UNIVERSIDAD NACIONAL DE CÓRDOBA

Facultad de Ciencias Exactas, Físicas y Naturales

Tesis Doctoral

Arquitecturas Reprogramables de Procesamiento de Flujos de Paquetes en Redes de Datos

Autor: Carlos Alberto Zerbini

Director: Prof. Dr. Jorge Manuel Finochietto

Junio de 2015

Arquitecturas Reprogramables de Procesamiento de Flujos de Paquetes en Redes de Datos

por

Ing. Carlos Alberto Zerbini

Prof. Dr. Jorge Manuel Ficnochietto

Director

COMISIÓN ASESORA

Prof. Dra. Graciela Corral-Briones

FCEFYN - UNC

Prof. Dr. Víctor Hugo Sauchelli

FCEFYN - UNC

Esta Tesis fue enviada a la Facultad de Ciencias Exactas Físicas y Naturales de la Universidad Nacional de Córdoba para cumplimentar los requerimientos de obtención del grado académico de Doctor en Ciencias de la Ingeniería.

Córdoba, Argentina, Junio de 2015

Prefacio

Esta Tesis es presentada como parte de los requisitos para optar al grado académico de Doctor en Ciencias de la Ingeniería de la Universidad Nacional de Córdoba, y no ha sido presentada previamente para la obtención de otro título en esta Universidad u otras. La misma contiene los resultados obtenidos en investigaciones llevadas a cabo durante el período comprendido entre el 1 de Marzo de 2009 y el 26 de Junio de 2015 en el Laboratorio de Comunicaciones Digitales (FCEFyN-UNC) y el Grupo de Ingeniería Clínica (UTN-FRC) bajo la dirección del Dr. Jorge M. Finochietto y la Co-dirección del Ing. Eduardo A. González.

Carlos Alberto Zerbini

carloszerbini@gmail.com

czerbini@electronica.frc.utn.edu.ar

carlos.zerbini@unc.edu.ar

Agradecimientos

Este camino comenzó cuando el Prof. Eduardo A. González me acogió en su grupo de trabajo de UTN-FRC para enseñarme el camino de la investigación y desarrollo en el ámbito universitario. En el momento en que finalicé mi carrera de grado y debía decidir qué camino seguir, fue él quien me mostró la senda del Doctorado. A él se sumaron otros profesores que apoyaron esta iniciativa, como los Profs. Fernando Cagnolo, Oscar Anunziata, Hugo Grazzini, Luis Canali y Eduardo Menso, a quienes estaré siempre agradecido por su apoyo.

Fue entonces, cuando yo debía dar los primeros pasos, que el Dr. Jorge M. Finochietto apostó por mí, dándome aliento para transitar el nuevo camino. Fue en los momentos más difíciles donde su experiencia y desinteresada guía estuvieron siempre. El Dr. Finochietto me presentó asimismo al grupo de trabajo del Laboratorio de Comunicaciones Digitales, en el cual conté además con el apoyo de su directora, Ing. Carmen Rodríguez, los Profs. Mario Hueda, Graciela Corral-Briones y todo su personal.

Comenzó así un camino de aprendizaje y experiencias que tuve el gusto de compartir con profesores y compañeros que forman parte de esta Tesis. Como el Ing. Santiago Paz, con quien exploramos durante largas jornadas el mundo del diseño digital para redes de datos, no siempre logrando los resultados esperados pero sí con la satisfacción de entender y comprobar cada nuevo concepto. O los Ings. Jairo Trad y Luis Romano, con quienes logramos una interesante interacción entre los mundos de la electrónica y la informática. Y por supuesto mi compañero desde los días de estudios secundarios, Ing. Guillermo Riva, con quien tuve el gusto de transitar cada curva del camino, cada uno a su estilo pero con la misma pasión por lo que hacemos.

En el curso de mi Doctorado he tenido el agrado de conocer y contar con la enseñanza de notables profesionales en las áreas que éste involucra. Agradezco al Dr. Gustavo Sutter, quien amablemente dirigió mi trabajo durante mi estadía en la Universidad Autónoma de Madrid. Con él y el Dr. Elías Todorovich compartí una gran experiencia tanto en el campo profesional como personal, la cual dio nuevo impulso a mi trabajo de Doctorado. Asimismo he tenido el agrado de contar con la enseñanza de profesionales como el Dr. Eduardo Romero, Dra. Gabriela Peretti, Dra. Patricia Kisbye, Ing. Pablo Cayuela, y MSc. Cristian Cisterna. Finamente, agradezco la esmerada revisión y

los aportes realizados por la Comisión Asesora, Dra. Ing. Graciela Corral-Briones y Dr. Ing. Víctor Hugo Sauchelli; y por el Tribunal Evaluador, Dra. Ing. Luciana De Micco, Dr. Ing. José Ignacio Álvarez-Hamelin y Dr. Mag. Ing. Jorge Castiñeira Moreira.

