se trata de un sistema de recepción que permite evitar la copia entre la tarjeta de red y una GPU. La idea inicial fue generar los flujos en una sonda, y retransmitir los resultados a una segunda sonda. La primera sonda se encargaría de la recepción de paquetes y reconstrucción de flujos mientras que la segunda clasificaría paquetes con zero-copy. Sin embargo, la arquitectura planteada a lo largo de este trabajo resultó ser eficiente para trabajar a 10Gbps sin pérdidas. Debido a esto, se decidió no incluir dicho módulo en la versión final [6].

El cuarto y último módulo está diseñado para capturar tráfico a alta velocidad sin perdidas.

Por tanto la motivación final del proyecto es construir una sonda muy eficiente, capaz de recibir tráfico de una red, preprocesarlo, construir flujos, clasificarlos y proporcionar un resultado a tiempo real. De esta manera se podrán sustituir sondas más caras por una más simple, barata y con capacidad para trabajar a tiempo real

1.2 Objetivos

Los objetivos del trabajo fin de grado son:

- Construir una sonda con Intel DPDK capaz de alcanzar 10 Gbps a máxima tasa sin pérdidas.
- Integrar la sonda con el módulo *FlowProcess*.
  - Una primera integración debe guardar los flujos reconstruidos a disco y retransmita su payload por una interfaz.
  - Una segunda integración tan solo debe comunicarse con el siguiente módulo sin retransmitir nada por la red.
- Integrar la sonda y el módulo *FlowProcess*, con el clasificador de flujo por DPI en GPU.

1.3 Fases de realización

El principal problema del trabajo recae en la complejidad que acarrea la unión entre diferentes módulos. Todos y cada uno de los módulos poseen su propio código, estilo y documentación. Esto complica aun más su interpretación y estudio. Para comprender totalmente los módulos hubo que indagar en el código fuente. De esta manera no solo se comprende la interfaz sino también el flujo de datos interno de los módulos. El objetivo del análisis fue asegurar una integración sencilla y con el menor número de imprevistos posible.

El primer paso consistió en leer una gran parte de la documentación de Intel DPDK. Dicha documentación era amplia y extensa. Esta documentación proporcionaba un manual de instalación, la API y un conjunto de directrices de programación. Las directrices se encontraban orientadas a obtener el mayor rendimiento posible de la máquina en la que se ejecutase. Algunas de estas directrices no son triviales, pues incluyen usos de caché a bajo nivel y predicciones de saltos. Una vez comprendido el módulo Intel DPDK se construyó una interfaz de programación para facilitar la integración entre dicho módulo y el resto de módulos. De esta manera se pretende facilitar futuras integraciones con diferentes módulos.

El módulo *FlowProcess* no estaba diseñado para trabajar a una tasa de 10 Gbps. Esto implicó realizar diversos cambios al módulo existente así como solventar los contratiempos inesperados que aparecieron. Además, el módulo trajo consigo la necesidad de implementar un módulo de marcado de paquetes (*timestamping*) que se comentará más adelante.

Finalmente, el módulo de clasificación de GPU. Para realizar la integración hubo que construir un módulo intermedio para servir de buffer entre la GPU y el Host. Como añadido se implementaron pequeñas mejoras con respecto al módulo original.

1.4 Estructura del documento

En el capítulo 2 se realizará una exposición del *Estado del Arte*, de cara a obtener una visión global de los trabajos e investigaciones actuales dentro del área de la clasificación en GPU. También se hará hincapié en las razones a que han llevado a elegir DPI como método de clasificación, así como sus ventajas y desventajas.

En el capítulo 3 se realizará un análisis de las características de nuestra sonda, desglosándolas en requisitos funcionales y no funcionales así como en los distintos módulos de los que está compuesta la sonda.

En el capítulo 4 se realizará una exposición de los módulos implementados así como su rendimiento unitario y las pruebas unitarias realizadas. El capítulo se divide en las siguientes secciones:

Sección 4.1: En esta sección se realizará una explicación mas detallada del módulo de captura, así como su arquitectura y los resultados obtenidos.

Sección 4.2: En esta sección se mostrará el problema de la toma de tiempos. Se expondrán y valorarán los métodos conocidos y se propondrá un método aproximado. A su vez, se explicará el módulo construido, sus funciones y el rendimiento ofrecido por cada una de ellas.

Sección 4.3: En esta sección se realizará una explicación mas detallada del módulo de construcción de flujos, así como su arquitectura y los resultados obtenidos. A su vez se explicarán los cambios realizados para hacer posible la integración y versiones alternativas construidas.

Sección 4.4: En esta sección se realizará una exposición del módulo de buffering de paquetes. Se mostrará su utilidad, así como los beneficios aportados a la sonda final.

Sección 4.5: En esta sección se realizará una explicación mas detallada del módulo de clasificación de flujos, así como su arquitectura y los resultados obtenidos. A su vez se explicarán los cambios realizados frente al módulo original.

En el capítulo 5 se mostrará la integración final. Junto a ella, se mostrarán las pruebas, los resultados y su correspondiente análisis. De igual modo, se explicará la complejidad en la realización de pruebas para sondas.

Finalmente, en el capítulo 6 se mostrarán tanto las conclusiones del proyecto como diversas propuestas para futuros trabajos.

2
Estado del Arte

2.1 Introducción

Tal y como se mencionó en el capítulo anterior la realización de una sonda clasificadora está sometida a una serie de problemas. Tradicionalmente, uno de los problemas fundamentales en redes ha sido obtener el máximo rendimiento de una tarjeta de red. Para solucionar este problema había que recurrir a la construcción o modificación de un driver y en algunos casos extremos a un dispositivo hardware programable (FPGA) [9–14]. Gracias a la aparición y apertura de Intel DPDK, estos problemas han sido simplificados en gran medida [8, 15].

La construcción de flujos no es tampoco un problema sencillo. Esto se debe a sus requerimientos de velocidad, y memoria. Ha habido diferentes aproximaciones para obtener un rendimiento a muy alta velocidad [10]. El módulo de creación de flujos utilizado en este trabajo fin de grado es una modificación del módulo utilizado en [16].

El problema de la clasificación de flujos es bien conocido y popular. Por ello hay multitud de investigaciones, algoritmos y dispositivos sobre los que clasificar flujos o paquetes. A continuación mencionaremos algunos de estos clasificadores así como sus beneficios e inconvenientes.

2.2 Tipos de algoritmos de clasificación

Para la clasificación de flujos hay diferentes métodos. Los métodos mas comunes son:

- Clasificación por puerto
- Deep Packet Inspection (DPI) o Inspección Profunda de Paquetes.
- Clasificación estadística.

Por ello existen diversos trabajos y estudios comparando ventajas e inconvenientes proporcionadas por cada uno de los métodos de clasificación previamente mencionados [17–27]. Tras sopesar cada uno de los métodos, en este trabajo se ha elegido utilizar DPI como método de clasificación, partiendo del trabajo realizado por [7, 25–27]. Para lograrlo se ha utilizado un Deterministic finite automaton (DFA) [25] y una implementación en una GPU de una manera similar, aunque optimizada y mejorada, a gregex [26]. Con estas ideas en mente, el trabajo relacionado con la GPU se basará fundamentalmente en [7].

Sin embargo DPI como método de clasificación también tiene sus propios inconvenientes. Hay una gran cantidad de investigaciones que asumen que los costes de DPI son muy elevados y buscan diversas alternativas como pueden ser las soluciones estadísticas [17–22]. De la misma forma existe una gran cantidad de trabajos que estudian diferentes formas de reducir los costes de DPI mediante diferentes técnicas, ideas y dispositivos [23–31]. En algunos casos estas ideas han sido llevadas a la práctica e implementadas en programas reales [27, 32, 33]. A pesar de ello, hay que reconocer que no todos estos programas son capaces de funcionar a la velocidad que deseamos alcanzar en este trabajo: 10 Gbps.

No obstante, los costes computacionales de DPI no tienen por que ser tan elevados como cabría esperar [22]. Si los costes se comparan con otras técnicas como la clasificación por puerto, que si bien es rápida, su fiabilidad es muy baja prefiriendo utilizar DPI. Es importante hacer hincapié en los protocolos multimedia. Estos protocolos requieren políticas de QoS, y dado que no suelen utilizar un puerto estándar, hacen inviable la utilización de este método de clasificación. Si hablamos de detectar un atacante la clasificación por puerto tiene aun menos sentido.

Utilizar DFA para realizar DPI tiene sus propios beneficios y perjuicios. Es bien sabido que las tablas DFA crecen de forma cuadrática lo que limita seriamente el número de firmas o protocolos que puede almacenar cada una de estas tablas. Por el otro lado, DFA proporciona un gran rendimiento dentro de una GPU [26]. Para resolver la limitación del número de firmas impuesta por DFA, se construirán tantas tablas DFA como sean necesarias para poder almacenar las firmas solicitadas. De esta forma se permite almacenar cuantas firmas o protocolos se deseen detectar sin limitar al sistema. Otra ventaja de DFA es ella misma como mecanismo de DPI. DFA proporciona uno de los elementos más importantes en clasificación de trafico: la fiabilidad. Las tablas DFA se construyen a partir de un conjunto de expresiones regulares. Estas, poseen tanta flexibilidad y fiabilidad como se desee por cada protocolo o firma.

2.3 Sondas clasificadoras en el mercado

Tal y como se mencionó en la introducción, existen diversos dispositivos comerciales que actúan como sondas DPI [2–5]. Cada uno de estos dispositivos poseen ciertas características. Una característica a señalar es el nivel al que trabaja la sonda. Como bien se ha comentado anteriormente, nos vamos a centrar en la clasificación de tráfico a nivel de flujo. No obstante, la mayoría de estas sondas proporcionan clasificación desde la capa 4 (Capa de transporte) hasta la capa 7 (Capa de aplicación). Esto proporciona a los usuarios de las sondas información adicional del trafico de su red. Un ejemplo sería cual es la web más accedida, la aplicación móvil con acceso a Internet más utilizada o el protocolo más utilizado. De forma adicional, dichas sondas pueden realizar una acción con cualquiera de los datos obtenidos al cumplirse una condición. Otro dato a favor de las clasificadoras actuales es su capacidad de procesamiento. Mientras que este trabajo se centra en construir una sonda a 10 Gbps, algunas sondas comerciales aseguran ser capaces de clasificar tráfico a 20 Gbps sin perdidas.

A pesar de esto, no todo son ventajas. Las sondas mencionadas son extremadamente caras. Este hecho complica el acceso a las sondas a ciertas empresas o usuarios que las requieran. Por tanto realizar una sonda de bajo coste es primordial para habilitar el acceso a estos dispositivos tan necesarios y solicitados.

2.4 Conclusión

En el mundo de las sondas clasificadoras se observa una gran actividad. Partiendo desde las tarjetas de red, hasta la clasificación de flujos o paquetes, nos encontramos con diversas investigaciones y productos compitiendo por un mejor servicio y rendimiento. Entre tanta diversidad es difícil catalogar que es mejor o que es peor. La solución ideal no existe, depende de una serie de necesidades puntuales. Una empresa o usuario requiere una sonda capaz de realizar una clasificación de tráfico a partir de unos requisitos que le son necesarios. Algunos de estos pueden ser fiabilidad, rendimiento o coste. Obtener el mejor de todos no es sencillo y finalmente hay que decantarse por una elección. Esto se presenta como un punto a favor de la sonda propuesta, ya que es “personalizable”.

En el siguiente capítulo se mostrará un análisis de la sonda. De esta forma, se pretende mostrar los requisitos funcionales y no funcionales que se han tenido en cuenta en la construcción de la sonda planteada en este trabajo fin de grado.

3
Análisis del problema

3.1 Introducción

El objetivo de este capítulo es mostrar de una forma clara y explícita los requisitos de la sonda propuesta. Para realizar esto, se enumerará cada módulo de forma individual junto con una breve descripción de sus requisitos. A continuación se desglosarán, los requisitos funcionales de cada módulo.

Finalmente, el capítulo concluirá con los requisitos no funcionales comunes a toda la sonda y un breve resumen.

3.2 Análisis de requisitos

3.2.1 Descripción de los módulos

A continuación se enumeran los módulos junto con una breve descripción de los mismos:

- **Módulo de captura**: Este módulo actuará a modo de interfaz entre las tarjetas de red y el resto del programa. Por tanto este módulo es el responsable de la recepción y envío de paquetes a la red.
- **Módulo de Timestamping**: Este módulo será responsable de marcar el tiempo a los paquetes en el momento en que sean recibidos por una interfaz.
- **Módulo de construcción de flujos**: Este módulo recibirá paquetes y expulsará flujos. Para ello, debe mantenerlos en memoria. Dada una condición debe generar una salida con la información de un flujo determinado.
- **Módulo de buffering**: Este módulo se encargará de mantener un buffer de flujos. Dada las limitaciones del clasificador de tráfico es necesario para obtener el máximo rendimiento.
- **Módulo de clasificación**: Este módulo recibirá un conjunto de flujos y los clasificará en paralelo. Finalmente los resultados se almacenarán para una posterior validación offline.

3.2.2 Requisitos funcionales

A continuación se enumerarán los requisitos funcionales de cada uno de los módulos:

Módulo de captura

RF1: El módulo será altamente configurable mediante parámetros

RF1.1: Mediante parámetros se decidirá que logical cores (LCOREs) se pueden utilizar.

RF1.2: Mediante parámetros se decidirá que tarjetas y puertos se utilizarán.

RF1.3: Mediante parámetros se decidirán las conexiones y colas entre los LCOREs disponibles.

RF1.4: Mediante parámetros se decidirá el uso de memoria máximo así como el tamaño de las diferentes colas creadas.

RF1.5: Mediante parámetros se decidirá el trabajo a desempeñar por cada LCORE.

RF2: El módulo debe esta desacoplado del resto de módulos.

RF3: El módulo debe asegurar la máxima tasa permitida por la tarjeta.

RF4: El módulo debe permitir capturar paquetes desde cualquier interfaz.

RF5: El módulo debe permitir enviar paquetes por cualquier interfaz.

RF6: El módulo debe poder gestionar su propia memoria y colas de paquetes.

RF7: El módulo debe poder gestionar los diferentes hilos en los que se ejecute y las conexiones entre los mismos.

RF8: La funcionalidad del módulo debe estar recogida en forma de API para el uso del resto de módulos

Módulo de Timestamping

RF1: El módulo será capaz de marcar paquetes con el tiempo aproximado de su llegada

RF2: El módulo permitirá diferentes configuraciones de marcado. La configuración se determinará en el código.

RF2.1: Marcado muy preciso: Se utilizará el *HPET*.

RF2.2: Marcado de precisión media: Se utilizará la función *gettimeofday*.

RF2.3: Marcado aproximado en función del tamaño y la velocidad actual.

RF3: El módulo debe esta desacoplado del resto de módulos.

RF4: La funcionalidad del módulo debe estar recogida en forma de API para el uso del resto de módulos

Módulo de construcción de flujos

RF1: El módulo será capaz de construir flujos a partir de paquetes independientes.

RF2: Dada una condición, el módulo expirará un flujo y ejecutara una función definida por el usuario.

RF3: La condición de espiración estará dentro del código del módulo.

RF4: El módulo debe esta desacoplado del resto de módulos.

RF5: La funcionalidad del módulo debe estar recogida en forma de API para el uso del resto de módulos.

Módulo de buffering

RF1: El módulo será capaz de almacenar flujos en un buffer en memoria.

RF2: El módulo será capaz de mantener en memoria tantos buffers como sean precisos.

RF3: Tanto el tamaño como la cantidad de buffers serán configurados en la inicialización del módulo.

RF4: El modulo reutilizará los buffers una vez dejen de ser útiles.

RF5: El módulo será responsable de almacenar los resultados del buffer o realizar otra acción relacionada antes de reutilizarlos.

RF6: La funcionalidad del módulo debe estar recogida en forma de API para el uso del resto de módulos.

Módulo de clasificación

RF1: El módulo será capaz de clasificar un buffer en memoria mediante GPU.

RF2: El módulo depositará los resultados de la clasificación en el mismo buffer una vez haya finalizado.

RF3: El módulo dispondrá de mecanismos para generar las tablas DFA a partir de un conjunto de listas de expresiones regulares

RF3.1: El módulo guardará las tablas DFA creadas en un fichero.

RF3.2: Si el conjunto de listas no se ha modificado, se cargará el fichero creado.

RF3.3: Si el conjunto de listas se ha modificado, se reconstruirá el fichero.

RF4: El modulo controlará la comunicación con la GPU así como todo lo relacionado con el dispositivo.

RF5: El módulo debe esta desacoplado del resto de módulos.

RF6: La funcionalidad del módulo debe estar recogida en forma de API para el uso del resto de módulos.

3.2.3 Requisitos no funcionales

A continuación se mostrarán los requisitos no funcionales. Estos requisitos son necesarios para el funcionamiento deseado de nuestra sonda. Sin embargo, no están relacionados de forma directa con la funcionalidad.

RNF1 *Rendimiento*: Una de las bases del trabajo es una sonda de alto rendimiento. Por tanto, el rendimiento de la aplicación es fundamental. La sonda debe poder operar a 10 Gbps al igual que cada uno de los módulos.

RNF2 *Flexibilidad*: Cada módulo debe ser lo mas independiente entre sí como sea posible. Esto permitirá su reutilización en futuros proyectos.

RNF3 *Coste*: Una de las bases del trabajo es una sonda de bajo coste. Por tanto, el sistema debe ser suficiente óptimo como para ejecutarse en un ordenador común.

RNF4 *Disponibilidad*: La sonda estará trabajando de forma continuada, por ello debe disponer de una alta disponibilidad.

RNF5 *Estabilidad*: La sonda estará trabajando de forma continuada, por ello debe disponer de una alta estabilidad.

RNF6 *Escalabilidad*: La sonda debe ser escalable para poder ejecutarse en diferentes ordenadores y exprimir el rendimiento al máximo.

RNF7 *Mantenibilidad*: El código debe estar bien organizado, limpio y documentado. Dado que habrá módulos que se reutilicen, la documentación es crucial para simplificar futuros trabajos de integración.

3.3 Resumen

En este capítulo hemos profundizado en los requisitos de la sonda. A partir de estos requisitos se construyeron, modificaron e integraron los diferentes módulos. En los siguientes capítulos veremos la construcción, funcionamiento y rendimiento de dichos módulos. A su vez, podremos apreciar las medidas tomadas para poder alcanzar los requisitos establecidos en este capítulo.

4
La arquitectura

4.1 Motor de captura: Intel DPDK

4.1.1 Introducción

Intel Data Plane Development Kit (DPDK) es un kit de desarrollo diseñado por Intel. Este kit de desarrollo ha sido construido con los siguientes propósitos [8]:

1. Obtener el máximo rendimiento (10 Gbps por tarjeta).
2. Realizar una implementación sencilla y rápida.
3. Explotar al máximo la arquitectura multicore [15].
4. Crear programas flexibles y escalables sin trabajo adicional.