Nada de lo mencionado hubiera ocurrido sin mi familia, que es la base de lo que soy. Mi madre y mi padre estuvieron siempre a mi lado en este camino e incentivaron siempre mi esencia humana, apoyando con su consejo cada pasaje de mi vida. Más allá de ellos, tengo la dicha de contar en mi familia con las voces más criteriosas y más arriesgadas, más técnicas y más artísticas. Todas ellas estuvieron presentes en esta etapa, como deseo lo sigan estando en cada momento de mi vida.

Debo a la Universidad Tecnológica Nacional (UTN), a la Universidad Nacional de Córdoba (UNC), a la Agencia Nacional de Promoción Científica y Tecnológica del Ministerio de Ciencia, Tecnología e Innovación Productiva (MINCyT) de la Nación y al Consejo Nacional de Investigaciones Científicas y Técnicas (CONICET) cada una de mis jornadas de trabajo en estos años; a estas Instituciones y sus hacedores les estaré siempre agradecido por permitir que yo y muchos otros estudiantes podamos desarrollar esta actividad con plenitud en nuestro país. Es mi deseo en la nueva etapa que comienza estar a la altura de sus expectativas; tengan por seguro que mis mejores esfuerzos estarán dedicados a ello.

UNIVERSIDAD NACIONAL DE CORDOBA
Facultad de Cs. Exactas, Fisicas y Naturales

ACTA DE EXAMENES

Libro: 00001 Acta: 03087 Hoja 01/01
LLAMADO: 1 26/06/2015
CATEDRA - MESA:

DI002 TESIS DOCTORADO EN CIENCIAS DE LA INGENIERIA

NUMERO	APELLIDO Y NOMBRE	DOCUMENTO	INGRESO	COND.	NOTA	FIRMA
27114861	ZERBINI, Carlos Alberto	DNI: 27114861	2009	T	Aprobado	

DE MICCO, Luciana - CASTIÑEIRA, Jorge - ALVAREZ - HAMELIN, José - HUEDA, Mario Rafaél -

Observaciones:

Dr. Ing. Luis A. Godoy
DIRECTOR
Doctorado en Ciencias de la Ingeniería
F C E F y N - U N Córdoba

Luis A. Godoy

Córdoba, ____/____/_____-.
Certifico que la/s firma/s que ha/n sido puesta/s en la presente Acta pertenece/n a:_____

Inscriptos	Ausentes	Examinados	Reprobados (0-3)	Aprobados (4-10)
1	0	1	0	1

19/06/2015 09:57:50 Libro/Acta: 0000103087 Hoja: 01/ 01

Resumen

Las redes de datos experimentan un sostenido crecimiento en cuanto a volumen y complejidad de la información transportada. La necesidad de cursar mayor volumen de datos en el mismo tiempo se traduce en la necesidad de mayores velocidades de transferencia, mientras que la mayor complejidad en la información requiere mecanismos de enrutamiento de paquetes de datos más elaborados. En particular, la *virtualización de recursos* en las redes actuales plantea nuevas exigencias de procesamiento a los equipos intervinientes en las redes de datos.

En el aspecto tecnológico, las plataformas basadas en dispositivos de lógica reconfigurable (FPGAs) son ampliamente adoptadas para su aplicación en redes de datos. Ésto se debe a sus características de flexibilidad, cercanas a las del software; y de desempeño, cercanas a las de circuitos de aplicación específica. Estos dispositivos se encuentran en permanente evolución en cuanto a prestaciones, ofreciendo un amplio conjunto de recursos computacionales de aplicación tanto general como específica al caso de comunicaciones.

A fin de diseñar arquitecturas de conmutación adecuadas a un escenario de redes con recursos virtualizados, resulta fundamental conocer los requerimientos que estas redes plantean. Para ello, se analizan en primer término las funciones básicas de procesamiento y conmutación requeridas, llamadas *primitivas*, y se evalúan las mismas en una implementación real sobre plataformas FPGA.

Sobre la base de la arquitectura implementada, se identifica la etapa crítica para su correcto desempeño, la cual es la función de *clasificación*. Esencialmente, esta función define *caminos de procesamiento* para cada paquete circulante en base a *reglas o filtros* previamente definidos; esto la transforma en una pieza esencial para el desempeño de la red. Si bien existe abundante trabajo sobre esta primitiva, la evolución de las redes de datos le impone nuevos desafíos; por lo que es necesario diseñar alternativas que se adapten a ellos. Entre estos desafíos, se destacan las necesidades de clasificación *multi-campo* y *multi-coincidencia*. La primera requiere considerar múltiples campos de un paquete para tomar una decisión sobre él, mientras que la segunda requiere que la decisión tomada se base en múltiples reglas coincidentes. Por otro lado, el uso de recursos virtualizados requiere comunmente *actualización dinámica* frecuente del conjunto de reglas, lo cual no era una exigencia en el pasado. De las distintas aproximaciones al problema de clasificación, se adopta en

este trabajo la técnica de *descomposición*, la cual considera cada campo por separado y luego agrega los resultados obtenidos. Esta técnica involucra dos etapas principales, llamadas *búsqueda (lookup)* en campos individuales y *agregación* de sus resultados.

En primer lugar, se observa que los trabajos existentes en clasificación por descomposición no permiten explotar en forma flexible el espectro de recursos computacionales que ofrecen los nuevos dispositivos FPGA. Sobre esta base, se presenta una nueva arquitectura de agregación específicamente orientada a combinar eficientemente recursos de lógica combinacional y de memoria, logrando un desempeño adecuado a los requerimientos actuales. La arquitectura propuesta se implementa en un FPGA, validando su eficiencia y comparando sus características con las de otros trabajos.

En segundo lugar, se observa que las técnicas de búsqueda (lookup) presentes en trabajos relacionados no son en general diseñadas para su óptima interacción con la etapa de agregación utilizada, lo cual afecta el rendimiento de clasificación. Para atacar este problema, se analizan exhaustivamente los esquemas de lookup existentes y su aplicación en clasificación multi-campo. Sobre esta base, se proponen y evalúan arquitecturas de lookup genéricas y se realizan implementaciones utilizando los distintos recursos computacionales disponibles en un FPGA, ponderando el rendimiento de ellas y su interacción con las arquitecturas de agregación existentes. Los resultados obtenidos aportan criterios de selección del esquema de lookup más adecuado para cada etapa de agregación. En particular, se identifica un esquema especialmente adecuado a las necesidades actuales, el cual sin embargo presenta serios problemas de implementación, y se propone una optimización especialmente orientada a mitigar los mismos.

Finalmente, a modo de generalización de las arquitecturas planteadas y como trabajo futuro, se proponen técnicas para *reducción de complejidad* del problema de clasificación, las cuales son completamente generales y aplicables a distintas arquitecturas de clasificación. Asimismo se estudian conjuntos de reglas reales (rulesets) a fin de evaluar las complejidades prácticas de clasificación.

La presente Tesis de Doctorado pretende, en primer término, aportar una nueva base teórica de acuerdo al estado del arte. A partir de ésta, se brindan en segundo lugar criterios de diseño que permitan un mejor entendimiento de los compromisos de implementación en FPGAs, conduciendo de esta forma a arquitecturas más eficientes. La efectividad de estas implementaciones se demuestra mediante pruebas de concepto sobre plataformas de lógica reconfigurable, las cuales son crecientemente adoptadas en este tipo de aplicaciones. Asimismo, se busca introducir este análisis en un contexto de redes de datos con recursos virtualizados, donde los problemas de procesamiento y conmutación plantean nuevos desafíos.

Summary

Data networks experience sustained growth both in volume and complexity of the conveyed information. The need for trasferring higher data volumes in the same time results in higher transfer rates, while higher data complexity requires more elaborated routing mechanisms. In particular, the adoption of *Resource virtualization* in current networks poses new processing demands at involved equipment.

From the technical aspect, platforms based on programmable logic devices (FPGAs) are widely adopted for data networks. This is due to their flexibility, which is near to that of software; and their performance, which is near to that of application-specific hardware. FPGA devices show sustained evolution in their features, offering a wide spectrum of both general-purpose and networing-specific resources.