Los propósitos de Intel DPDK coinciden con los requisitos previamente expuestos. Por este motivo se ha escogido Intel DPDK como método para implementar el motor o módulo de captura.

4.1.2 Instalación y configuración

Parte del módulo de captura consiste en un driver de linux proporcionado por Intel DPDK. Dicho driver debe ser configurado, compilado e instalado antes de poder empezar a programar aplicaciones sobre Intel DPDK. Los requisitos del driver no se encuentran disponibles en todas las distribuciones de Linux por defecto. A continuación se mencionan y explican su uso dentro del driver, así como sus implicaciones y utilidades a nivel de aplicación.

HPET (Opcional)

El dispositivo High Precision Event Timer (HPET) permite obtener con precisión el tiempo actual a un bajo coste para la CPU. Dicho dispositivo no es fundamental para el funcionamiento de nuestra aplicación, o para el driver de Intel DPDK. No obstante, Intel DPDK nos ofrece una API para utilizarlo si la máquina dispone de uno, ofreciéndonos la oportunidad de marcar paquetes a un bajo coste.

Para poder utilizar este dispositivo, el Kernel de Linux en el que se ejecute Intel DPDK debe tener instalado y cargado el driver de HPET. La recompilación del Kernel o la instalación de un módulo independiente depende de cada distribución. Suponiendo una distribución genérica, a continuación se muestran los pasos a seguir para realizar la instalación del módulo.

- El primer paso es situarse en la carpeta en la que se encuentre el Kernel. Habitualmente esta carpeta se encontrará en la dirección **/usr/src/linux**
- El siguiente paso es añadir el driver HPET al fichero .config del Kernel. Hay diferentes formas de hacerlo, una sencilla y visual es ejecutar **make menuconfig** (Fig. 4.1).

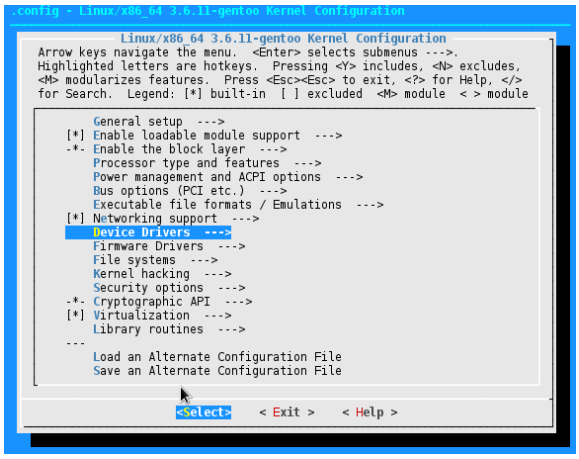

Figura 4.1: Configuración del kernel: make menuconfig

- Utilizando la herramienta *menuconfig*, se debe activar la casilla "*Device drivers/Character devices/HPET*" tal y como se muestra en la figura 4.2. La activación se puede hacer presionando la tecla "y" haciendo que el driver se compile en modo *built-in* y no genere un módulo aparte.

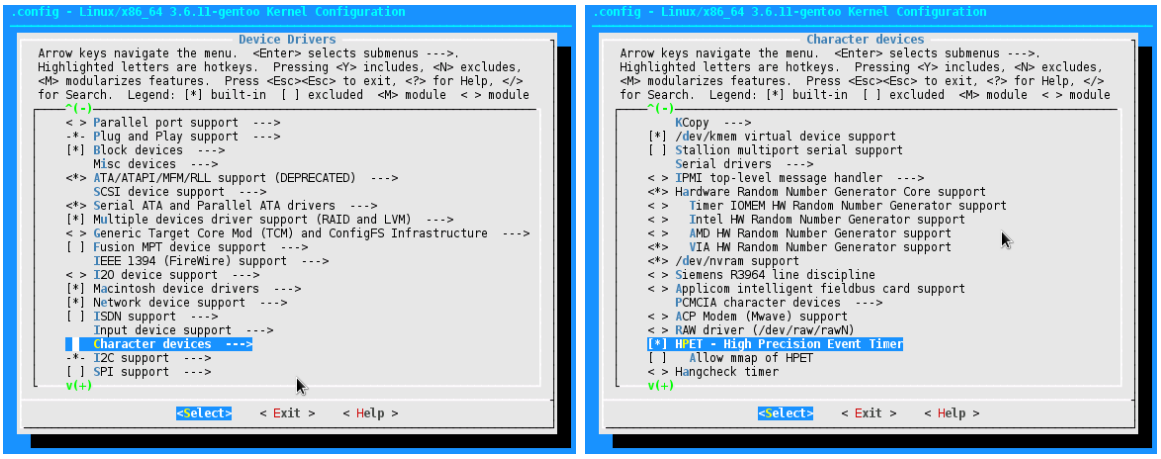

Figura 4.2: Compilación de HPET

- Una vez configurado, hay que guardar la configuración y salir del menu. Modificada la configuración se puede compilar usando el comando **make -jX**. Para que la compilación tarde menos, habrá que substituir la letra X por el número de cores lógicos disponibles más uno.

- Compilado el kernel, el siguiente paso es instalarlo. La instalación varia con la distribución, en la mayoría de los casos bastaría con ejecutar: **make install && make modules_install**.

- Si todo ha salido bien, al reiniciar debería aparecer el dispositivo *hpet* en la carpeta */dev/*

Hugepages

Comúnmente, los sistemas operativos utilizan páginas de 4096 bytes, es decir 4KB. Sin embargo, algo no demasiado conocido es que las CPU modernas pueden trabajar con diferentes tamaños de página. Las páginas mas grandes de 4KB son denominadas Hugepages. Los microprocesadores actuales ofrecen una bandera para indicar si soportan este tipo de páginas y su tamaño máximo. Dichas páginas pueden tener tamaño de 2MB o de 1GB.

Este tipo de paginación puede ofrecer grandes ventajas para aplicaciones de alto rendimiento:

- Son una amplia memoria contigua. Esto supone un menor número de fallos de caché.

- Es memoria del kernel mapeable a nivel de usuario. Esto supone que tanto drivers del kernel como una aplicación de usuario puede acceder a dicha memoria sin copias intermedias. Como se menciona en [6], la mayor parte de la perdida de eficiencia viene dado por copias de memoria entre los niveles de kernel y usuario.

- Por defecto es memoria no swapeable o paginable. Si la memoria reservada no puede volcarse a disco, se evitan problemas de rendimiento. Además, es posible asegurar un correcto funcionamiento de dispositivos externos como DMA.

Dentro del driver Intel DPDK, el manejo de las Hugepages es critico. Toda la memoria del driver se encuentra reservada en Hugepages, lo que incluye las colas de paquetes de las tarjetas de red. Para realizar esto, Intel DPDK tiene su propio gestor de memoria para las Hugepages. De cara a facilitar el trabajo al programador, Intel DPDK también ofrece una API para poder reservar y liberar memoria. Dicha API, ofrece ventajas respecto a sistemas convencionales de reserva de memoria como *malloc*. La mayor diferencia radica en que la API permite la alineación de memoria a la hora de la reserva. En cierta medida, esto puede evitar fallos de caché.

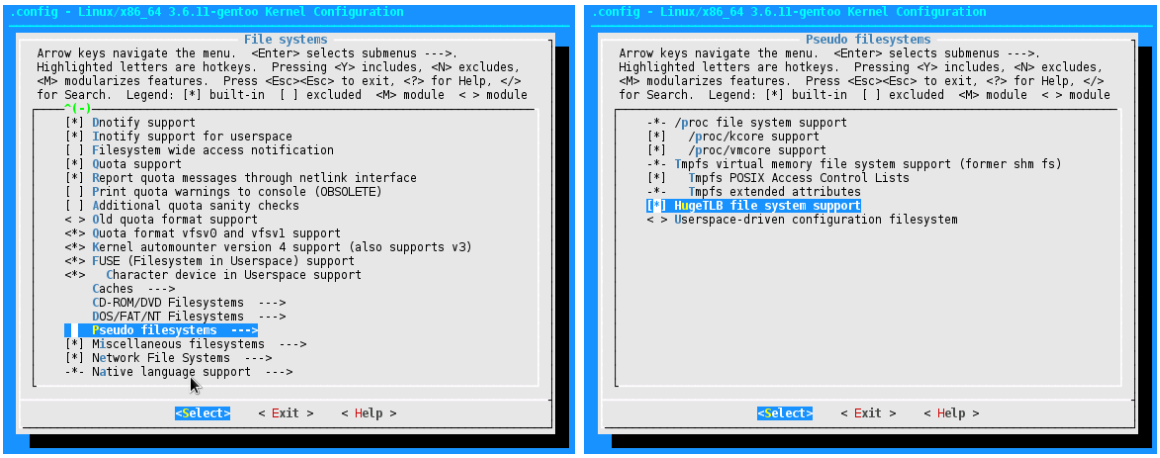

Figura 4.3: Compilación del HugeTLBfs driver

A pesar de la utilidad de las Hugepages, el driver puede no venir compilado por defecto en el Kernel de Linux de una distribución dada. De ser así, hay que recompilar nuevamente el Kernel si deseamos utilizar Intel DPDK. Para realizar la compilación habrá que seguir unos pasos similares a la sección anterior. En este caso deberemos activar la opción del kernel "*File systems/Pseudo filesystems/HugeTLB file system suport*" tal y como se muestra en la figura 4.3.

Una vez compilado el driver, las Hugepages deben ser configuradas. Uno de los beneficios de las Hugepages consiste en una enorme cantidad de memoria contigua en memoria. Sin embargo, la única forma de asegurar que dos o más páginas sean contiguas entre sí es reservándolas en el arranque. Para realizar esto, basta con modificar el GRUB e indicárselo al Kernel mediante los parámetros *default_hugepagesz*, *hugepagesz* y *hugepages*. Estos parámetros definen el tamaño por defecto, el tamaño actual y en número de páginas que se reservarán respectivamente.

Para la realización de la sonda, se han escogido 8 páginas de 1GB. En la figura 4.4 se muestra un ejemplo de la configuración para un GRUB 2 de un Fedora 17.

```
### BEGIN /etc/grub.d/10_linux ###
menuentry 'Fedora (3.8.4-102.fc17.x86_64)' --class fedora --class gnu-linux --class gnu --class os $menuentry_id_option 'gnulinux-sim
        load_video
        set gfxpayload=keep
        insmod gzio
        insmod part_gpt
        insmod ext2
        set root='hd0,gpt2'
        echo 'Loading Fedora (3.8.4-102.fc17.x86_64)'
        linux   /vmlinuz-3.8.4-102.fc17.x86_64 root=/dev/mapper/vg_azufre-lv_root ro default_hugepagesz=1G hugepagesz=1G hugepages=8
        echo 'Loading initial ramdisk ...'
        initrd /initramfs-3.8.4-102.fc17.x86_64.img
}
```

Figura 4.4: Configuración del grub: Hugepages

Una vez configurada la memoria, hay que montarla si se desea usar a nivel de usuario. El montaje puede realizarse mediante una entrada en el fstab o mediante el siguiente comando: "*mount -t hugetlbfs none /mnt/huge*". Es importante remarcar que la utilización de Hugepages puede limitar el número de instancias que se pueden crear de Intel DPDK.

El driver igb_uio de Intel DPDK

Una vez preparados los requisitos necesarios de Intel DPDK, se puede iniciar la instalación y compilación del driver en sí. La realización consta de los siguientes pasos:

1. Se debe descargar el driver oficial [8].
2. A continuación hay que descomprimir el paquete a una carpeta y entrar en ella.
3. Una vez dentro de la carpeta, se debe configurar el compilador (gcc o icc). Ejecutando *make config* se mostrarán las opciones disponibles. Seleccionar una y ejecutarla.
4. Configurado el compilador, basta con ejecutar *make* para que el driver se compile.
5. Para poder utilizar el driver hay que cargarlo ejecutando los siguientes comandos:
   (a) *modprobe uio*
   (b) *insmod <perfil de compilación seleccionado>/kmod/igb_uio.ko*

Una vez compilado el driver, es posible iniciar la construcción del módulo de captura.

4.1.3 Flujo de datos

Para construir un sistema escalable e independiente, se ha elegido una arquitectura de ejemplo ofrecida por Intel DPDK. Dicha arquitectura proporciona una alta flexibilidad, personalización, y en general los requisitos necesarios para el módulo de captura. Esta arquitectura de ejemplo se denomina *load balancer*.

A partir de aquí la idea es sencilla: Dividir entre recepción, proceso y envío de paquetes. Esto no solo permite una mayor independencia para otros módulos sino que también proporciona mecanismos sencillos para evaluar el rendimiento de forma independiente. Para realizarlo, se ha encapsulado cada componente en un logical core (LCORE).

A partir de aquí, denominaremos a los núcleos con la siguiente nomenclatura:

I/O RX Núcleo encargado de la recepción de paquetes.

Worker Núcleo encargado del procesamiento de paquetes.

I/O TX Núcleo encargado del envío de paquetes.

Cada núcleo se encuentra conectado al núcleo de la siguiente etapa mediante anillos (Colas circulares). De una forma similar, los núcleos *I/O RX* y *I/O TX* se encuentran conectados a las Interfaces de red (NICs) mediante anillos.

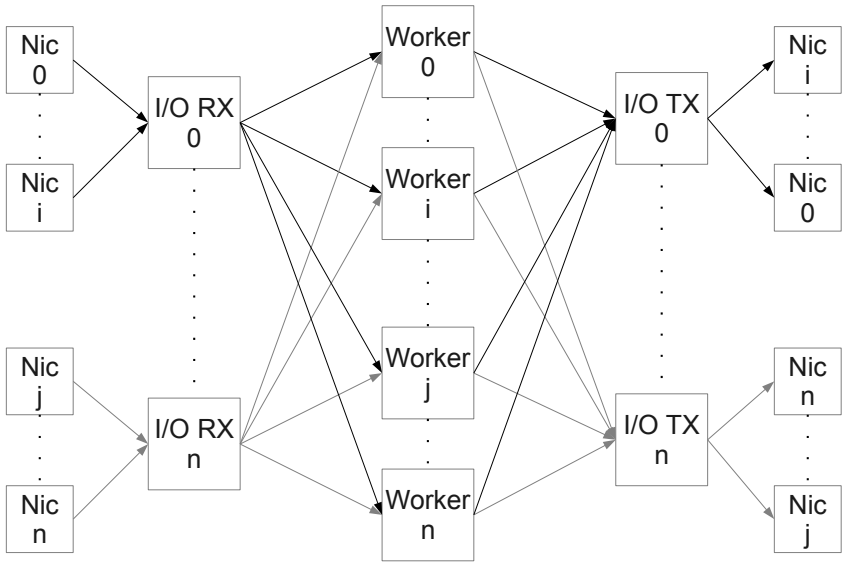

Figura 4.5: Flujo de datos y arquitectura del motor de captura

Tal y como se observa en la figura 4.5, la arquitectura escogida permite variar con facilidad el número de núcleos, colas y NICs. De esta forma es posible cumplir con los requisitos planteados en el capítulo 3.

Otra ventaja de la arquitectura escogida es su simplicidad en la codificación. Cada núcleo posee un código sencillo (Fig. 4.6), radicando la complejidad en la función Work() propia de cada núcleo. En el caso de los núcleos *I/O RX* y *I/O TX*, esta función está vacía. El único trabajo realizado por esos núcleos consiste en la lectura desde red. La complejidad de estos núcleos radica en la copia entre anillos. Dicha copia debe ser extremadamente rápida y eficiente.

Como muestra la figura 4.5, los núcleos *I/O RX* y *I/O TX* pueden elegir un núcleo de destino. Si suponemos que cada núcleo *Worker* es idéntico al resto, la paralización del trabajo se realiza de forma instantánea. Esta paralelización es realizada con tan solo un parámetro en la ejecución, que se transformará en otro núcleo *Worker*. En el caso concreto de nuestra sonda, esto podría significar tener diferentes instancias de clasificadores GPU, cada una utilizando una GPU diferente. De igual manera puede verse como diferentes instancias de un constructor de flujos u otro componente.

```
while true do
  for all R_i ⊂ anillosConectados do
    CP ← obtenerConjuntoDePaquetes(R_i)
    for all P_i ⊂ CP do
      P_j ← trabajar(P_i)
      R_j ← seleccionarAnilloDeDestino(P_j)
      encolarEnAnillo(P_j, R_j)
    end for
  end for
end while
```

Figura 4.6: Algoritmo genérico

No obstante, la división entre núcleos no es trivial. Cada núcleo posee su propia memoria asociada obligando a un conjunto de flujos a caer en siempre en un mismo núcleo. Para realizar esto, hay diferentes formas. La más sencilla radica en dividir el tráfico utilizando un byte de la dirección IP o el puerto TCP/UDP. Aunque parezca una buena solución, la división depende de los flujos de entrada. Esto puede generar casos en los que el reparto no sea equitativo entre los Workers. Por ello, aunque es un reparto sencillo y rápido, debe de estudiarse bien que byte se va a utilizar como divisor dada una subred concreta. En algunos casos, es posible que resulte necesario buscar otro algoritmo de balanceo de carga entre diversos núcleos.

A pesar de las grandes ventajas y flexibilidad que aporta la arquitectura escogida, solo son necesarios 2 núcleos para alcanzar los 10 Gbps sin pérdidas. Si queremos reenviar el tráfico o enviar los flujos, requeriremos un núcleo más y al menos dos NICs (Fig. 4.11). Adicionalmente, se puede estimar el número de núcleos necesario si deseásemos trabajar a mayor velocidad (Fig. 4.7).

$$Nucleos\ necesarios = \frac{20}{Velocidad\ en\ Gbps}$$

Figura 4.7: Cálculo de núcleos necesarios

4.1.4 Ejecución, pruebas unitarias y resultados

La ejecución

La alta configurabilidad de este módulo supone una gran cantidad de parámetros a la hora de la ejecución. Intel DPDK obliga de forma explícita el uso de los siguientes parámetros para todas aquellas aplicaciones que lo utilicen:

-c COREMASK: Máscara de lcores en hexadecimal.

-n NUM: Número de canales de memoria.

Estos parámetros permiten inicializar automáticamente los recursos proporcionados por Intel DPDK (Memoria, Hugepages, HPET...). Aunque la API proporciona otros parámetros útiles (Localización de las hugepages, deshabilitación del HPET...), solo los dos previos son necesarios para ejecutar. A partir de estos parámetros, es posible añadir nuevos parámetros a la aplicación o módulo realizado. Dada la alta configurabilidad de este módulo, es importante explicar los parámetros, que en tiempo de ejecución, dan forma a la arquitectura:

–**rx "(PORT, QUEUE, LCORE), ..."** Este parámetro representa una lista de NICs, colas y núcleos. Con este parámetro es posible definir que núcleo atenderá cada NIC.

–**tx "(PORT, LCORE), ..."** Este parámetro representa una lista de NICs y núcleos. Al igual que el parámetro anterior, este parámetro define que núcleo se encargará de la trasmisión de datos por una NIC.