To design architectures which fit the actual networking needs, a deep understanding of the involved requirements is in order. To this end, we first dissect and study the involved processing and switching functions, thereafter called *primitives*; and evaluate them on a real FPGA platform.

On this basis, we select the most critical function for network performance, which is the *clasification* primitive. Considering *rules or filters* previously defined, this primitive defines *processing paths* for each packet; this functionality makes classification a key piece for performance of the whole network. Even though extensive research exists on this function, the evolution of networks poses new challenges to it; as a consequence new proposals are needed which fit these challenges. Most remarkable challenges are the need for *multi-field* and *multi-match* classification. The first implies considering multiple headers of the packet header to take a desicion on it; while the latter implies that all of the matched rules affect the taken decision. In addition the adoption of virtualized resources introduces the need of frequent *dynamic updating* of the ruleset, which was not a requirement in the past. From the present approaches to the clasification problem, we adopt that based on *decomposition*; which first considers individual fields and then aggregates the obtained results for each field into the final result. Accordingly, this technique involves two main processing stages, i.e., *lookup* on each field and subsequent *aggregation* of lookup results.

In first place, we note that present work on decomposition-based classification do not allow considering different trade-offs between the wide spectrum of available FPGA resources. On tis basis, we propose a new aggregation architecture aimed at combining efficiently combinational logic and memory on FPGAs, with performance according to actual needs. Th eproposed architecture is evaluated for FPGAs, demonstrating its efficiency against existent work.

On second place, we note that previously proposed lookup techniques are mostly designed without considering their interaction with aggregation stages, which affects the performance of the whole multi-field classification architecture. To tackle this problem, we make an exhaustive survey os current lookup schemes and their application for multi-fiels classification. From the so-observed problems, we propose and evaluate a reduced group of generic lookup schemes, we implement them on FPGAs to asses their performance and finally compare them with current aggregation schemes. The obtained results provide selection criteria for effectively pairing lookup and aggregation schema. In particular, we identify the lookup scheme which seems more adequate for future network processing needs, which at the moment presents severe difficulties for its implementation. To deal with these problems, we propose an effective optimization.

As generalization of proposed schemes and future work, we finally propose techniques for *complexity reduction* of the classification problem. These algorithmic techniques are general and can be applied to several classification architectures. We also study real rulesets in order to assess their complexities and how far are they from theoretical bounds.

On first place, this Thesis contributes an updated theoretical ground according to the state-of-the-art in classification techniques. On this basis, new design criteria are discussed which allow better understanding of implementation trade-offs on FPGAs and in turn more efficient designs targeting such devices. The efficiency of the proposed designs is demonstrated through concept proofs on reconfigurable logic platforms. In particular, the proposed techniques can be efficiently applied in the context of virtualizad resources, which steadily pose new challenges to packet processing and switching in current data networks.

Zusammenfassung

Datennetze erleben nachhaltiges Wachstum in Umfang und Komplexität vor übertragenen Informationen. Die Notwendigkeit, mehr Daten in der gleichen Zeit zu übertragen fordert höhere Übertragungsraten an, während die komplexeren Informationsinhalt fragt nach verarbeiteren Routing Mechanismen. Insbesonders Ressource Virtualisierung in bestehenden Netze stellt neue Anforderungen an die beteiligten Verarbeitung.

Auf der Technologie, rekonfigurierbaren Logikbausteinen (FPGAs) Plattformen weit verbreitet sind für den Einsatz in Datennetze. Dies ist auf ihre Flexibilität, die in der Nähe der Software; und Leistung, die in der nähe der von ASICs liegt. Diese Geräte sind in der Leistung entwickelt, bieten eine breite Palette von IT-Ressourcen sowohl allgemeine als auch spezifische Anwendung auf den Fall der Kommunikation.

Um Schaltarchitekturen, die virtualisierten Netzwerkressourcen effizient unterstützen, ist es wichtig, die Anforderungen, die diese Netzwerke stellen zu wissen. Um dies zu tun, zum einen werden die grundlegenden Bausteine für Verarbeitung und Schalt, namens *Primitives* untergesucht, und werden sie in einer tatsächlichen Implementierung auf FPGA-Plattformen bewertet.