–**w "LCORE, ..."** Este parámetro define la lista de los núcleos *Worker* que realizarán el trabajo pesado.

–**rsz "A, B, C, D"** en donde:

A : Tamaño en número de paquetes del anillo entre la NIC y el I/O RX.

B : Tamaño en número de paquetes del anillo entre el I/O RX y el Worker.

C : Tamaño en número de paquetes del anillo entre el Worker y el I/O TX.

D : Tamaño en número de paquetes del anillo entre la I/O TX y la NIC.

–**bsz "(A, B), (C, D), (E, F)"** en donde:

A : Tamaño del bloque de paquetes a leer desde una NIC por un I/O RX.

B : Tamaño del bloque de paquetes a escribir desde un I/O RX hasta un Worker.

C : Tamaño del bloque de paquetes a leer desde un I/O RX por un Workr.

D : Tamaño del bloque de paquetes a escribir desde el Worker hasta un I/O TX.

E : Tamaño del bloque de paquetes a leer desde un Worker por un I/O TX.

F : Tamaño del bloque de paquetes a escribir desde un I/O TX hasta una NIC.

–**pos-lb POS** Este parámetro permite escoger la posición del Byte de un paquete que será utilizado como divisor entre la lista de *Workers* proporcionada. Por defecto es 29.

Pruebas y resultados

Probar este módulo es posiblemente un problema en sí mismo, pues supone generar y retransmitir tráfico a alta velocidad. Para probarlo se han utilizado las herramientas disponibles en el laboratorio: Generador FPGA, y un reproductor de pcap software basado en packetshader [11]. De cara a mostrar la potencia del módulo construido se muestran los resultados del generador hardware (Fig. 4.8. Dicho generador construye paquetes de tamaño fijo de forma casi aleatoria y los envía a la máxima tasa posible.

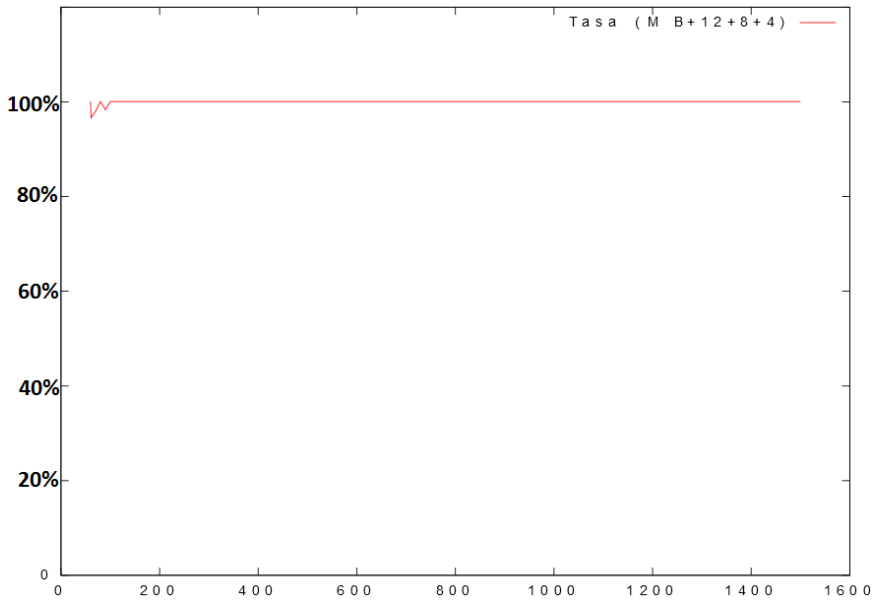

Figura 4.8: Flujo de paquetes en un enlace a tasa máxima

En la gráfica 4.8 es posible apreciar un rendimiento aceptable. No obstante, también se aprecia un pequeño descenso indicando pérdidas en la recepción para algunos tamaños de paquete. Tras investigar, se detectó que la FPGA estaba generando paquetes malformados para ciertos tamaños de paquete. No obstante, se ha considerado válido el diseño ya que para un tamaño de paquete mínimo (60 bytes) el módulo es capaz de capturar los 14.9 Millones de paquetes retransmitidos sin ninguna pérdida, siendo este, el caso más extremo posible en una red a 10 Gbps.

4.1.5 Conclusiones

La alta versatilidad del módulo construido hace posible un módulo capaz de adecuarse a diferentes circunstancias. Su independencia entre módulos, a su vez, facilita la capacidad de implementar y paralelizar trabajo con el menor efecto perjudicial posible a la captura de tráfico. La captura es capaz alcanza la mayor tasa de paquetes por segundo posible en una red a 10 Gbps, alcanzando el requisito de captura de tráfico sin pérdidas planteado en este Trabajo fin de grado.

A continuación se da paso al primer problema tras la captura de paquetes: El marcado de paquetes.

4.2 Marcado de paquetes: Timestamping

4.2.1 Introducción

El problema del marcado de paquetes surgió al poco de iniciarse las primeras pruebas. Para poder exportar un flujo es necesario que cumpla una condición previamente definida. De cara a construir un módulo basado en el sistema Netflow [34], se ha decidido utilizar la misma condición utilizada por dicho sistema: la espiración por caducidad. Sin embargo, para realizar una caducidad temporal es necesario el marcado temporal de cada uno de los paquetes. Aunque pueda parecer trivial, obtener el tiempo actual no lo es cuando si se requiere precisión y una baja latencia al obtener el valor [12]. Según las pruebas realizadas, las funciones habituales no son suficientemente precisas o el coste de llamarlas es demasiado grande para mantener el throughput. Para solucionar este problema se ha planteado este módulo.

4.2.2 Funciones disponibles

De cara a la realización de este módulo, se han tenido en cuenta diversas funciones y métodos para medir tiempo. Como medida de referencia se ha tomado el peor caso posible a 10 Gbps. Es decir, aproximadamente 14.88 millones de paquetes marcados por segundo, por tanto el tiempo disponible para procesar un único paquete y marcarlo no puede superar los **67.2ns**. A continuación se muestran las diferentes funciones junto con sus ventajas y desventajas.

Función time

La función necesaria para medir el tiempo debe proporcionarnos al menos una resolución en nanosegundos. En cambio la resolución de la función *time* solo alcanza los segundos. Por ello dicha función no podrá ser utilizada en este módulo.

Función gettimeofday

Tras medir el rendimiento de la función *gettimeofday*, se ha calculado un coste añadido por paquete de **31ns**. Sin embargo, esta función trabaja con microsegundos, y dado que se requiere una resolución en nanosegundos, no es suficientemente precisa. Su uso provocaría el marcado de diferentes paquetes con una única marca temporal. Esto puede causar problemas problemas de coherencia. Además su utilización supone el consumo del 46.13 % del tiempo de proceso disponible para un paquete.

Función rte_get_hpet_cycles

La función *rte_get_hpet_cycles* se encuentra contenida dentro del API de Intel DPDK. Esta función proporciona el número de ciclos que han transcurrido desde que se inició el dispositivo HPET. Conociendo la frecuencia del dispositivo, es posible calcular cuanto tiempo ha pasado desde una llamada previa a la función. Por tanto, si se relaciona una hora obtenida con *gettimeofday* a un valor de *rte_get_hpet_cycles*, es posible calcular con gran precisión la hora exacta.

El coste de esta función es inestable. Según las pruebas realizadas, la función sufre una dependencia de la versión de Kernel instalada, del driver HPET y la configuración del dispositivo. Para la versión del Kernel 3.8.x, el rendimiento fue **23.16ns** por paquete (un 34.5 % del tiempo disponible). Por otro lado, para la versión 3.5.x, presentó un rendimiento de **608.25ns** (un 905 % del tiempo disponible). Esta función no cumple los requisitos ni de rendimiento ni de estabilidad.

En adición, esta función está sometida a una dependencia con Intel DPDK así como al dispositivo y driver HPET. Este dispositivo puede no estar presente en algunos equipos. En cuanto al driver de HPET, solo está disponible a partir de la versión de Linux 2.6.24, limitando su uso a versiones posteriores del Kernel. Es importante conocer el beneficios y perjuicios aportados por la función. Por ello se permitirá la utilización de esta función dentro del módulo bajo los riesgos del usuario. Sin embargo, al no cumplir ciertos requisitos de rendimiento y estabilidad se ha evitado la utilización de la función dentro de la sonda final.

4.2.3 La función aproximada

Para resolver el problema del marcado de paquetes se ha optado por la construcción de una función aproximada. Supongamos un enlace con un tráfico de paquetes a máxima tasa. En este caso, entre paquete y paquete solo hay una espera conocida como *interframe gap*. (Fig. 4.9). Para este caso en particular el problema resulta sencillo de resolver: Dado el tamaño de un paquete y la velocidad de un enlace así como el tamaño del *interframe gap* utilizado, se puede calcular cuanto tiempo ha pasado desde el inicio del paquete hasta el inicio del siguiente paquete. Es una forma sencilla de calcular el tiempo siempre y cuando el enlace se mantenga a tasa constante.

Figura 4.9: Flujo de paquetes en un enlace a tasa máxima

En cambio, en un enlace normal, la distancia entre paquetes varía. Estos cambios vienen producidos por las aplicaciones y los usuarios que las utilizan. Por ello no es extraño observar dentro de un enlace tráfico en ráfagas. Un ejemplo de ráfaga podría ser la producida por apertura de un vídeo online. Los vídeos online envían en una ráfaga el comienzo de la reproducción para que el usuario pueda comenzar a ver el video lo antes posible. Elementos como este, desestabilizan el planteamiento anterior y por tanto hay que refinar un poco el algoritmo.

La mejora del algoritmo no es sencilla. A continuación se plantea un modelo para realizar una aproximación por bloques de paquetes.

El modelo

Supongamos que medimos la velocidad de un bloque de paquetes. Suponemos también que la velocidad con respecto al próximo bloque de paquetes, en media, no variará en gran medida con respecto de la anterior medición. Si las suposiciones se cumplen, se puede crear un modelo a partir de una relación entre la velocidad de transmisión y el tamaño de paquete. Por tanto, se puede seguir utilizando la primera idea suponiendo que trabajamos con un enlace de velocidad variable.

En la figura 4.10 se muestra como se realizaría dicha conversión. El primer enlace representa una conexión real con espacios variables entre paquetes. El segundo enlace es un modelo generado a partir de la suposición anterior: Un enlace con una velocidad menor y distancia equitativa entre paquetes.

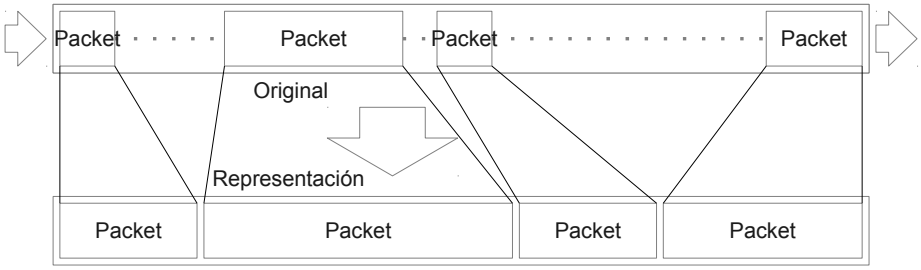

Figura 4.10: Flujo de paquetes en un entorno más realista

En la figura 4.10 se aprecia otro hecho: La perdida de precisión. Para el caso de la sonda planteada, la perdida de precisión no es un elemento crítico. Es más, se obtiene un beneficio con respecto a la función *gettimeofday*: Coherencia. Como se comentó previamente, la función *gettimeofday* tarda en modificar el valor de salida, produciendo un marcado de paquetes con timestamps idénticos.

Finalmente, hay que ajustar la velocidad virtual del enlace. Para hacerlo, la función aproximada realiza una serie de llamadas a una función del sistema por cada bloque de paquetes. Gracias a estas llamadas se puede ajustar la velocidad virtual del modelo y evitar grandes desviaciones con respecto a la hora real. No obstante, si el proceso de reajuste no se hace correctamente pueden suceder "'vueltas atrás en el tiempo" tal y como se explica en la sección 5.3.1.

4.2.4 Integración, pruebas unitarias y resultados

Una vez planteadas las funciones disponibles es posible describir el módulo realizado. Partiendo de la idea de un módulo independiente y flexible, la realización del módulo de Timestamping ha consistido en el agrupamiento de diferentes métodos de obtención de tiempo.

En la fase de inicialización, el módulo permite escoger una función temporal de sistema: *gettimeofday* o *rte_get_hpet_cycles*. A su vez, se inicializan las relaciones necesarias previamente mencionadas. Tras la inicialización, el módulo ofrece las siguientes funciones: *realtime_get(void)* y *realtime_getAprox(unsigned tam)*. La función *realtime_get(void)* permite obtener la hora exacta ofrecida por una función del sistema, mientras que la función *realtime_getAprox(unsigned tam)* ofrece una aproximación temporal en función del tamaño del último paquete recibido.

Para probar el rendimiento se ha medido el tiempo en marcar 16.2 Millones de paquetes por cada función. Esta prueba se ha realizado 25 veces por función, obtenido así una media. En el caso de la función *rte_get_hpet_cycles*, adicionalmente se han probado diferentes configuraciones y versiones del kernel. Finalmente, el coste temporal se ha dividido por paquete. A su vez se obtiene el tiempo consumido. Esta medida se representa como el porcentaje temporal de proceso consumido por paquete. En el caso particular de la función *gettimeofday*, 46.13 % significa que solo queda 53.87 % para construcción de flujos y resto de proceso de clasificación. En la tabla 4.1 se resumen los resultados obtenidos.

En las pruebas de validación tanto de *rte_get_hpet_cycles* como de *realtime_getAprox* no se apreció una gran desviación con respecto a *gettimeofday*.

Cuadro 4.1: Rendimiento del módulo Timestamping

Función	**Coste por paquete**	**Tiempo consumido**
gettimeofday	31ns	46.13 %
rte_get_hpet_cycles	23.16ns	34.5 %
	608.25ns	905 %
realtime_getAprox	12.20ns	18.15 %

4.2.5 Conclusiones

En esta sección se ha expuesto el módulo de toma de tiempo o *timestamping*. Este módulo permite diferentes modos de funcionamiento ofreciendo diferentes resultados y costes. Cada función del módulo tiene sus propias ventajas y desventajas y es necesario conocer ambas para poder utilizar de forma correcta este módulo. De la misma forma es necesario si se necesita obtener el máximo rendimiento de la ampliación en la se desee ejecutar.

Tal y como se comenta en la siguiente sección, la construcción de flujos requiere bastante cantidad de proceso. Por ello el coste del marcado de paquetes debe ser lo más pequeño posible. Dado que la perdida de precisión de *realtime_getAprox(unsigned tam)* no es muy grande y su coste temporal es muy pequeño, se ha escogido su utilización como método de marcado dentro de la sonda.

4.3 Construcción de flujos: FlowProcess

4.3.1 Introducción

La construcción de flujos representa un problema debido fundamentalmente a dos factores: La cantidad de memoria necesaria para almacenar los flujos concurrentes y los accesos a los registros de estos flujos. Para resolver estos problemas, se ha optado por realizar una modificación del módulo utilizado en [16,35]. A su vez, dicho módulo está basado en el sistema Cisco Netflow [34]. En esta sección se expondrá el funcionamiento del módulo así como las modificaciones, problemas encontrados y resultados obtenidos.

4.3.2 Implementación e Integración

La construcción de un módulo similar a Netflow, presenta una serie de problemas que deben ser resueltos. El primer problema a resolver es el almacenamiento de los flujos activos. Cada flujo que entra en la sonda, debe crear un registro. Dicho registro contiene la identificación del flujo así como estadísticas asociadas y el payload recibido hasta el momento. Almacenar estos registros en memoria es necesario si se quiere aplicar la política de espiración de Netflow. Esta política consiste en la caducidad de los flujos pasado un tiempo desde la recepción del último paquete. Tradicionalmente un flujo caduca a los 15 segundos como mínimo.

Para poder cumplir con dicha política, es necesario marcar temporalmente los paquetes tal y como se comentó en la sección 4.2. No obstante el marcado de paquetes no es el único problema a la hora de implementar esta política de espiración. El almacenamiento de flujos presenta un grave problema de memoria. Supongamos que cada registro ocupará 64 bytes de estadísticas e identificación junto con 256 bytes de payload. Esto hacen aproximadamente 320 bytes por flujo. Si disponemos de una memoria de 8 GBytes podremos almacenar a lo sumo 15 millones de flujos. Si suponemos a su vez el peor caso posible (todos los paquetes de tamaño mínimo, pertenecientes a un flujo distinto y máxima tasa), nos encontramos que 8 GBytes de memoria solo son capaces de almacenar apenas un segundo. Por tanto las sondas constructoras de flujo se ven seriamente limitadas por el número de flujos concurrentes en la red y la memoria disponible.

Flujo de datos

Para entender el flujo de datos utilizado junto con la integración con el módulo de captura, se muestra a continuación un resumen del módulo:

1. Un nuevo paquete entra en el constructor de flujos.
2. Se realiza una búsqueda del flujo dentro de una tabla Hash.
   (a) Si existe: Se obtiene un puntero para actualizar el registro.
   (b) Si no existe: Se crea y almacena un nuevo registro.
3. Se añade el payload (contenido útil) del paquete al registro del flujo. Si no hay espacio se copia hasta donde quepa.
4. Se actualizan los diferentes campos del registro (Número de paquetes, último paquete recibido...)
5. Se verifica que paquetes han alcanzado la condición de espiración y se exportan liberando los registros asociados.

Tal y como muestra la figura 4.11, la construcción de flujos se encuentra encapsulado en el núcleo *Worker* del módulo de captura. Como se ha mencionado en la sección 4.1, los núcleos *Worker* son los encargados de realizar los procesos pesados. Este núcleo, tiene también la obligación de marcar temporalmente los paquetes (sección 4.2) y comunicarse con la GPU (sección 4.5). Es por esto que cada acción realizada en este núcleo deba ser medida, valorada y optimizada para poder alcanzar 10 Gbps sin perdidas.

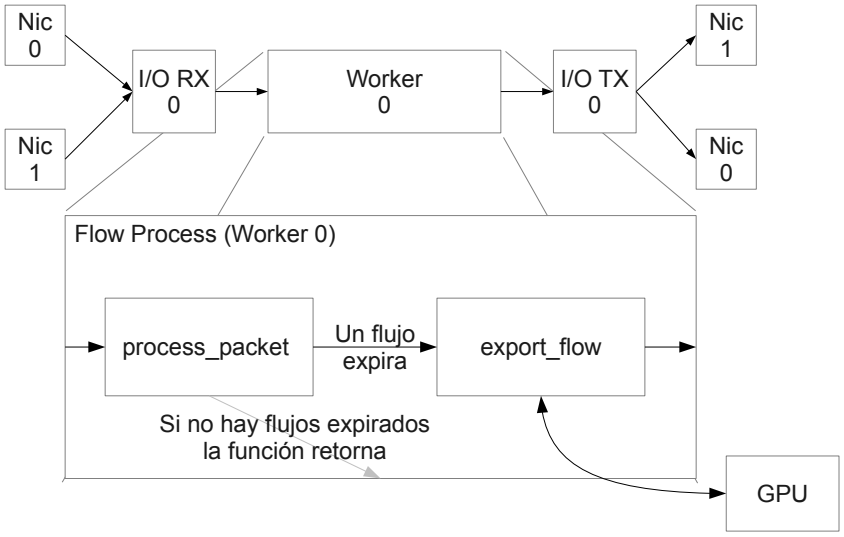

Figura 4.11: Exportando flujos a la GPU : Núcleo Worker

Estudio de rendimiento

Dado el flujo de datos del módulo, hay que recalcar dos costes computacionales asociados al tráfico recibido ya que influencian directamente en el rendimiento. El primero y más importante peso del módulo recae en el almacenamiento de flujos. Por cada paquete recibido se realiza una búsqueda en una tabla Hash, sin embargo, si dicho registro no existe se debe de crear y almacenar. El coste de crear un flujo nuevo, supone la petición a un pull de memoria junto con la inicialización de sus campos y su posterior almacenamiento en la tabla Hash.

Otro factor a tener en cuenta radica en las copias de memoria. Dado que el tamaño del buffer de payload es pequeño (256 bytes), es relativamente sencillo que uno o dos paquetes lo llenen. El número de copias, por tanto también está relacionado directamente con el número de flujos nuevos que entran al módulo y su tamaño. Un gran número de flujos diferentes implica por tanto un mayor número de copias y un menor rendimiento.

Debido a esto, queda claro que el rendimiento del módulo se encuentra influenciado por el número de flujos en vez de por el número de paquetes en sí. El rendimiento se este módulo se medirá en megaflujos por segundo (Mf/s).

4.3.3 Ejecución, pruebas y resultados

Como se ha mencionado al inicio de la sección, el módulo utilizado se ha basado en el trabajo realizado por [16, 35]. Dado que las modificaciones realizadas sobre el módulo han sido solo optimizaciones, el rendimiento del módulo se mantiene con respecto al original. Los resultados obtenidos muestran que el módulo es capaz de trabajar con al menos 2.25 millones de flujos por segundo a una tasa de 1.65 megapaquetes por segundo [35].

Adicionalmente, se han realizado nuevas pruebas de rendimiento a partir de una integración parcial con el módulo de captura y marcado. Dichas pruebas se comentan de forma más desarrollada junto con los resultados obtenidos en el capítulo 5.

4.3.4 Conclusiones

Tras una serie de pruebas se ha verificado el funcionamiento correcto del módulo junto con su integración. No obstante, este módulo resulta muy dependiente del tráfico de una red. La relación entre la velocidad de la red y la cantidad de flujos concurrentes es un tema aun en estudio.

Una vez asumidos los riesgos y dependencias de este módulo, puede darse paso al módulo de buffering junto con todo el proceso que conlleva en realidad exportar un paquete.

4.4 Buffering de flujos

4.4.1 Introducción

Las GPUs actuales requieren realizar una copia de memoria desde el Host hasta la GPU para poder trabajar. Dicha copia se realiza a través del puerto PCIe. Para obtener un buen rendimiento mediante PCIe se deben utilizar pocas y grandes copias de memoria en vez de muchas y pequeñas copias. Este suceso se produce por el coste que conlleva reprogramar sucesivamente una DMA, así como el propio funcionamiento del puerto PCIe. La construcción de un buffer de flujos resulta necesaria para poder minimizar los costes de transferencia a la GPU.

En esta sección se explica la interfaz creada ente la GPU y el constructor de flujos para procesar un bloque de paquetes.

4.4.2 Implementación

El módulo implementado ofrece una API por la cual requiere una inicialización y configuración. La API construida incluye una función por la cual el módulo es informado de un nuevo flujo exportado. A partir de este punto, se procede a almacenar el flujo en memoria dentro de un buffer. No obstante, debido a la implementación del módulo clasificador, no basta con un único buffer sino que es requerido un conjunto fijo de buffers (ver sección 4.5 para más información).

Para solucionar el problema de los múltiples buffers se plantea un anillo como estructura de datos de almacenamiento. En esta estructura se mantendrá un buffer activo en donde se escribirán los nuevos flujos entrantes. Al llenar el buffer activo, se informará a la GPU de que un nuevo buffer está listo. En ese momento el buffer activo será marcado como ocupado, y el siguiente buffer del anillo se marcará como el nuevo buffer activo. Simultáneamente, la GPU informará al terminar de utilizar un buffer. Por el funcionamiento del módulo de clasificación, la GPU copia los resultados de la clasificación en el buffer que contenía los flujos originales. Es por tanto, tarea de este módulo, utilizar los resultados antes de marcar el buffer como libre y reutilizarlo. En la sonda realizada, el módulo guarda en disco los resultados para una posterior verificación. Sin embargo, es importante tener en cuenta que se podrían realizar diferentes acciones, como pipes o la propia red, para informar a un dispositivo acerca del tráfico del enlace. (Fig. 4.12).

A pesar de todo, puede sufrirse un degradamiento del rendimiento debido a la transferencia de memoria entre el Host y la GPU. Para resolver este problema, CUDA ofrece los conocidos *streams*. Un stream de CUDA, puede transferir memoria en paralelo mientras se ejecuta código tanto en el Host como en la GPU. Esto es posible gracias al dispositivo DMA, ofreciendo un mejor rendimiento al sistema. A pesar del buen rendimiento ofrecido, los CUDA streams poseen ciertos requisitos. El más importante radica en la memoria del Host. Los flujos retransmitidos deben encontrarse contiguos en memoria especial. Esta memoria es conocida como *page-locked memory* o *pinned memory*. Para obtener dicha memoria se ha optado por la utilización de las Hugepages ofrecidas por la API de Intel DPDK. Su utilización rompe el esquema de independencia entre módulos planteada inicialmente, sin embargo, se ha considerado que la utilización de dicha memoria supone un rendimiento y simplicidad superior a la memoria ofrecida por la API de CUDA cuyo uso también rompería la independencia modular.

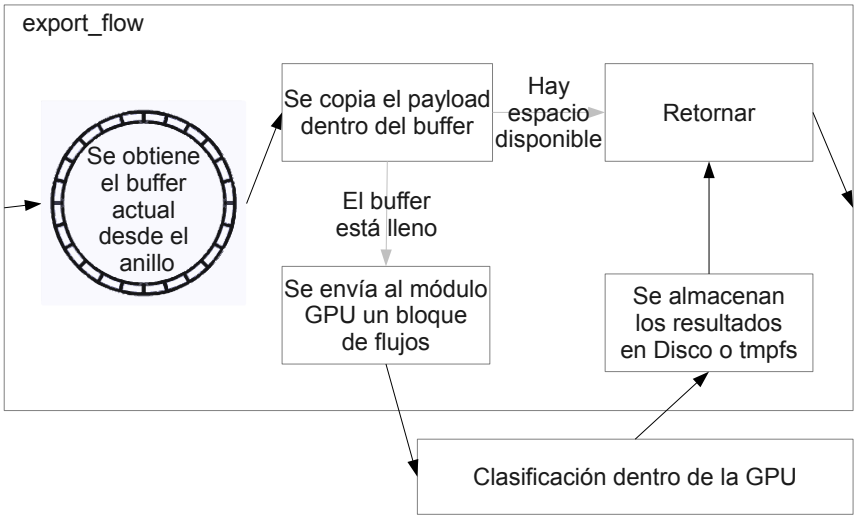

Figura 4.12: Exportando flujos a la GPU : Comunicación con la GPU

Independientemente de la fuente de la memoria utilizada, existe un problema. La forma mas eficiente de gestionar un buffer es mediante punteros a cada flujo exportado. No obstante, la condición de flujos contiguos en memoria implica que la implementación mencionada no sea viable. Por ello, se debe reservar gran cantidad de memoria por cada buffer y realizar numerosas copias de flujos al mismo. Aunque la implementación del módulo resulta sencilla, es importante tener en cuenta el coste que estas copias pueden conllevar al rendimiento global de la sonda.

4.4.3 Pruebas y resultados

Debido a la simplicidad del módulo y a su dependencia con el resto de módulos, solo se han realizado pruebas de validación, ya que medir el rendimiento del mismo de forma unitaria resulta difícil. Medir este rendimiento impondría, a su vez, un coste computacional adicional a la sonda. Tras la integración no ha detectado un coste computacional significativo frente al resto de módulos. El funcionamiento del módulo ha sido verificado.

4.4.4 Conclusiones

La construcción del modulo de buffering ha sido esencial en la integración entre los módulos de clasificación y construcción de flujos. Este hecho ha impedido en gran medida cumplir con los requisitos generales de independencias entre módulos. En cambio, las decisiones tomadas en la construcción del módulo han sido enfocadas de forma casi exclusiva a una optimización del rendimiento.

Una vez comprendido el módulo de buffering y su utilidad, es posible centrarse en el módulo clasificador y en su arquitectura. Ambos se presentan y explican en la siguiente sección.

4.5 Clasificación de flujos en GPU

4.5.1 Introducción

El método DPI consiste en un análisis de la carga útil proveniente de la capa de transporte de un único paquete o de un flujo. Este método es utilizado para determinar la naturaleza los paquetes o flujos para diversos objetivos, como la detección de intrusiones, detección de tráfico P2P, o la realización de diversas técnicas de QoS. El funcionamiento de DPI para la clasificación del tráfico TCP y UDP radica en la realización de una búsqueda para encontrar el protocolo al que pertenece un flujo determinado. Para realizar la búsqueda, se revisa si un conjunto de bytes del flujo cumplen una cierta expresión regular asociada a un protocolo en concreto. Si la expresión regular se cumple, se puede decir que el flujo pertenece al protocolo. Según las investigaciones realizadas en [24], se estima que a partir de 256 o más, la fiabilidad de una firma ni aumenta ni disminuye para la mayoría de los protocolos bien conocidos. Por esto, es posible decir que 256 bytes es suficiente para realizar DPI en una clasificación TCP/UDP. Tal y como se ha comentado en secciones anteriores, la sonda realizada trabajará internamente y en cada uno de los módulos diseñados, utilizando 256 bytes para almacenar el contenido de cada flujo.

Las expresiones regulares o firmas utilizadas en este Trabajo fin de Grado, han sido las obtenidas a partir de la aplicación L7-filter [32]. Adicionalmente han sido diseñadas nuevas expresiones regulares para ciertos protocolos. El módulo construido está basado en el trabajo realizado en [7], aunque ha recibido varias modificaciones y mejoras.

4.5.2 Arquitectura de la GPU: Búsqueda de un alto rendimiento

En la clasificación de flujos a tiempo real, la latencia no es muy importante. De hecho, el tiempo desde la espiración de un flujo hasta que este resulta clasificado, en la mayoría de los casos, no resulta importante. Sin embargo, al centrarse en redes de alta velocidad (10 Gbps o más), el rendimiento necesario fuerza una reducción de la latencia aumentando la importancia de la misma. Esto significa que el módulo construido debe ser capaz de soportar la espiración de flujos a dicha velocidad. La productividad o throughput, también debe ser constante e independiente a los tipos de flujos de entrada. La mayoría de aproximaciones de GPU están basadas en la minimización de la latencia. Un ejemplo son los CUDA streams, diseñados para dividir un trabajo en múltiples tareas pequeñas. Esta aproximación permite al sistema terminar antes que el trabajo original. Para poder optimizar la productividad, se ha decidido dividir el trabajo en diferentes tareas, uniéndolas en una arquitectura diseñada para maximizar la productividad: el pipeline. En los siguientes apartados se expondrán las diferentes tareas individuales que componen el pipeline.

Tabla de transición de estados

Antes de que la GPU pueda clasificar, cada firma debe ser preprocesada. Cada firma está compuesta por el nombre del protocolo y la correspondiente expresión regular. De una forma similar a gregex [26], un conjunto de expresiones regulares [36] debe ser transformada a su equivalente tabla DFA. Para realizar la transformación se ha utilizado un conjunto de herramientas para expresiones regulares [37]. Dicho conjunto de herramientas, proporciona diferentes fórmulas para la construcción de tablas DFA así como optimizaciones de las mismas.

Una vez construidas las tablas DFA, se puede construir la tabla de transición de estados. Cada fila de la tabla representa el estado actual mientras que cada columna representa el byte

de entrada actual. Cada celda ocupa 4 byte y se encuentra formada por los siguientes campos:

Primer byte: Este byte es utilizado para representar hasta 8 firmas encontradas. Cada bit i representa con un 1 si la firma iésima ha sido encontrada. Con un cero, si no.

Segundo, tercer y cuarto byte: Estos bytes son utilizados para representar hasta 224 posibles estados destino. En general es suficiente para un conjunto de 8 expresiones regulares complejas.

Procesamiento de flujos

Para poder procesar los flujos, se ha utilizado un CUDA kernel (ver [38] para más información sobre CUDA). Este kernel es el encargado de utilizar la tabla de transición de estados para clasificar los flujos. Para poder ejecutar el kernel, tanto los flujos como la tabla de transición debe encontrarse en la memoria de la GPU. No obstante, los flujos deben ser almacenados en columnas si se desea minimizar los fallos de caché y con ello, obtener el máximo rendimiento posible [26]. La transposición de flujos se realiza dentro de la GPU para maximizar el rendimiento del sistema [38].

Frente al módulo original [7], se han realizado modificaciones en el tratamiento de los hilos. En el módulo actual, cada hilo procesa simultáneamente 4 flujos, leyendo en una única iteración 4 byte (Fig. 4.13). Gracias a la presencia de las cachés L1 y L2 presentes en la arquitectura Fermi de NVIDIA, los accesos a memoria se convierten en transacciones de 128 byte. Esto significa, un único acceso a memoria y por ende, una única transacción.

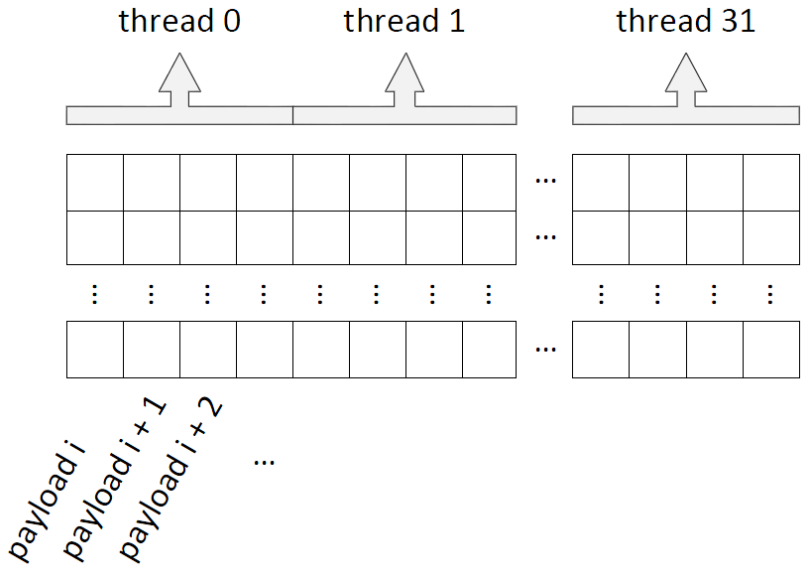

Figura 4.13: Matriz de flujos

Por cada byte de entrada procesado, un byte es escrito. Cada byte escrito contiene las firmas encontradas en su correspondiente byte de entrada. El iésimo bit del byte escrito, representa una única firma. Dado que esto causa un conjunto de 256 bytes por cada flujo, es necesario realizar una reducción de los datos para obtener el resultado final. La solución óptima requiere de otro kernel que aplique la operación OR a todos y cada uno de los bytes procesados. Esta reducción también supone una menor transferencia entre la GPU y el Host.

Mientras que el almacenamiento de firmas en un único byte aumenta el rendimiento, también limita el número de firmas que se pueden procesar a 8. Para solucionar el problema de una forma sencilla, se ha optado por ejecutar el kernel de clasificación tantas veces como conjuntos de 8 firmas tengamos. Dichos kernels, tienen la capacidad de ejecutarse en paralelo. No obstante, también comparten la caché y esto penaliza el rendimiento global.

El pipeline

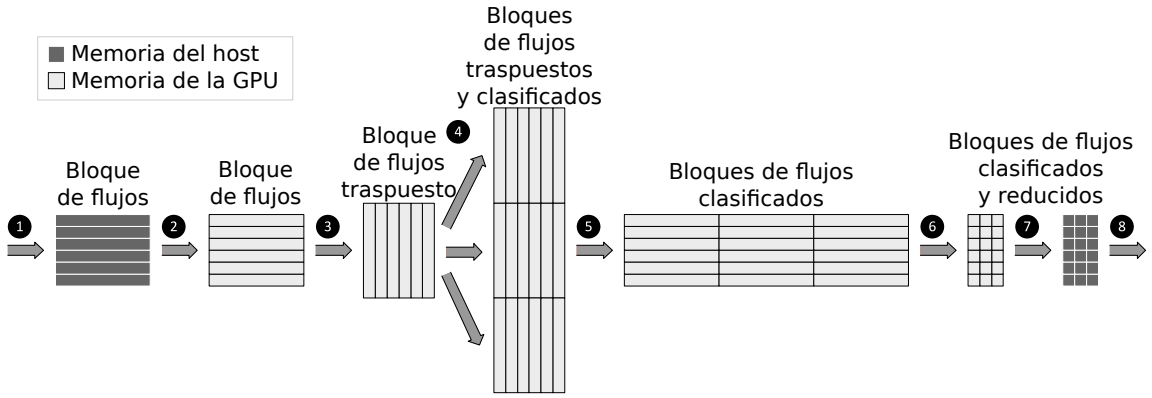

Figura 4.14: Pipeline dentro de la GPU

Para obtener una productividad elevada y constante [37], se ha propuesto la realización de un pipeline. El pipeline consiste en la división del proceso en diferentes pasos, siendo los típicos:

1. Trabajo computacional en el host.
2. Transferencia de datos desde el host hacia el dispositivo GPU.
3. Trabajo computacional en el dispositivo GPU (ejecución del kernel).
4. Transferencia de datos desde el dispositivo GPU hacia el host.

Cada paso es independiente. Esto significa 4 tareas que pueden ejecutarse en paralelo. Sin embargo, este paralelismo solo puede alcanzarse si la tarjeta NVIDIA utilizada tiene una "compute capability" igual o superior a 2.0. No obstante, son necesarios la ejecución de más de un kernel dentro de la GPU para realizar las tareas previamente mencionadas. Por ello, el pipeline previamente mencionado, debe contener más de una tarea de proceso. Este nuevo pipeline esta formado por los siguientes pasos (Fig. 4.14):

1. **Buffering de flujos:** Como se comento en la sección 4.4, los flujos deben ser almacenados dentro de un buffer. Este buffer debe ser grande, contiguo y en memoria no paginable. Tal y como se explica en diferentes trabajos [26], esto proporciona el mejor rendimiento durante la copia a la GPU. Este paso es realizado por la CPU cada vez que expira un paquete.
2. **Transferencia a la GPU:** Un bloque de flujos es copiado a la memoria de la GPU a la espera de ser procesado.
3. **Transposición:** Los flujos deben ser traspuestos en columnas para obtener el mejor rendimiento posible.