Auf der Grundlage der gabauten Architektur, wird die kritischsten Phase für die ordnungsgemässe Erfüllung identifiziert, die die *Klassifizierung* Primitive ist. Im Wesentlichen definiert diese Funktion *Verarbeitungspfade* für jedes Paket auf Basis zuvor definierten *Regeln* oder Filtern; auf diesem Grund ist sie ein wesentliches Baustein für die Leistung des Netzwerks. Zwar gibt es viel Arbeit auf diesen Primitiven, die Entwicklung der Datennetze stellt aber neue Herausforderungen an; so ist es notwendig, Alternativen, die ihnen passen zu entwerfen. Unter diesen Herausforderungen befindet sich, Hervorhebung der Bedürfnisse auf *Mehrfeld* und *Mehrzustimmung* Klassifizierung. Die erste erfordert die Berücksichtigung von mehreren Feldern eines Pakets, um eine Entscheidung darüber zu machen, während die zweite erfordert, dass die Entscheidung über die mehrfachen Übereinstimmungsregeln basiert. Weiterhin, der Einsatz von virtualisierten Ressourcen erfordert üblicherweise häufige *dynamische Aktualisierung* des Regelwerks, die nicht eine Anforderung in der Vergangenheit war. Aus den verschiedenen Ansätze für das Problem der Klassifizierung, wird in dieser Arbeit die Technik der *Zersetzung* angenommen, die jedes Feld getrennt hält und dann die Ergebnisse zusam-

menfügt. Diese Technik umfasst zwei Hauptschritte, genannt *Suche (Lookup)* in einzelne Felder und *Aggregation* der Ergebnisse.

Zunächst wird es bemerkt, dass bestehende Arbeiten erlauben nicht das flexibel Nutzen des von neue FPGA-Geräte angeboten Rechenressourcen. Auf dieser Basis wird eine neue Architektur vorgestellt, die speziell orientiert wird zum effizient bündeln der kombinatorischen Logik und Speicher Ressourcen in FPGAs. Die vorgeschlagene Architektur ist in einem FPGA implementiert zum Bestätigung ihrer Effizienz und Vergleich ihrer Eigenschaften mit denen von anderen Untersuchungen.

Zweitens wird es bemerkt, dass Suchtechniken in damit verbundenen Arbeiten werden in der Regel nicht für die optimale Interaktion mit dem Aggregationsstufe ausgedacht, was die Leistung der Klassifizierung negativ beeinflusst. Um dieses Problem anzugreifen, werden die bestehende Lookup Verfahren und ihre Anwendung in der Mehrfeld-Klassifizierung ausfürlich untergesucht. Auf dieser Basis schlagen wir vor generische Lookup Architekturen und Implementierungen die mit verschiedenen Rechenressourcen im FPGA durchgeführt werden, und dann bewerten wir ihre Leistung und ihre Wechselwirkung mit vorhandenen Aggregations Architekturen. Die Ergebnisse liefern Auswahlkriterien, die Lookup Schemas die am besten den verschiedene Aggregationsstufe anpasst zu wählen. Insbesonders bestimmen wir, welches Lookup Verfahren das geeignesten auf die aktuellen Bedürfnisse Regelung ist, das aber schwerwiegende Probleme bei der Aktualizierung des Regelwerks trifft. Eine Optimierung wird dann vorgeschlagen, die spezifisch darauf ausgerichtet ist, diese Probleme zu mildern.

Als Verallgemeinerung von dargestellten Architekturen und anschliessende Beitrag, Techniken für *Reduzierung der Komplexität* des Problems der Klassifizierung diskutiert werden, die ganz allgemein sind. Eigentlichen Regelwërke (Rulesets) werden auch untergesucht, um tatsächliche Komplexität der Klassifizierungsverfahren zu bewerten.

Diese Dissertation zielt darauf ab, zum einen, um eine neue theoretische Grundlage nach dem Stand der Technik aufzuweisen. Daraus werden dann Design-Kriterien gewieten, die ein besseres Verständnis der Verpflichtungen von Implementierung in FPGAs ermöglichen, was zu effizienteren Architekturen führt. Die Wirksamkeit dieser Implementierungen wird durch Versuchen auf rekonfigurierbare Logik Plattformen, die sich zunehmend in diesen Anwendungen eingesetzt werden demonstriert. Außerdem wird es gesucht, diese Analyse im Rahmen der Datennetze mit virtualisierten Ressourcen zu vorstellen, wo Schaltverarbeitung Probleme und neue Herausforderungen trifft.

Lista de acrónimos

ACL *Access Control List*, Lista de control de acceso

ALM *Arithmetic-Logic Module*, Módulo aritmético-lógico

AR *Arbitrary Range of values*, Rango arbitrario de valores

ARM *Advanced RISC Machine*, Máquina RISC avanzada

ATM *Asynchronous Transfer Mode*, Modo de transferencia asíncrona

ASIC *Application-Specific Integrated Circuit*, Circuito integrado de aplicación específica

BM *Best Match*, Mejor coincidencia

BS *Binary Search*, Búsqueda binaria

BV *Bit Vector*, Vector de bits

CAM *Content-Addressable Memory*, Memoria accesible por contenido

CIDR *Classless Inter-Domain Routing*, Enrutamiento entre dominios sin uso de clases

CoS *Class of Service*, Clase de servicio

DCFL *Distributed Crossproducting of (unique) Field Labels*, Producto cruzado distribuido de etiquetas (únicas) de campo

DCFLE *Extended Distributed Crossproducting of (unique) Field Labels*, Producto cruzado distribuido extendido de etiquetas (únicas) de campo

DCFV *Distributed Crossproducting of (unique) Field Values*, Producto cruzado distribuido de valores (únicos) de campo

DHP *Dynamic Hardware Plugin*, Plugin de hardware dinámico

DE *Delay Element*, Elemento de retardo

DMA *Direct Memory Access*, Acceso directo a memoria

DoS *Denial of Service*, Denegación de servicio

DPI *Deep Packet Inspection*, Inspección profunda de paquetes

ERM *Explicit Range Matching*, Comprobación explícita de rangos

EX *EXact Value*, Valor exacto

ETCAM *Extended Ternary Content-Addressable Memory*, Memoria ternaria accesible por contenido extendida

FPGA *Field-Programmable Gate Array*, Matriz de compuertas lógicas programables

FPX *Field-Programmable Port eXtender*, Extensor de puertos programable en campo

FSB *Front Side Bus*, Bus frontal

FW *Firewall*, Cortafuegos

GPP *General-Purpose Processor*, Procesador de propósito general

GR *General Range*, Rango arbitrario de valores de campo

{**HDL** *Hardware Description Language*, Lenguaje de descripción de hardware

HLL *High Level Language*, Lenguaje de alto nivel

HOL *Head Of Line*, Cabecera de línea

HPC *High Performance Computing*, Computación de elevado rendimiento

IND *INDexing*, Direccionamiento

IRQ *InteRrupt Request*, Solicitud de interrupción

IP *Internet Protocol*, Protocolo inter-red

IPC *IP Chain*, Cadena IP

IPv4 *Internet Protocol version 4*, Protocolo inter-red version 4

IQ *Input Queueing*, Almacenamiento (encolamiento) en puertos de entrada

LAB *Logic Array Block*, Bloque de matriz lógica

LAN *Local Area Network*, Red de área local

LUT *Look-Up Table*, Tabla de búsqueda

MAC *Media Access Control*, Control de acceso al medio

MAN *Metropolitan Area Network*, Red de área metropolitana

MM *Multi-Match*, Múltiples coincidencias

MB *Memory Bus*, Bus de memoria

NAT *Network Address Translation*, Traducción de direcciones de red

NIC *Network Interface Card*, Placa de interfaz de red

NIDS *Network Intrusion Detection System*, Sistema para detección de intrusiones en redes

TCAM *Ternary Content-Addressable Memory*, Memoria ternaria accesible por contenido

MPLS *Multi-Protocol Label Switching*, Conmutación basada en etiquetas multi-protocolo

NP *Network Processor*, Procesador orientado a aplicaciones de redes

OQ *Output Queueing*, Almacenamiento (encolado) en puertos de salida

PCI *Peripheral Component Interconnect*, Interconexión de componentes periféricos

PE *Processing Element*, Elemento de procesamiento

PLD *Programmable Logic Device*, Dispositivo de lógica programable

PLL *Phase-Locked Loop*, Lazo enganchado en fase

PPC *Parallel Packet Classification*, Clasificación paralela (o concurrente) de paquetes

PX *PrefiX value*, Valor de prefijo

QoS *Quality of Service*, Calidad de servicio

RAD *Reprogrammable Application Device*, Dispositivo de aplicación reprogramable

RFC *Recursive Flow Classification*, Clasificación recursiva de flujos

RISC *Reduced Instruction Set Computer*, Computadora de grupo de instrucciones reducido