4. **Clasificación:** Por cada conjunto de 8 firmas, se lanzará un kernel independiente. Cada kernel trabajará sobre su propia tabla de transiciones pero todos ellos compartirán los flujos previamente traspuestos. Los resultados se guardarán contiguos en memoria de la GPU para facilitar el siguiente paso.

5. **Transposición:** Los datos deben ser traspuestos de nuevo para facilitar la lectura a la CPU. Este trabajo es realizado por otra instancia del kernel de transposición.

6. **Reducción:** Para poder minimizar las transferencias de memoria, los datos deben ser reducidos. Para ello se aplica una reducción en paralelo utilizando la función OR. Los resultandos de cada conjunto de firmas se almacenan de forma contigua, formando por tanto un resultado de tantos bytes como conjuntos de firmas y flujos haya.

7. **Transferencia al Host:** Los datos simplificados son enviados a la memoria del Host.

8. **Operar con los resultados:** Cuando los resultados están dentro del Host, pueden empezar a tomarse decisiones. Como se comentó en la sección 4.4, el módulo de buffering será el encargado de realizar las operaciones sobre los resultados. Algunos ejemplos posibles son: guardar estadísticas a disco, alertar de una intrusión mediante un paquete, o programar una regla de QoS para beneficiar o censurar un flujo determinado.

4.5.3 Ejecución, pruebas unitarias y resultados

Dentro del pipeline, el cuarto paso implica ejecutar varios kernels dentro de la GPU. Para la arquitectura Fermi de NVIDIA, solo se permiten la ejecución de 16 kernels simultáneamente. Esto significa un máximo de 88 firmas antes de que se produzca una serialización del kernel. Adicionalmente, implica más fallos de caché entre los kernels, lo que finalmente se convierte en una pérdida de rendimiento tal y como muestra la figura 4.15.

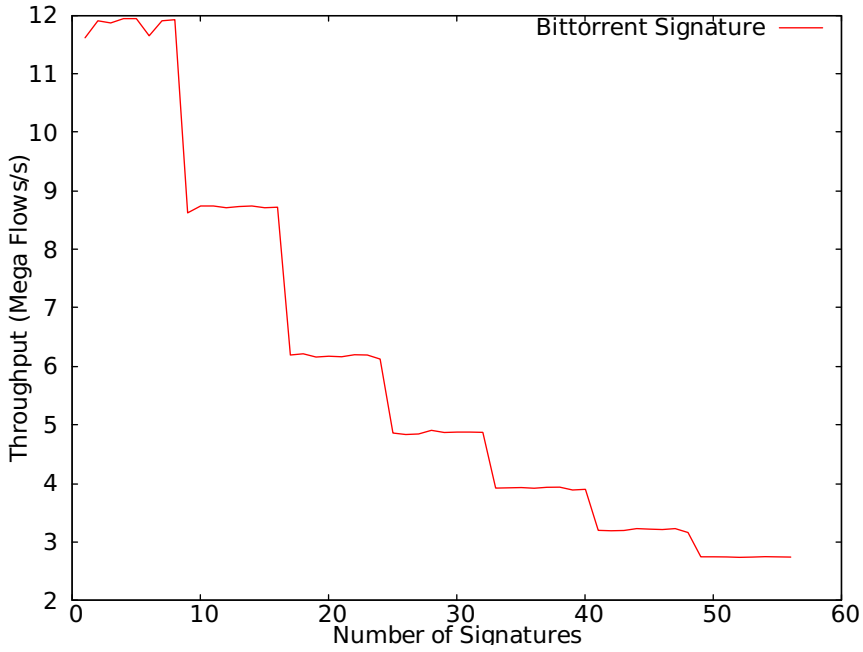

Figura 4.15: GPU performance

En la tabla 4.2, se puede observar el rendimiento para un número fijo de firmas. Para obtener los datos, se han ejecutado cada kernel de forma serie e individualmente. A pesar de ser mediciones útiles para detectar cuellos de botella en la implementación no incluye las penalizaciones que sufre la caché al ser compartida por el resto de los kernels simultáneamente. Esto se refleja en una importante diferencia entre el kernel con menos rendimiento y el rendimiento global.

Cuadro 4.2: Rendimiento de la GPU por cada CUDA kernel (in Gbps)

Firmas	**Trans.**	**Trasp.**	**Clas.**	**Trasp.**	**Reduc.**	**Trans.**	**Global(Gbps/Mfps)**
8	**35.92**	90.84	51.40	98.27	64.83	260.42	24.41 / 11.9
16	**32.42**	84.46	47.35	53.69	34.49	169.84	17.86 / 8.7
24	33.17	72.67	43.28	36.51	**23.39**	124.01	12.54 / 6.1
32	31.38	64.57	39.56	27.80	**17.72**	100.81	9.99 / 4.8
40	29.76	58.74	27.08	22.45	**14.33**	85.38	7.98 / 3.9

Otro dato importante mostrado en la tabla 4.2 es el número de flujos procesados por segundo. Dado que un enlace a 10 Gbps soporta hasta 14.9 Millones de paquetes, la GPU podría procesar hasta un 79.87 % de los paquetes. No obstante, esto solo sucedería si cada paquete representase un único flujo, lo que es muy poco probable en un entorno real. Supongamos ahora otro caso extraño. Si cada flujo estuviese formado por 4 paquetes de tamaño mínimo. Esto supondría una tasa de 3,7 Millones de flujos por segundo. Dicha tasa es fácilmente soportable por este módulo incluso para 40 firmas.

Todos resultados mostrados anteriormente han sido realizados mediante pruebas offline. De cara a obtener resultados neutrales e independientes de las firmas, se ha utilizado una única firma compleja: *bittorrent*. Debido a la naturaleza del clasificador, la complejidad de las firmas solo varía el tamaño de la tabla de transiciones. Esto, a lo sumo, puede causar una variación del número de fallos de la caché debido a la distancia entre dos elementos de la tabla. Por tanto, los datos mostrados no deberían variar de forma notable si se utilizase un conjunto cualquiera de firmas cuya complejidad no sea excesivamente superior a un conjunto de firmas *bittorrent*.

4.5.4 Conclusiones

El módulo de clasificación proporciona un rendimiento razonable para trabajar a 10 Gbps. Dado que está construido para funcionar dentro de una GPU, obtiene una serie de ventajas respecto a otros módulos. Las GPUs pueden encontrarse en el mercado por muy bajo precio (unos cientos de euros) pudiendo, en algunos casos, llegar a ser más baratas que una única CPU, facilitando así un mayor acceso a las GPUs.

Una vez se han descrito todos y cada uno de los módulos, es posible comprender la sonda en su conjunto así como la integración entre cada uno de los distintos módulos. En el siguiente capítulo se da paso a un resumen de cada módulo, su integración, junto con las diferentes pruebas y resultados de rendimiento y validación de las integraciones parciales.

5 Integración, pruebas y resultados globales

5.1 Introducción

La integración entre diferentes módulos es siempre un problema y suele presentar multitud de imprevistos. Del mismo modo, realizar las pruebas de una sonda tan compleja, es en sí un problema. La virtualización tanto de un entorno real como un entorno extremo requiere equipamiento específico para dicho cometido. Para poder verificar la integración y medir el rendimiento se ha optado por la realización de varias pruebas parciales. Esto permite verificar con una mayor facilidad el funcionamiento y detectar cuellos de botella. Tras las verificaciones parciales, es posible realizar una verificación de todo el sistema.

En este capítulo se presenta un resumen de la integración, las pruebas realizadas y el equipamiento utilizado. También se explican los diferentes problemas encontrados y las soluciones propuestas a los mismos.

5.2 Equipamiento de pruebas

La realización de las pruebas así como el rendimiento obtenido depende en gran parte del sistema en el que es probado. Dado que el proyecto realizado es una sonda de red, depende a su vez de los equipos suministradores de tráfico además de la sonda en sí misma.

5.2.1 Hardware utilizado

Para realizar las pruebas se han utilizado 2 generadores de tráfico diferentes, uno software y uno hardware. El generador de tráfico software es capaz de reproducir una traza en memoria RAM a alta velocidad. Por otro lado, el generador hardware es capaz de enviar tráfico de tamaño variable a máxima tasa. La diferencia entre ambos radica en su construcción. Mientras que el generador software es capaz de enviar trazas grandes, está limitado por el propio software. Esto provoca pequeñas anomalías en el envío de tráfico, como el tráfico a ráfagas. Además el control de la velocidad a la que se trasmite es muy poco preciso, ya que solo es posible variar el tiempo transcurrido entre una ráfaga y otra. Dicho generador fue construido en base a packetshader [11].

Por otro lado, el generador hardware disponible está implementado en una NetFPGA. Esto permite un mayor control sobre la velocidad de los paquetes enviados. Sin embargo, las capacidades de la FPGA limitan seriamente el tamaño de las trazas que pueden ser enviadas por este dispositivo debido a la falta de memoria. El uso de este generador se encuentra limitando por tanto a la construcción de paquetes aleatorios de tamaño configurable y velocidad configurable. Este dispositivo solo ha sido usado para probar el módulo de captura de tráfico (tal y como se ha explicado en la sección 4.1.4).

En la tabla 5.1 se muestran los datos técnicos de los equipos utilizados.

Cuadro 5.1: Equipamiento de pruebas

	Sonda construida	**Generador Software**	**Generador Hardware**
Placa base	Supermicro X9DRG-HF	-	-
CPU	Intel Xeon E5-2609 10M L3 Cache, 2.40GHz	Intel Xeon E5620 12M L3 Cache, 2.40GHz	Intel Core i7 920 8M L3 Cache, 2.67GHz
RAM	16 GB	12 GB	6 GB
GPU	Tesla M2090 5375 MB RAM	-	-
Tarjeta de red	Intel Corporation 82599EB 10-Gigabit	Intel Corporation 82599EB 10-Gigabit	NetFPGA-10G Xilinx Virtex-5 VTX240T
S.O.	Fedora 17	Ubuntu 10.04 LTS	Ubuntu 12.04.1 LTS
Linux	3.5.3-1	2.6.34	3.2.0-24

5.2.2 Tráfico utilizado

Una vez obtenido un reproductor de trazas pcap, es importante escoger las trazas a utilizar. Esto es fundamental para poder verificar correctamente la sonda construida y enfrentarla a redes reales y casos extremos. Para este cometido se han utilizado principalmente las siguientes trazas:

Traza de CAIDA: Esta traza ha sido obtenida a partir de la organización CAIDA [39]. La organización tiene en posesión diversas trazas de entornos reales, lo que permite simular el comportamiento de la sonda en un entorno real. Sin embargo, al provenir de en un entorno real, el contenido útil de los paquetes ha sido anonimizado. Esto incapacita la clasificación por DPI. La traza contiene 968744 flujos concurrentes a tasa de captura.

Traza Multimedia: Esta traza contiene mayoritariamente tráfico multimedia retrasmitido a través de Real Time Protocol (RTP). A pesar de ser una traza con un gran número de paquetes, la cantidad de flujos de la traza es extremadamente pequeña. La traza fue obtenida de un entorno real

Traza a medida: De cara a obtener una traza con un gran número de flujos y contenido útil, se ha construido una traza a medida. Utilizando el comando *tcpdump*, se ha capturado el tráfico de una subred. El tráfico de la subred ha sido creado mediante el uso de arañas web, diferentes descargas torrent, scripts de *bash* y gestores de correo. De esta forma se obtiene una traza con diversos flujos y protocolos. A tasa de captura la traza contiene 3044 flujos concurrentes.

5.3 Pruebas parciales

5.3.1 Integración hasta *Flowprocess*

Las primeras pruebas parciales comenzaron con la integración de los 3 primeros módulos: Captura, marcado y construcción de flujos. Los primeros resultados mostraron pequeños fallos tanto en el módulo de marcado como en el de construcción. Resolverlos no fue sencillo debido a la complejidad del sistema y dado que la construcción de flujos offline funcionaba sin ningún error.

El primer error encontrado fueron las "vueltas atrás" causadas por el módulo de marcado. En los reajustes temporales producidos por el módulo, hay una cierta probabilidad de causar una vuelta atrás en el tiempo. Cuando sucede, causa incoherencias en el orden de llegada de los paquetes. Dado que el sistema de construcción de flujos no estaba preparado para soportar estas anomalías, hubo que realizar pequeños cambios en ambos módulos para solventar el problema.

El segundo error encontrado fueron los "tamaños de cola". El módulo de captura sufría una falta de memoria al trabajar a máxima velocidad provocando pequeñas pérdidas de paquetes. Por otro lado, el módulo de construcción de flujos también necesitó ampliar el tamaño de listas y buffers para adecuarse al tráfico de las pruebas.

Una vez solventados los errores de integración, se inició una prueba de validación entre el modo *online* y *offline* del módulo *Flowprocess*. Los resultados mostraron pequeñas variaciones en el orden de espiración de los flujos. Esto es fácilmente explicable debido a las pequeñas variaciones sufridas durante el marcado, envío y recepción de los paquetes. Para verificar la equivalencia entre ambas salidas, basta con agrupar los flujos y ordenarlos por IP. Si en ambas salidas los flujos contienen el mismo número de paquetes las salidas son equivalentes. Tras realizar la prueba, ambos modos retornan salidas equivalentes.

Una vez la integración parcial fue validada, se comenzó a realizar las pruebas de rendimiento. Dichas pruebas revelaron buenos resultados de forma general pero presentaban inconvenientes para ciertos casos concretos. Si bien el módulo de captura, en condiciones normales, no pierde paquetes, el constructor de flujos no siempre es capaz de consumir todo el tráfico suministrado por el capturador. El mejor ejemplo es la utilización del generador hardware. Dado que este generador crea un flujo por paquete, la cantidad de flujos por segundo supera rápidamente la memoria disponible causando un error crítico y terminando el programa.

Por otro lado, los primeros resultados obtenidos mediante el generador software tampoco fueron ideales. Tras retransmitir las trazas anteriores a máxima tasa, el módulo de construcción de flujos puede presentar pérdidas de hasta el 9 % del tráfico. Dichas perdidas se encuentran sujetas a la traza utilizada, el número y tamaño de las ráfagas causadas por el generador y la configuración de los buffers y colas. No obstante estas perdidas solo se manifiestan bajo la presencia de ráfagas, una variación brusca en el número de flujos o una mala configuración. En la mayoría de los casos, basta con adecuar la configuración al trafico de entrada. Tras un reajuste en la configuración, el número de perdidas sufridas por este módulo es reducible a 0. Por tanto la integración parcial es capaz de construir flujos a 10 Gbps sin pérdidas en condiciones normales.

Un detalle obtenido a partir de las pruebas fue la cantidad de flujos emitida por el módulo. A partir de las trazas retransmitidas, la relación entre paquetes y el número de flujos es muy pequeña. Esto causa que a partir de una traza retransmitida a 10 Gbps se han obtenido tasas de espiración de 73.2 KiloFlujos por segundo. En el caso concreto de la traza de CAIDA [39], de obtuvo una tasa de espiración de 0.7 megaflujos por segundo. Ambos números son buenos ya que entran dentro de las capacidades de cómputo del módulo clasificador (sección 4.5).

5.3.2 Integración completa

Una vez se obtuvieron los resultados de la integración parcial, se inició la integración de los dos módulos restantes. Esta fue con diferencia la integración más sencilla pues no supuso ningún imprevisto durante el proceso. Durante las pruebas realizadas, las variaciones del rendimiento en el núcleo *Worker* con respecto a la versión sin clasificador fueron mínimas. Por ello, se ha decidido mantener todo el proceso de clasificación en un único núcleo dado que la clasificación en GPU no supone una sobrecarga tangible para el núcleo *Worker*.

El problema de los protocolos: RTP

Uno de los objetivos a cubrir es la construcción de una sonda capaz de trabajar en un entorno real. Como método de prueba, se ha propuesto retransmitir y clasificar al vuelo la traza multimedia. En un entorno real, los resultados serian enviados a un sistema de QoS que tomaría decisiones sobre los flujos del enlace.

La mayoría del tráfico multimedia se trasmite a través del protocolo RTP. Por tanto, para poder realizar QoS, es necesario clasificar correctamente dicho protocolo. Sin embargo, el protocolo RTP [40] no da demasiada información sobre el mismo (Fig. 5.1). Si se sigue el RFC al pie de la letra, el único campo estático del protocolo son los dos primeros bit de versión (10), siendo el resto de campos variables. Si la firma construida para RTP solo contemplase los dos bit de versión, el 25 % de los flujos serían clasificados de forma incorrecta como RTP (falsos positivos). Por el otro lado, no habría ningún falso negativo.

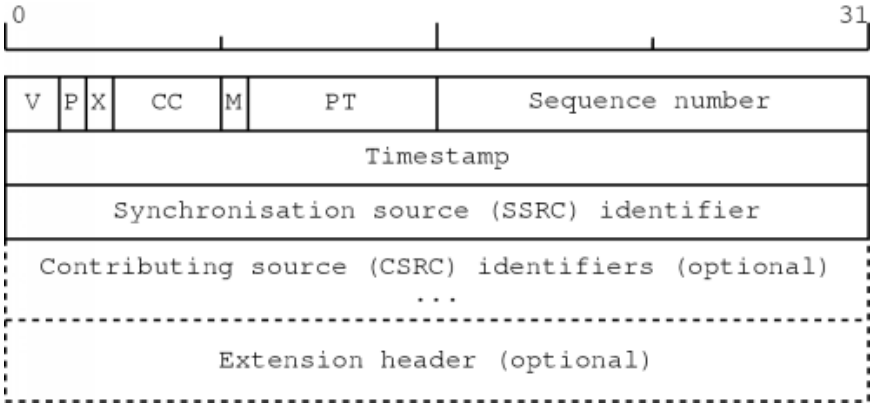

Figura 5.1: Cabecera del protocolo RTP [41]

No obstante, es posible plantear otros métodos de construcción firmas. El protocolo RTP posee un campo denominado *PT*. Este campo representa que tipo de contenido multimedia contiene el flujo RTP (Audio, vídeo...). Por tanto, este campo puede ser utilizado como método para identificar no solo un flujo RTP, sino también el contenido del mismo. En la figura 5.2 se muestra un ejemplo de firma RTP para un *PT* 0x21. Se ha comprobado que dicha firma es capaz de clasificar correctamente los flujos RTP de la traza multimedia con 0 falsos positivos y 0 falsos negativos.

$$\hat{}[\backslash x80\backslash x90\backslash xA0\backslash xB0]\backslash x21$$

Figura 5.2: Firma para el protocolo RTP y el flujo multimedia 0x21

Comúnmente audio y vídeo no tienen la misma prioridad dentro de un enlace. La clasificación de los tipos de flujos multimedia facilita la aplicación de técnicas de QoS para diferentes tipos de flujos multimedia. Con esta aproximación, los ISP podrían ofrecer mejores servicio multimedia a los usuarios a un bajo coste.

A pesar de todo, el problema de los protocolos queda en el aire. Como se ha comentado previamente, la utilización de tablas DFA y expresiones regulares permite la construcción de firmas de forma sencilla y tan precisas como se desee. Sin embargo, los protocolos binarios como RTP, presentan un problema a la hora de realizar DPI. Si estos protocolos no poseen uno o varios campos identificativos y suficientemente grandes pueden causar una gran cantidad de falsos positivos. Esto presenta un problema a tener en cuenta durante la construcción de las firmas.

No obstante, no se puede olvidar la existencia de diversos programas capaces de realizar DPI mediante expresiones regulares. Tras verificar las firmas utilizadas por el programa L7-filter [32] se ha concluido que la mayoría de esas expresiones regulares no son capaces de clasificar los protocolos de forma correcta. Esto obliga a la construcción de firmas propias si se desea obtener una fiabilidad óptima en la clasificación, junto con los problemas que conllevan.

5.4 Integración final: Una visión global

Una vez se ha realizado la integración de todos los módulos es posible construir una idea general del funcionamiento de la sonda. En la figura 5.3 se muestra la interconexión entre los módulos previamente expuestos. En dicha figura es posible observar todo el recorrido y proceso que debe realizar un paquete hasta que finalmente es clasificado. Aunque el sistema es capaz de alcanzar una gran tasa, es posible observar que la complejidad del sistema impone una serie de latencias insalvables.

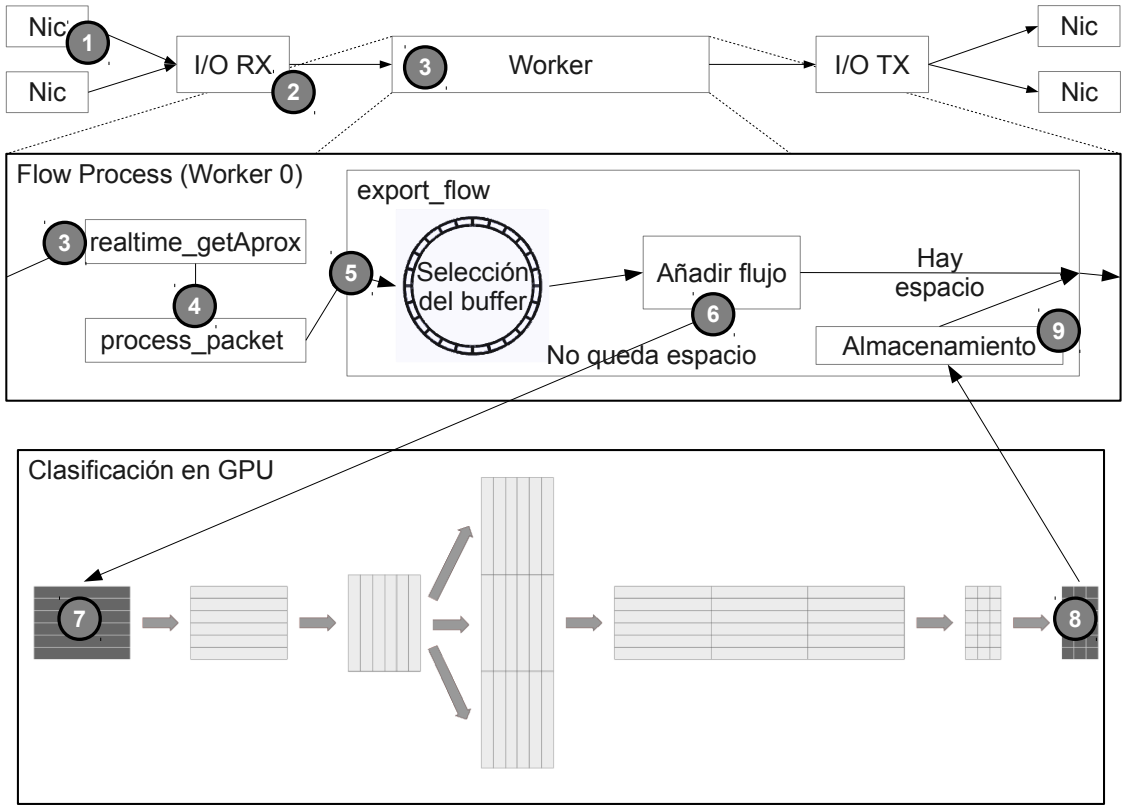

Figura 5.3: Integración final de la sonda

El flujo final de un paquete es el siguiente:

1. Un paquete nuevo entra en una NIC. La tarjeta lo copia su correspondiente anillo de recepción a la espera de que un LCORE lo coja.

2. El núcleo *I/O RX* obtiene un conjunto de paquetes a partir del anillo de la NIC. Acto seguido, encalo dicho bloque al anillo de entrada del núcelo *Worker*.

3. El núcleo *Worker* obtiene un conjunto de paquetes a partir de su anillo de entrada. Sin embargo, los procesa de forma individual. El primer paso realizado es el marcado temporal del paquete mediante el módulo de marcado.

4. Una vez el paquete ha sido marcado, es enviado al módulo de construcción de flujos mediante la función process_packet. Dentro de dicha función se creará un registro para identificar el flujo así como el contenido útil (payload) del paquete recibido.

5. Eventualmente el flujo construido alcanzará la condición de espiración. En ese momento se llamará a la función export_flow.

6. Una vez dentro, la función copiará el contenido util dentro del buffer activo. Eventualmente, dicho buffer se llenará y dará pie al siguiente paso.

7. El flujo es copiado a la memoria de la GPU. A partir de aquí, se iniciará todo el proceso necesario en la clasificación de un flujo. (ver sección 4.5). Para que el flujo termine el proceso, es necesario que otros 6 nuevos buffers de flujos entren en el módulo clasificador.

8. Una vez la clasificación ha terminado, los resultados son copiados al mismo buffer en el que se encontraban los datos originales.

9. Finalmente, el módulo de buffer realiza un volcado a disco de los resultados para su posterior análisis. La función retorna y el ciclo se repite.

5.5 Conclusiones

Utilizando los medios disponibles se han diseñado diferentes pruebas para cada módulo e integración parcial. Dada la complejidad, tanto del sistema como de las integraciones parciales, tomar medidas de rendimiento y verificar el funcionamiento es una tarea compleja. A pesar de todo, ha sido posible verificar el correcto funcionamiento de cada módulo y las pequeñas integraciones parciales entre módulos.

Finalmente y tras las pruebas realizadas, es posible asegurar que, bajo una buena configuración y una red a 10 Gbps común, es posible clasificar el tráfico a máxima tasa, sin pérdidas y al vuelo.

6
Conclusiones y trabajo futuro

En este trabajo fin de grado se ha construido una sonda para la clasificación de flujos en una red a 10 Gbps Ethernet a partir de un conjunto de módulos independientes.

El módulo de captura ha representado una pieza clave dentro del funcionamiento de la sonda. Intel DPDK ha permitido capturar trafico a alta velocidad sin pérdidas. No obstante, es necesario un correcto ajuste del tamaño de las colas y los bloques de paquetes para poder exprimir la potencia del módulo. Para los casos probados, la utilidad de Intel DPDK ha quedado probada. La eficiencia del módulo construido, la flexibilidad y la simplicidad hacen de Intel DPDK un kit de desarrollo muy recomendable para la realización de futuras sondas y para el tratamiento de trafico a alta velocidad.

Por otro lado, están los módulos de marcado de paquetes y construcción de flujos. Ambos módulos requieren un refinamiento ya que sus costes son elevados y representan en su conjunto el cuello de botella de la sonda. Estos limitan el número de flujos concurrentes que la sonda es capaz de procesar. Sin embargo, y gracias a una buena configuración, es posible mantener un rendimiento sin pérdidas de paquetes en normales.

Tanto el módulo de captura como el de clasificación han explotado los conceptos de la paralelización. En el caso de la GPU, ha vuelto a quedar demostrado la capacidad de procesamiento de este dispositivo. Las modificaciones del módulo de clasificación han aumentado en 10 Gbps el rendimiento del dispositivo con respecto al módulo original [7]. La capacidad de paralelización de la GPU, ha permitido la escalabilidad necesaria como para poder aumentar el número de firmas con respecto al módulo original [7].

La capacidad de procesamiento de la GPU alcanza los 24.4 Gbps. Puesto que cada flujo ocupa 256 bytes, esto es traducible a 12 megaflujos por segundo. Por otro lado, el máximo número de flujos posible en una red a 10 Gbps es de 14.88 megaflujos. En escenarios habituales, tanto 14 como 12 megaflujos son cantidades muy raramente alcanzadas. La traza de CAIDA [39] puede llegar a alcanzar hasta 1 megaflujo/s .Por tanto, en condiciones normales, la capacidad de clasificación de la GPU alcanza perfectamente el rendimiento necesario.

Es muy importante tener en cuenta el problema de los protocolos tratado en la sección 5.3.2. La fiabilidad de la sonda depende de las firmas construidas. Esto obliga a mantener las firmas actualizadas acorde con las últimas versiones de los protocolos que quieran clasificarse. Tampoco hay que olvidar las firmas de L7-filter [32], que según se han probado, dan muy malos resultados.

Una versión resumida de este trabajo fin de grado, ha sido aceptada en la conferencia SaCoNeT'2013 [42]. Se incluye en el apéndice A el artículo y la carta de aceptación.

6.1 Trabajo futuro

Como trabajo futuro se proponen las siguientes ideas:

- **Realizar un módulo de captura con procesamiento multicapa**. De la misma forma que se ha establecido una capa de recepción, una de proceso y otra de transmisión, una mejora interesante sería poder disponer de varias capas de procesamiento. Esto permitiría segmentar con mayor facilidad un trabajo y alcanzar througput aun mayores.
- **Realizar un módulo de marcado mejorado**. El sistema de marcado de paquetes sigue consumiendo gran parte del tiempo de procesamiento. La reducción de estos costes permitiría aumentar el número de flujos concurrentes que soporta la sonda. Adicionalmente, un tema interesante a investigar son los posibles y diferentes métodos para realizar aproximaciones en la toma de tiempos.
- **Mejorar el rendimiento de Flowprocess**. El módulo de creación de flujos utilizado no explota las capacidades que ofrece una arquitectura paralela. La construcción de una versión paralela capaz de alcanzar el límite de 14.88 megaflujos por segundo es un reto interesante.
- **Utilización de múltiples GPU**. Dado que el rendimiento del módulo clasificador depende del número de firmas, el uso de múltiples GPUs minimizaría el impacto del uso de un gran número de firmas a costa de tantas GPUs como el rendimiento deseado. Permitir el uso de más de una GPU es necesario si se desea trabajar con un gran número de firmas o si se desea alcanzar el reto de 20 Gbps.
- **Reconfiguración automática**. A lo largo del trabajo se ha mencionado varias veces la importancia de una buena configuración para exprimir el rendimiento del sistema. Sin embargo, el tráfico que recibe una sonda varía a lo largo del día y la noche. Como mejora se podría implementar un mecanismo de autoconfiguración diseñado para adecuarse al tráfico y sus variaciones.
- **Sonda clasificadora a 20 Gbps**. El siguiente, y más inmediato reto, consiste en ampliar este trabajo fin de grado para alcanzar los 20 Gbps.

Bibliografía

[1] Smartphone use per capita in 2011 by country. [Online]. Available: http://communities-dominate.blogs.com/brands/2011/12/smartphone-penetration-rates-by-country-we-have-good-data-finally.html

[2] "Procera NVL." [Online]. Available: http://goo.gl/44vM5

[3] "Veho DPI probe." [Online]. Available: http://www.veheretech.com/products/veho-dpi/

[4] "DPX network probe." [Online]. Available: http://www.ipoque.com/en/products/dpx-network-probe

[5] "Qosmos deepflow® probes." [Online]. Available: http://www.qosmos.com/products/deep-packet-inspectio-probes-for-telecoms/

[6] J. A. A. Sáez, "Procesamiento mediante gpu en netfilter (linux)," Master's thesis, Escuela Politécnica Superior, Universidad Autónoma de Madrid, 2012.

[7] P. G. Nieto, "Clasificación de tráfico tcp/ip mediante dispositivos gpu," Master's thesis, Escuela Politécnica Superior, Universidad Autónoma de Madrid, 2012.

[8] "Intel data plane development kit." [Online]. Available: http://intel.com/go/dpdk

[9] "Netfpga." [Online]. Available: http://netfpga.org/

[10] M. Forconesi, G. Sutter, S. López-Buedo, and C. Sisterna, "Clasificación de flujos de comunicación en redes de 10 gbps con fpgas," in *Jornadas SARTECO 2012*, 2012. [Online]. Available: http://www.jornadassarteco.org/js2012/papers/paper_64.pdf

[11] S. Han, K. Jang, K. Park, and S. Moon, "Packetshader: a gpu-accelerated software router," *SIGCOMM Comput. Commun. Rev.*, vol. 40, no. 4, pp. 195–206, Aug. 2010. [Online]. Available: http://doi.acm.org/10.1145/1851275.1851207

[12] V. Moreno, P. Santiago del Rio, J. Ramos, J. Garnica, and J. Garcia-Dorado, "Batch to the Future: Analyzing Timestamp Accuracy of High-Performance Packet I/O Engines," *Communications Letters, IEEE*, vol. 16, no. 11, pp. 1888 –1891, november 2012. [Online]. Available: http://arantxa.ii.uam.es/~vmoreno/Publications/Journals/morenoIEEECOMML2012.pdf

[13] L. Rizzo, M. Carbone, and G. Catalli, "Transparent acceleration of software packet forwarding using netmap," in *INFOCOM, 2012 Proceedings IEEE*, 2012, pp. 2471–2479.

[14] N. Bonelli, A. Di Pietro, S. Giordano, and G. Procissi, "On multi—gigabit packet capturing with multi—core commodity hardware," in *Proceedings of the 13th international conference on Passive and Active Measurement*, ser. PAM'12. Berlin, Heidelberg: Springer-Verlag, 2012, pp. 64–73. [Online]. Available: http://dx.doi.org/10.1007/978-3-642-28537-0_7

[15] "Models for packet processing on multi-core systems," White Paper, Intel, Dec. 2008. [Online]. Available: http://www.intel.com/content/www/us/en/intelligent-systems/intel-technology/ia-multicore-packet-processing-paper.html

[16] P. M. S. del Río, J. Ramos, J. L. García-Dorado, J. Aracil, A. C. Sánchez, and M. del Mar Cutanda-Rodríguez, "On the processing time for detection of skype traffic," in *IWCMC*, 2011, pp. 1784–1788.

[17] M. Crotti, M. Dusi, F. Gringoli, and L. Salgarelli, "Traffic classification through simple statistical fingerprinting," *SIGCOMM Comput. Commun. Rev.*, vol. 37, no. 1, pp. 5–16, Jan. 2007.

[18] J. Erman, A. Mahanti, M. Arlitt, and C. Williamson, "Identifying and discriminating between web and peer-to-peer traffic in the network core," in *Proceedings of the 16th international conference on World Wide Web*, ser. WWW '07. New York, NY, USA: ACM, 2007, pp. 883–892.

[19] J. Erman, M. Arlitt, and A. Mahanti, "Traffic classification using clustering algorithms," in *Proceedings of the 2006 SIGCOMM workshop on Mining network data*, ser. MineNet '06. New York, NY, USA: ACM, 2006, pp. 281–286.

[20] L. Bernaille, R. Teixeira, I. Akodkenou, A. Soule, and K. Salamatian, "Traffic classification on the fly," *SIGCOMM Comput. Commun. Rev.*, vol. 36, no. 2, pp. 23–26, Apr. 2006.

[21] L. Bernaille, R. Teixeira, and K. Salamatian, "Early application identification," in *Proceedings of the 2006 ACM CoNEXT conference*, ser. CoNEXT '06. New York, NY, USA: ACM, 2006, pp. 6:1–6:12.

[22] N. Cascarano, A. Este, F. Gringoli, F. Risso, and L. Salgarelli, "An experimental evaluation of the computational cost of a dpi traffic classifier," in *Proceedings of the 28th IEEE conference on Global telecommunications*, ser. GLOBECOM'09. Piscataway, NJ, USA: IEEE Press, 2009, pp. 1132–1139.

[23] R. Smith, C. Estan, S. Jha, and S. Kong, "Deflating the big bang: fast and scalable deep packet inspection with extended finite automata," *SIGCOMM Comput. Commun. Rev.*, vol. 38, no. 4, pp. 207–218, Aug. 2008.

[24] N. Cascarano, L. Ciminiera, and F. Risso, "Optimizing deep packet inspection for high-speed traffic analysis," *J. Netw. Syst. Manage.*, vol. 19, no. 1, pp. 7–31, Mar. 2011.

[25] M. Becchi, M. Franklin, and P. Crowley, "A workload for evaluating deep packet inspection architectures," in *Workload Characterization, 2008. IISWC 2008. IEEE International Symposium on*, Sept., pp. 79–89.

[26] L. Wang, S. Chen, Y. Tang, and J. Su, "Gregex: Gpu based high speed regular expression matching engine," in *Innovative Mobile and Internet Services in Ubiquitous Computing (IMIS), 2011 Fifth International Conference on*, 30 2011-July 2, pp. 366–370.

[27] G. Vasiliadis, S. Antonatos, M. Polychronakis, E. Markatos, and S. Ioannidis, "Gnort: High performance network intrusion detection using graphics processors," in *Recent Advances in Intrusion Detection*, ser. Lecture Notes in Computer Science, R. Lippmann, E. Kirda, and A. Trachtenberg, Eds. Springer Berlin Heidelberg, 2008, vol. 5230, pp. 116–134.

[28] B. Hutchings, R. Franklin, and D. Carver, "Assisting network intrusion detection with reconfigurable hardware," in *Field-Programmable Custom Computing Machines, 2002. Proceedings. 10th Annual IEEE Symposium on*, 2002, pp. 111–120.

[29] C. Clark and D. Schimmel, "Scalable pattern matching for high speed networks," in *Field-Programmable Custom Computing Machines, 2004. FCCM 2004. 12th Annual IEEE Symposium on*, 2004, pp. 249–257.

[30] J. Moscola, J. Lockwood, R. Loui, and M. Pachos, "Implementation of a content-scanning module for an internet firewall," in *Field-Programmable Custom Computing Machines, 2003. FCCM 2003. 11th Annual IEEE Symposium on*, 2003, pp. 31–38.

[31] H.-J. Jung, Z. Baker, and V. Prasanna, "Performance of fpga implementation of bit-split architecture for intrusion detection systems," in *Parallel and Distributed Processing Symposium, 2006. IPDPS 2006. 20th International*, 2006, pp. 8 pp.–.

[32] l7-filter. [Online]. Available: http://l7-filter.clearfoundation.com

[33] V. Paxson, "Bro: a system for detecting network intruders in real-time," *Computer Networks*, vol. 31, no. 23-24, pp. 2435 – 2463, 1999.

[34] "Netflow." [Online]. Available: http://www.cisco.com/go/netflow

[35] P. S. del Río, "Internet traffic classification for high-performance and off-the-shelf systems," Ph.D. dissertation, Universidad Autónoma de Madrid, May 2013. [Online]. Available: http://arantxa.ii.uam.es/~psantiago/Publications/dissertation.pdf

[36] Perl regular expressions. [Online]. Available: http://perldoc.perl.org/perlre.html

[37] Regular expression processor. [Online]. Available: http://regex.wustl.edu/index.php/Main_Page

[38] Maximizing gpu efficiency in extreme throughput applications. [Online]. Available: http://www.nvidia.com/content/GTC/documents/1122_GTC09.pdf

[39] "The caida ucsd anonymized internet traces 2009 - 20091217 - 045900-052900 sanjose," http://www.caida.org/data/passive/passive_2009_dataset.xml.

[40] H. Schulzrinne, S. Casner, R. Frederick, and V. Jacobson, "RTP: A transport protocol for real-time applications," RFC 3550, Jul. 2003. [Online]. Available: http://www.ietf.org/rfc/rfc3550.txt

[41] "Rtp header." [Online]. Available: http://research.edm.uhasselt.be/jori/thesis/onlinethesis/images/chapter_5/fig_rtpheader.png

[42] R. Leira Osuna, P. Gomez Nieto, I. Gonzalez, and J. Lopez de Vergara, "Multimedia flow classification at 10 gbps using acceleration techniques on commodity hardware," in *4th IEEE Technical CoSponsored International Conference on Smart Communications in Network Technologies 2013 (SaCoNeT 2013)*, Paris, France, Jun. 2013.

A
Artículo realizado

A continuación se muestra el artículo realizado, junto con la carta de aceptación del mismo para el SaCoNeT'2013.

```
Fecha:  26/04/13 (12:30:23 CEST)
De: SaCoNeT 2013 <saconet2013-chairs@edas.info>
Para: Rafael Leira Osuna <rafael.leira@estudiante.uam.es>
Cc: [Mostrar direcciones - 17 destinatarios]
Asunto:  [SaCoNeT 2013] Your paper #1569745783 ('Multimedia flow
 classification at 10 Gbps using acceleration techniques on commodity
 hardware')  has been accepted



Dear Mr. Rafael Leira Osuna:

Congratulations - your paper #1569745783 ('Multimedia flow
classification at 10 Gbps using acceleration techniques on commodity
hardware') has been accepted by IEEE ComSoc Technical co-sponsorship
SaCoNeT'2013 which will be published in the Proceedings of IEEE
SaCoNeT 2013 and IEEE Xplore.

Please carefully read the contents below, and follow up them.

Submission of the final version of your paper: The final manuscript
is due by April 15 and you must be registered for IEEE SaCoNeT 2013
to upload your paper. The IEEE SaCoNeT 2013 Registration will open on
9 April 2013, detailed instructions will be provided at
http://www.lissi.fr/saconet2013/wiki/registration.
Note that the final manuscripts are to be uploaded through EDAS.
```

Registration: To be published in the IEEE SaCoNeT 2013 Conference Proceedings and IEEE Xplore, an author (including students) of an accepted paper is required to register for the conference at the FULL rate and the paper must be presented at the conference.
Each paper must be registered with at least one Full Author registration by April 15, 2013 at the latest. For authors with multiple accepted papers, a reduced fee of each additional paper is requested. The full author registration fees include: the welcome cocktail, the coffee breaks, the gala dinner, the digital proceedings (including archiving, publication, and indexing), and the participation to all the Workshops under SaCoNeT 2013.
Accepted and presented papers will be published in the IEEE SaCoNeT 2013 Conference Proceedings and in IEEE Xplore.

Selected papers from those accepted and presented in the conference papers will be considered for publication in Network and Computer Applications Journal (Elsevier Ed., ISI and Scopus Indexed) and Annals of Telecommunication Journal (Springer Ed., ISI and Scopus Indexed).

Paper Revision: Note that each accepted paper is limited to 5 pages. Each extra page (for papers with more than 5 pages) will be charged with €75 over length charge. The final submitted version must be revised according reviewers' comments. However, both the title and author list of an accepted paper can NOT be changed in the final manuscript. Failure to abide this policy may result in dropping your paper from the program.

No-Show Policy: The organizers of IEEE SaCoNeT 2013 as well as attendees expect accepted papers to be presented at the conference. SaCoNeT reserves the right to exclude a paper from distribution after the conference (e.g., removal from IEEE Xplore) if the paper is not presented at the conference.

We suggest you make your airline and hotel reservations as early as possible. All informations are available for your venue (Hotels, Visa, etc.) at http://lissi.fr/saconet2013/wiki/venue.

Looking forward to seeing you in Paris.

Reviewers' Comments: The reviews are displayed below or can be found at http://edas.info, using your EDAS user name rafael.leira@estudiante.uam.es.

======= Review 1 =======

> *** Submission Policy: Does the paper list the same author(s), title
> and abstract in its PDF file and EDAS registration?

```
 Yes

> *** Strong aspects: Comments to the author: what are the strong
> aspects of the paper?

 The use of DPI is an interesting and timely topic that attracts much
 interest now. So, the topic is clearly interesting. The work is
 interesting.

> *** Weak aspects: Comments to the author: what are the weak aspects
> of the paper?

 The paper describes deep packet inspection using commodity hardware.
 It has a litterature review describing what has been done in the
 academic world on the subject. But i do not find a description of
 what is available off the shelf from commercial providers. This is my
 main problem with the paper. I know that i can buy systems that
 handle 2000 signatures on 10 Gbps. Probably from a handful of
 suppliers. So, why publish papers about systems that have
 significantly lower performance and flexibility? Possibly there is a
 really good motivation on why this is needed, but i don't find it in
 the paper. In fact, there is no discussion about commercial systems
 in the paper.
 Also, the number of signatures used in the evaluation seems very low
 to me. I know systems that have thousands of signatures. Therefore
 the GPU performance evaluation seems to be done with unrealistic
 parameters. What would the performance be in a realistic case? I
 guess, much lower?
 The tests and results section is very short and contains no analysis.
 The structure of the paper is poor. It contains basically one very
 long section (II) and this involves both description of the
 architecture and some kind of performance discussion. Also, i find it
 difficult to understand the figures, they are referred to, but not
 really explained. The language is poor, and it seems like the paper
 was written in haste.

> *** Recommended changes: Recommended changes. Please indicate any
> changes that should be made to the paper if accepted.

 The structure should be updated so that it is easier to read.
 The language must be worked with, i suggest to make an internal
 review of the language.
 There should be a description on state-of-the-art of commercial
 tools, as well as a motivation on what value this research brings in
 relation to this.
 There should be an extended results and/or analysis section,
 analyzing and discussing the results.

> *** Relevance and timeliness: Rate the importance and timeliness of
```

```
> the topic addressed in the paper within its area of research.
 Acceptable (3)

> *** Technical content and scientific rigour: Rate the technical
> content of the paper (e.g.: completeness of the
 analysis or simulation study, thoroughness of the treatise, accuracy
 of the models, etc.), its soundness and scientific rigour.
 Valid work but limited contribution. (3)

> *** Novelty and originality: Rate the novelty and originality of the
> ideas or results presented in the paper.
 Minor variations on a well investigated subject. (2)

> *** Quality of presentation: Rate the paper organization, the
> clearness of text and figures, the completeness and accuracy of
> references.
 Substantial revision work is needed. (2)

 ======= Review 2 =======

> *** Submission Policy: Does the paper list the same author(s), title
> and abstract in its PDF file and EDAS registration?

 The paper lists the same authors, title and abstract in its PDF file
 and EDAS registration.

> *** Strong aspects: Comments to the author: what are the strong
> aspects of the paper?

 The paper addresses important issue of multimedia flow classification
 on the fly. The approach of using commodity hardware is good and the
 presented results are promising. Paper is clearly written and reads
 well.

> *** Weak aspects: Comments to the author: what are the weak aspects
> of the paper?

 The language used in paper is sometimes too verbal. E.g. non-relevant
 sentences like "as mentioned previously", "as is explained before",
 "as we said in", "as we said before" are used often.

 After almost four pages, the authors state "now we can focus on our
 topic". If all material before that was not in the topic, it should
 be compressed a lot. Now the material in the authors' topic is only
 couple of sentences. I recommend removing this confusing statement.

> *** Recommended changes: Recommended changes. Please indicate any
> changes that should be made to the paper if accepted.
```

Section I: "Each network flow on the Internet has its own protocol".
That reads like there would be as many protocols as network flows.
Should be reformulated.

Mid-section I "... in almost multimedia protocols..", perhaps add "all"?

2.A.1 "... get the less ...", reformulate. "... as faster as...",
reformulate. "... to remark that could be...", reformulate.
"...basics knowledges..", knowledge is uncountable.

2.A.2 "Every network cards give us...", reformulate. "... not at
least with a performance loss", "without"? "...hinders parellelism
computing...", reformulate. "...flow until is exported as...",
reformulate".

Word "income" is used in a strange way in some places. Consider
replacing with something else.

"B "...from 256 bytes onwards, false positives and false negatives do
not increase..." Is this a good reason for only including 256 bytes
from the beginning? This sounds like you would like to maximize the
amount of false positives and negatives which probably is not the
case.
".. system have to support expiration..." = "has to support"
"For reach this goal...", reformulate.
"With this is possible..", reformulate.
"...DMA performance is archived...", "achieved"?
"Before start matching", reformulate.
"...each thread process..." = "processes"?
Figure 4 and 5: When printed with black and white, some information
is lost. Consider replacing colors with something what is disctinct
also in black and white. Add legend of colors to Fig 4.
"...instead of the the match...", remove "the"
"...to buffer the flows payloads...", "flow's" or "flows'" ?
"NVIDA"="NVIDIA"
"...implies have at most...", reformulate
"useful for detect bottlenecks", "detecting"?

Legend of fig 6 refers to Bittorrent. This is confusing as Bittorrent
is not mentioned anywhere else. Check and correct.

III.A. "we have disused", what its this? perhaps "discussed"?
The authors write about 25% probability of false positives if
classifying only based on RTP RFC. Can you describe more how you end
up with this number? Is it that you assume fixed version number in
the first two bits?

IV "From an accuracy point of view.." Do you mean something else than
accuracy?

```
 Can you describe table I a bit more. It is not clear why "Global"
 figures are significantly lower than the minimum of each pipeline
 step separately.

> *** Relevance and timeliness: Rate the importance and timeliness of
> the topic addressed in the paper within its area of research.
 Excellent (5)

> *** Technical content and scientific rigour: Rate the technical
> content of the paper (e.g.: completeness of the
 analysis or simulation study, thoroughness of the treatise, accuracy
 of the models, etc.), its soundness and scientific rigour.
 Valid work but limited contribution. (3)

> *** Novelty and originality: Rate the novelty and originality of the
> ideas or results presented in the paper.
 Some interesting ideas and results on a subject well investigated. (3)

> *** Quality of presentation: Rate the paper organization, the
> clearness of text and figures, the completeness and accuracy of
> references.
 Well written. (4)

 ======= Review 3 =======

> *** Submission Policy: Does the paper list the same author(s), title
> and abstract in its PDF file and EDAS registration?

 Yes

> *** Strong aspects: Comments to the author: what are the strong
> aspects of the paper?

 The paper proposes an implementation using common graphic cards with
 GPU to perform flow classification at high speed (10Gbps) using DPI
 techniques. With the growth of internet trafic, ISP have to
 differentiate flows (eg multimedia) to ensure an exceptable quality
 of experience to their customers, so this research is very important
 to provide a future internet with a high quality.

> *** Weak aspects: Comments to the author: what are the weak aspects
> of the paper?

 The results part is a bit disappointing as it only refers to RTP
 recognition. Futher, I would have liked to see some comparison with
 previous flow-classification techniques (accuracy, speed, ...)

> *** Recommended changes: Recommended changes. Please indicate any
```

```
> changes that should be made to the paper if accepted.

 The paper is well written and pleasant to read. However, I suggest to
 proof-read it to fix several grammatical mistakes.

> *** Relevance and timeliness: Rate the importance and timeliness of
> the topic addressed in the paper within its area of research.
 Excellent (5)

> *** Technical content and scientific rigour: Rate the technical
> content of the paper (e.g.: completeness of the
 analysis or simulation study, thoroughness of the treatise, accuracy
 of the models, etc.), its soundness and scientific rigour.
 Solid work of notable importance. (4)

> *** Novelty and originality: Rate the novelty and originality of the
> ideas or results presented in the paper.
 Some interesting ideas and results on a subject well investigated. (3)

> *** Quality of presentation: Rate the paper organization, the
> clearness of text and figures, the completeness and accuracy of
> references.
 Well written. (4)



 Best regards,
 Abdelhamid Mellouk
 General chair

 Nadjib Aitsaadi, Xavi Masip-Bruin, Mohammed Younis
 TPC co-chairs
```

Multimedia flow classification at 10 Gbps using acceleration techniques on commodity hardware

R. Leira, P. Gomez, I. Gonzalez, J.E. Lopez de Vergara
High Performance Computing and Networking Research Group, Escuela Politénica Superior
Universidad Autónoma de Madrid
Madrid, Spain
Email: rafael.leira@estudiante.uam.es, pedro.gomezn@gmail.com, {ivan.gonzalez, jorge.lopez_vergara}@uam.es

***Abstract*—Internet is an ever-growing network. The network equipment has to be improved to cope with this growth, including those devices used to classify the network traffic. Internet service providers and network operators require to apply different QoS policies for specific protocols. Then, such classifying systems are critical. However, classification by port does not provide good results, and it is necessary to apply other more complex techniques. These classification techniques have to be fast enough to work at line rates. This paper presents a system that unifies the entire process involved in flow classification at high speed. It captures the traffic, builds flows from the received packets and finally classifies them inside a GPU. All the process is possible at 10 Gbps using commodity hardware. Our results show that the achieved performance is very influenced by the number of protocols to find, and it is limited by the number of network flows. In any case, our system reaches up to 24.4 Gbps using commodity hardware.**

I. Introduction

The Internet size is growing and changing day by day, with more and more websites, protocols and end users. Each network flow on the Internet has an associated protocol. However classifying those protocols and flows is not a trivial task. Network operators need to classify flows at high speed. The classification is commonly used to identify network intrusions or to provide QoS (Quality of Service). For this purpose, a system is required that can support a sustained high throughput. The system also needs to build flows from the network at real time. And finally, the system has to run on commodity hardware to compete with expensive commercial products (like [1]).

Many works have already proposed different traffic classification methods. The most common classifiers are *classification by port*, *DPI (Deep Packet Inspection)* and *statistical classification*. In the same way, there are many works comparing the benefits and disadvantages between each of them [2]–[10]. After assessing each method, we have finally chosen DPI. Our work is based on [8]–[10] using also a DFA (Deterministic Finite Automaton) [8] implementation on a GPU, in a similar way as [9]. Some works have searched for other strategies different from DPI, assuming that DPI computational cost is very high [2]–[5]. In the same way, other works have studied on how to reduce DPI costs using different techniques and ideas [6]–[10]. In fact, some applications [10]–[12] have been developed that implement DPI. However, not all of those programs can work at high speed.

DPI costs are not as bad as expected [5], especially if we compare it with older techniques such as classification by port [13], in which the relation between reliability and computational costs promotes a lot the use of DPI instead of Port Classification. It is important to remark that in almost multimedia protocols, the port does not mean anything itself.

Using DFA for DPI has its own benefits and drawbacks. It is commonly known that DFA tables size grows quadratically, which severely limits the number of protocols. On the other hand, it gives a very good performance inside a GPU [9]. To solve DFA limitation, building as many DFA tables as necessary gives the possibility of handling and classifying as many protocols as desired. Then, the proposed system is not limited to a small number of protocols. Another advantage is the DFA itself as a mechanism of DPI. It gives one of the most important things in flow classification, the reliability. DFA tables are built using regular expressions which have all the precision and flexibility we need for each protocol signature.

In this paper, we propose a new way to speed up DPI processing for multimedia protocols using CUDA and GPUs. It also shows a new way to integrate it inside a network probe. This shows that Deep Packet Inspection with DFA can be used at very high speed with practically no system overhead. On the other hand, the Internet growth makes much more problems than high speed protocol recognition, which sometimes, can be seen as an obvious thing: obtain real high speed traffic (10Gbps and over) and build the flows on the fly. For this, we take advantage of the Intel DPDK. We address all of these issues along the paper.

The paper is organized as follows: In Section II we describe our architecture that includes how Intel DPDK runs with the data-flow and its performance, a description of the data-flow inside the GPU and the performance reached with it. In Section III we expose our multimedia approach, our results and problems. Finally, Section IV concludes the paper and proposes future works.

II. The architecture

A. *Intel® DPDK*

Intel® DPDK (Data Plane Development Kit) is a development kit created by Intel which has recently got opened to the public. It has been built with some of those simple purposes [14]:

1) Get the best performance (10 Gbps).
2) An easy implementation.
3) Exploit the full potential of multi-core architectures [15].
4) Make flexible and scalable programs.

1) Our data-flow approach: In order to make a scalable and independent system, we have selected and relied on an architecture example from the Intel® DPDK. That architecture was the best approach to our initial data-flow idea. The example that we selected was the "load balancer example". The idea is clear: split between packet reception, packet processing and packet transmission. In order to test every component individually with the less overhead as possible and have a big independence between them. This means to encapsulate each component in a core.

From now on, we would use the following notation:

- "I/O RX" for a reception core
- "worker" for a packet processor core
- "I/O TX" for a transmission core

Each core is connected to the next one using rings (circular queues). In a similar way as every NIC is connected to its designated core. This makes a really simple code for each core (Fig. 1). Especially if we are talking about I/O RX or I/O TX cores, which work consists on insert in another ring very quickly. However I/O RX cores can have some low cost process for split traffic between rings. Some fast splits could be: Split by some byte of the IP address, split by port, split between TCP and UDP protocols. This first classification is useful if we are trying to parallelize some kind of work in many cores. As mentioned before our approach processes inside the GPU. So it does not need any previous split. However this first split could be useful if there are more than one GPU to work with.

while true do
  for all $R_i \subset ConnectedRings$ **do**
    $PB \leftarrow getPacketBulk(R_i)$
    for all $P_i \subset PB$ **do**
      $P_j \leftarrow work(P_i)$
      $R_j \leftarrow selectDestinationRing(P_j)$
      $enqueue(P_j, R_j)$
    end for
  end for
end while

Fig. 1. Generic core algorithm

Having the basic knowledge, we can start talking about our data-flow. Our test just consists in 3 different cores, and 2 different NICS (Fig. 2). As mentioned previously, it could be as scalable as we want, but for our experiments 3 cores was enough and gave us the top 10 Gbps speed per NIC. One NIC for input packet and other for output packets.

2) Working with packets and building flows: As mention before, in this paper we classify flows on the fly. That give us a new problem: Each network card provides packets (or a bulk of them) but rarely it provides flows, not at least without a performance loss. Creating flows from packets is not trivial, neither for hardware or software. It requires a big amount of memory to save each flow state. It also requires a hash table to address each flow which hinders parallelism computing. Those problems are increased if we are talking about high speed networks and high number of flows. There are different ways to implement flow building, for example Intel® DPDK provides a hash-table to help in this task. However we have chosen to build our own solution, which is very similar to [16].

The process until a flow is built from many packets is as follows:

1) A new packet enters into the flow builder.
2) We find if the flow exists. If not, we create a register.
3) The flow builder adds to the register the packet payload. (if there are space left for it)
4) Exports all the flows which have reached an expiration condition. (Fig. 2)

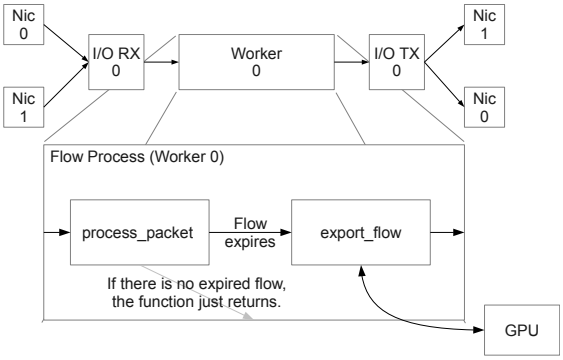

Fig. 2. Exporting flows to the GPU : The worker

B. Deep Packet Inspection over GPU

DPI (Deep Packet Inspection) consists in analyzing the transport layer payload of a packet in order to determine its nature within several contexts such as intrusion detection, P2P traffic detection, as well as our final goal: multimedia classification. The use of DPI for TCP/UDP traffic classification is based on finding out which protocol a flow belongs to. In order to do this, it is checked if a certain amount of bytes of the payload matches a regular expression, also known as signature of a protocol. If so, it may be said the flow belongs to that protocol. According to the research carried out in [7], it is estimated that from 256 bytes onwards, the signature accuracy do not increase or decrease for several well-known application layer protocols so we may say that 256 bytes is enough to perform DPI for TCP/IP traffic classification. In this work, we have used the l7-filter signatures [11] for TCP/IP traffic classification and the ones that we developed.

1) GPU architecture for a high throughput: In real time flow classification latency is not very important. In fact, it is not usually required to have small time between the flow is expired and the flow gets classified. However, if we are talking about 10 Gbps networks, the throughput increases the

latency significance. This means the system has to support expiration flows at those speed. The throughput also needs to be constant for any input flow. Typical GPU approaches are based in minimizing latency. An example is the CUDA streams. They are designed to split any work in smaller tasks. That approach allows the system to finish faster the original work. In order to maximize throughput, we have opted to split DPI in multiple tasks. Those tasks are finally joined inside a pipeline. In the next subsections we are going to explain each task individually. Ultimately we expose the final join between the tasks and the results given by the GPU.

2) The memory copy: Current GPUs needs a copy from the host memory to the GPU memory to work with. This copy is done over the PCIe bus, which is well-known that a one big data transfer has better performance than many smaller transfers. That forces us to have many buffers to cache the flows and minimize the data transfers costs. (Fig. 3).

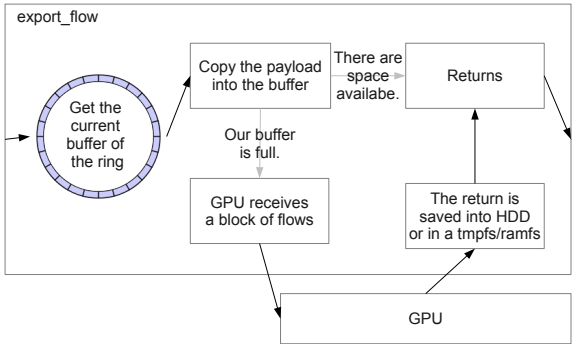

Fig. 3. Exporting flows to the GPU : GPU communication

Nevertheless there could be a performance loss during data transfers between the host and the GPU. In order to solve this problem, CUDA offers *streams*. A CUDA stream can transfer memory in parallel while any other CUDA kernel runs. This can be achieved thanks to DMA transfers and provides a better GPU performance. However, CUDA streams have some requirements. The most important one is the host memory. The flow payloads should be in a page-locked memory or in a pinned memory. This implies that the buffer mentioned before should be in one of those memories. Even if Huge-Pages have been chosen, it still implies one copy to the buffer. That copy implies some performance loss. On the other hand, the best DMA performance is achieved having all the flows contiguous in memory.

3) State transition table: Before the GPU can match, every protocol signature must be preprocessed. Each protocol signature is composed by a name and a regular expression. In a similar way to [9], the set of regular expressions [17] must be transformed into a equivalent DFA. For this purpose, we use the toolset Regular Expression Processor [18]. This toolset has several source codes which offer the generation of a DFA from a set of regular expressions. After the DFA has been built, we create a state transition table. The rows denote the actual state while the columns denote the actual input character. Each cell is an 4 byte integer with the following fields:

- 1st byte is used to represent up to 8 matched signatures. The bit i denote with 1 that the signature i has been matched.
- 2nd, 3rd y 4th bytes are used to represent up to 224 possible destination states. It is generally enough for a set of 8 complex regular expressions.

4) Payload processing: In order to process the payloads, we use a CUDA kernel. It obtains matched signatures from a set of payloads through a state transition table previously stored in global memory. Each payload is processed by a different thread. The payloads are stored in global memory by columns instead of rows in a similar way as [9]. The flow transposition is done inside the GPU to gain performance [19]. For each input byte processed, an output byte is written. Each output byte contains the signatures matched for its input byte. The ith bit of the byte, represents the ith signature matched for a specific input byte. To obtain the final result of the match instead of the match time, we have to do a reduction. The best performance solution is another parallel kernel which applies an OR operation over every processed byte. The size reduction also decreases the GPU to host data transfers. While saving matches in one byte improves performance, it also limits the number of signatures at 8. To fix this is required to run as many matching kernels as sets of 8 signatures we have. Those kernels can run in parallel but shares the memory cache. As we discuss in the following the cache sharing may penalize the global performance.

5) The pipeline: In order to obtain a high and constant throughput, [18] purposes a pipeline. The pipeline consists in split the process into several steps. The typical steps are the following:

1) Computing inside the host
2) Transfer data to de GPU
3) Computing inside the GPU
4) Transfer data to de host

Every step is independent. That means there are 4 tasks that can run in parallel. Nevertheless that parallelism can only be achieved if our NVIDIA card has a compute capability greater or equal to 2.0. For that, it is required to use more than one kernel to transpose, match and reduce the results. This involves the use of the previous pipeline with more than one computing step. This new pipeline has the following steps (Fig. 4):

1) Payload buffering: As we said in Section II-A, we have to buffer the flow's payloads. This buffer is reserved in big, contiguous and non-swappable memory. As is explained in [9], this provides the biggest performance during the copy to the GPU. This step is done inside the CPU and the DPDK when every flow expires.
2) Transfer to GPU: The buffer is copied to the GPU memory waiting to be processed.
3) Transpose: As we said before, the flows must be transposed in columns in order to obtain the best performance.
4) Signature matching: Every set of 8 signatures launches an independent kernel. Every kernel provides its own

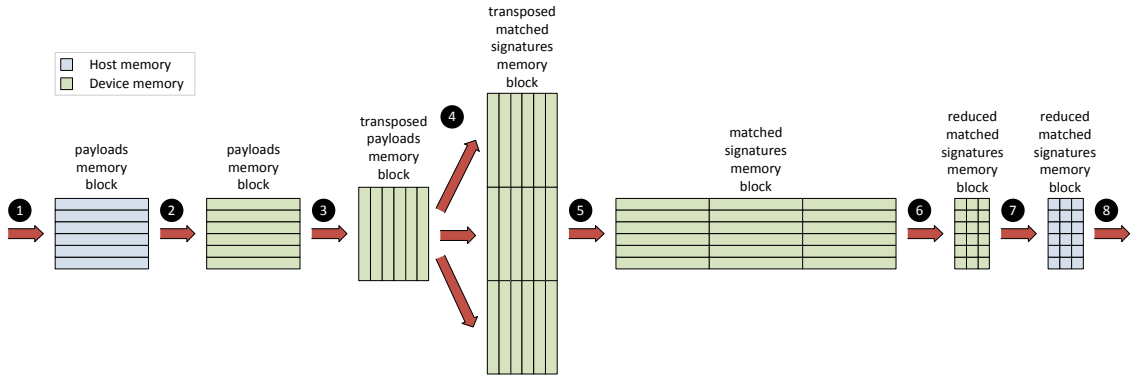

Fig. 4. Pipeline inside the GPU

result set. Those sets are consecutive in GPU memory to make easier the next step.

5) Transpose: The data should be transposed again to be more readable for the CPU. This work can be done by another parallel instance of the other transpose kernel.
6) Reduction: In order to minimize data transfer the memory have to be minimized. For that, we applies a parallel reduction over the results. The reduction orders the matches computed for each flow, and save the results consecutively in memory saving memory.
7) Transfer to CPU: The reduced results are copied to host memory.
8) Work with the results: When the results are in the host memory, we can work or have decisions with it. Some examples are: save statistics to disk, alert of an intrusion sending a packet, and our goal: control a QoS system and advise which flows have priority over other flows.

6) The DPI performance: Inside the pipeline, the fourth step implies many kernels running inside the GPU concurrently. This implies more memory cache conflicts between kernels. For NVIDIA Fermi architecture, it also implies having a maximum of 16 concurrent kernels inside the GPU. This means at most of 88 signatures without kernel serialization. Nevertheless the number of kernels and the memory accesses penalizes performance as is shown in Fig. 5.

In Table I, we can view each step performance for a specific number of signatures. To take individually measures, we have to execute each kernel serially. Individually kernel measures are quite useful for detecting bottlenecks but it do not includes the cache penalization while it is share by other kernels. That is the reason for the Global Performance differs a lot of the lower value of its line. Using different test traces, we have found from an input of 10 Gbps Ethernet network, we obtain an output of 73.2 KFlows/s. This means, with 40 signatures, our GPU can handle, on the fly, 50 times more flows that are in our traces (sent at 10 Gbps). However, there could be networks with much more flows, which imply a limitation in our architecture. In the worst case one packet represents one different flow and has the minimum size allowed. We

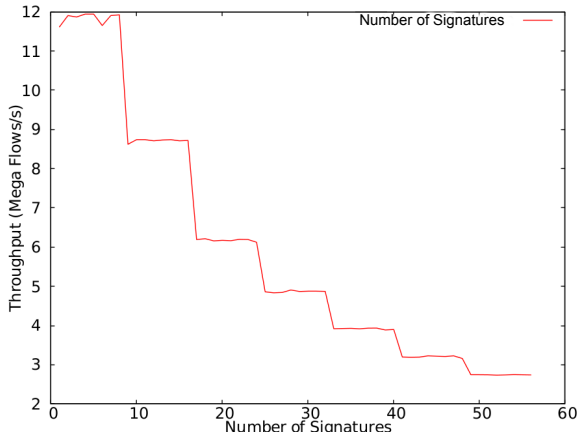

Fig. 5. GPU performance

can suppose that minimum size for each flow inside Ethernet is 77 bytes (including preamble and interframe gap of 5). In this case, the network has about 16 MFlows/s. Our approach can afford about 73% of the worst case. Nevertheless, in real world is very difficult to reach 16 MFlows/s inside a 10 Gbps network. CAIDA traces [20] has about 0.7 MFlows/s at 4 Gbps. This means about 1.7 MFlows/s at 10 Gbps that perfectly fits with the proposed system.

III. The Multimedia Approach: Results, Problems and Future Work

A. The RTP problem

In the last few lines we have discussed about a generic DPI machine which works at 10Gbps, however our final goal is to guarantee the Quality of Service of a network.

To provide QoS to multimedia traffic it is important to correctly classify RTP protocol. However, RTP does not provide much information for such classification [21]. If we are strict with the RFC, we have only the 2 first bits of the flow for classifying using DPI. This means that each flow has the probability of be bad classified as RTP of 25% (false positive). On the other hand, with that approach, we will not have any false negative.

TABLE I
GPU PERFORMANCE FOR EACH CUDA KERNEL (IN GBPS)

Number of signatures	Transfer to GPU	Transpose	Matching	Transpose	Reduction	Transfer to CPU	Global (Gbps/MFlowsps)
8	35.9195	90.843	51.398	98.2704	64.834	260.417	24.414 / 11.9
16	32.417	84.4595	47.3485	53.6942	34.4923	169.837	17.857 / 8.7
24	33.1741	72.6744	43.2825	36.507	23.3907	124.008	12.540 / 6.1
32	31.3755	64.5661	39.557	27.8025	17.7154	100.806	9.978 / 4.8
40	29.7619	58.7406	27.0797	22.4497	14.3349	85.3825	7.984 / 3.9

However, there are many other ways to do a signature for RTP, especially if we look at the PT field. The PT field represents which type of multimedia is inside the RTP flow. In other words, we can use PT field as a magic word to identify an RTP flow and the content of it at the same time. This solution provides a way to do a different QoS for each type of multimedia, not just for multimedia flows in general. That could bring an easy way for ISPs to provide a better service to the users with low costs.

Nevertheless is important to remark that classification reliability depends in a big part on the signature's quality.

B. Tests and Results

To test our DPI engine over a multimedia network, we have retransmitted a trace from the IPNQSIS project. The goal was to detect all of the RTP and RTCP flows from that capture. The results show only 2 false negatives in RTCP classification. After verifying thoroughly our system, we found only a problem inside the signature which was very strict. This reminds a problem mentioned before: The system accuracy is as accurate as the signature set accuracy.

IV. CONCLUSION AND FUTURE WORK

In this paper we have proposed an application for flow classification that works at 10 Gbps, reading from a NIC. In addition, the GPU processes up to 24.4 Gbps which means about 12 Mega Flows per second. From an accuracy point of view, our GPU can address around 73% of the worst-possible scenario. However in common scenarios the application can easily handle a 10 Gbps network.

The results also show how important signatures are when using DPI for flow classification. Signatures define the accuracy of protocol classification. The accuracy of each signature and how it influences false positives and false negatives should be studied in future works.

We also plan to apply the methodology of this paper to other flow protocols, such as P2P, HTTP, or attacks and intrusions, and many others.

ACKNOWLEDGMENT

This work has been carried out in the framework of the Celtic and EUREKA initiative IPNQSIS and has been partially funded by CDTI under Spanish project PRINCE.

REFERENCES

[1] "Procera nvl." [Online]. Available: http://goo.gl/44vM5
[2] J. Erman, M. Arlitt, and A. Mahanti, "Traffic classification using clustering algorithms," in *Proceedings of the 2006 SIGCOMM workshop on Mining network data*, ser. MineNet '06. New York, NY, USA: ACM, 2006, pp. 281–286.
[3] L. Bernaille, R. Teixeira, I. Akodkenou, A. Soule, and K. Salamatian, "Traffic classification on the fly," *SIGCOMM Comput. Commun. Rev.*, vol. 36, no. 2, pp. 23–26, Apr. 2006.
[4] L. Bernaille, R. Teixeira, and K. Salamatian, "Early application identification," in *Proceedings of the 2006 ACM CoNEXT conference*, ser. CoNEXT '06. New York, NY, USA: ACM, 2006, pp. 6:1–6:12.
[5] N. Cascarano, A. Este, F. Gringoli, F. Risso, and L. Salgarelli, "An experimental evaluation of the computational cost of a dpi traffic classifier," in *Proceedings of the 28th IEEE conference on Global telecommunications*, ser. GLOBECOM'09. Piscataway, NJ, USA: IEEE Press, 2009, pp. 1132–1139.
[6] R. Smith, C. Estan, S. Jha, and S. Kong, "Deflating the big bang: fast and scalable deep packet inspection with extended finite automata," *SIGCOMM Comput. Commun. Rev.*, vol. 38, no. 4, pp. 207–218, Aug. 2008.
[7] N. Cascarano, L. Ciminiera, and F. Risso, "Optimizing deep packet inspection for high-speed traffic analysis," *J. Netw. Syst. Manage.*, vol. 19, no. 1, pp. 7–31, Mar. 2011.
[8] M. Becchi, M. Franklin, and P. Crowley, "A workload for evaluating deep packet inspection architectures," in *Workload Characterization, 2008. IISWC 2008. IEEE International Symposium on*, Sept., pp. 79–89.
[9] L. Wang, S. Chen, Y. Tang, and J. Su, "Gregex: Gpu based high speed regular expression matching engine," in *Innovative Mobile and Internet Services in Ubiquitous Computing (IMIS), 2011 Fifth International Conference on*, 30 2011-July 2, pp. 366–370.
[10] G. Vasiliadis, S. Antonatos, M. Polychronakis, E. Markatos, and S. Ioannidis, "Gnort: High performance network intrusion detection using graphics processors," in *Recent Advances in Intrusion Detection*, ser. Lecture Notes in Computer Science, R. Lippmann, E. Kirda, and A. Trachtenberg, Eds. Springer Berlin Heidelberg, 2008, vol. 5230, pp. 116–134.
[11] l7-filter. [Online]. Available: http://l7-filter.clearfoundation.com
[12] V. Paxson, "Bro: a system for detecting network intruders in real-time," *Computer Networks*, vol. 31, no. 23-24, pp. 2435 – 2463, 1999.
[13] M. Dusi, F. Gringoli, and L. Salgarelli, "Quantifying the accuracy of the ground truth associated with internet traffic traces," *Computer Networks*, vol. 55, no. 5, pp. 1158 – 1167, 2011.
[14] "Intel dpdk." [Online]. Available: http://intel.com/go/dpdk
[15] "Models for packet processing on multi-core systems," White Paper, Intel, Dec. 2008.
[16] P. M. S. del Río, J. Ramos, J. L. García-Dorado, J. Aracil, A. C. Sánchez, and M. del Mar Cutanda-Rodríguez, "On the processing time for detection of skype traffic," in *IWCMC*, 2011, pp. 1784–1788.
[17] Perl expressions. [Online]. Available: http://perldoc.perl.org/perlre.html
[18] Regular expression processor. [Online]. Available: http://regex.wustl.edu/index.php/Main_Page
[19] Maximizing gpu efficiency. [Online]. Available: http://www.nvidia.com/content/GTC/documents/1122_GTC09.pdf
[20] "The caida ucsd anonymized internet traces 2009 - 20091217 - 045900-052900 sanjose," http://www.caida.org/data/passive/passive_2009_dataset.xml.
[21] H. Schulzrinne, S. Casner, R. Frederick, and V. Jacobson, "RTP: A transport protocol for real-time applications," RFC 3550, Jul. 2003.