RoS *Router on a Stick*, Enrutador de un puerto

RTL *Register Transfer Level*, Nivel de transferencia de registros

SB *Shared Bus*, Bus compartido

SM *Shared Memory*, Memoria compartida

SoC *System on Chip*, Sistema en un chip

SDS *Space Domain Switching*, Conmutación en el dominio del espacio

SMA *Sequential Memory Access*, Acceso secuencial a memoria

TCP *Transport Control Protocol*, Protocolo para control de transporte

TDS *Time Domain Switching*, Conmutación en el dominio del tiempo

TCAM *Ternary Content-Addressable Memory*, Memoria ternaria accesible por contenido

TTL *Time To Live*, Tiempo de persistencia

UDP *User Datagram Protocol*, Protocolo de datagramas de usuario (Capa 4 Modelo OSI)

UV *Unique Value*, Valor único (o particular)

UR *Unique Region*, Región única (o particular)

VDP *Virtual Data Plane*, Plano virtual de datos (o información)

VHDL *Very high-speed integrated circuit Hardware Description Language*, Lenguaje de descripción de hardware para circuitos integrados de alta velocidad

VLAN *Virtual Local Area Network*, Red virtual de área local

VM *Virtual Machine*, Máquina virtual

VN *Virtual Network*, Red virtual

VOQ *Virtual Output Queueing*, Almacenamiento (encolado) en puertos virtuales de salida

XPROD *Crossproducting*, Producto cruzado

XQ *Crossbar Queueing*, Almacenamiento (encolado) en la matriz de conmutación

WC *WildCard value*, Valor comodín

WUGS *Washington University Gigabit Switch*, Conmutador Gigabit de la Universidad de Washington

Índice general

Índice de figuras

Índice de cuadros

Capítulo 1

Introducción

1.1. Motivación

Las redes de datos han experimentado en los últimos años un rápido avance en cuanto a velocidad, así como profundos cambios en sus arquitecturas. En redes Ethernet, por ejemplo, las tasas de datos han pasado del orden de los 10 Mbps en la década de 1980 a tasas de 100Gbps en la actualidad [1]. Por otra parte, continuamente surgen nuevas aplicaciones que generan grandes volúmenes de datos, tales como la tele-presencia, streaming multi-media y alojamiento de datos en servidores de almacenamiento. Como indicador del impacto de estos servicios, el tráfico global de internet ha pasado de 100 Gbits por día en 1992 a 30000 Gbits por segundo en 2013; mientras que el tráfico IP *(Internetwork Protocol, IP)* anual se prevee que supere el zettabyte (1000 exabytes) a fines de 2016 [2]. Otro factor importante es la *consolidación* de servicios sobre redes Ethernet y la aplicación de estas redes en el contexto de redes extensas *(Wide Area Networks, WANs)* [3] [1]. En la actualidad, los enlaces mediante tecnología de fibra óptica permiten alcanzar las velocidades necesarias, mientras que el procesamiento del tráfico a nivel de paquetes representa una importante limitación.

1.1.1. Flujos de procesamiento en redes

Uno de los factores que más limitan la velocidad de procesamiento es su *granularidad*. En redes conmutadas por paquetes como es el caso de redes IP, cada paquete debe ser procesado independientemente. El tamaño de los paquetes es variable, mientras que la mayor exigencia en velocidad de procesamiento se da para paquetes de tamaño mínimo, que en el caso de redes IPv4 sobre Ethernet es 18 (mínimo campo Ethernet) + 8(padding, relleno) + 20(IPv4) + 20(TCP) = 64 bytes. Esto se traduce, para el caso de redes de 100 Gbps (100GbE), en $100e^9/(8{,}64) = 195312500$ paquetes por segundo (*pps*) o $1/192312500 = 5\ ns/paquete$. Esta exigencia puede reducirse agregando grupos

de paquetes en *flujos* de acuerdo a algún criterio pre-establecido; por ejemplo cuando los paquetes comparten iguales direcciones de origen y destino, requerimientos de latencia, de seguridad, etc.. En especial, en las redes de datos se identifican flujos llamados *elephant flows* (flujos elefante), los que en contraposición con los *mice flows* (flujos ratón) involucran la gran mayoría del tráfico [4]. Ya que el número de estos flujos es mucho menor que la cantidad de paquetes, el procesamiento necesario se alivia significativamente. Adicionalmente, la agregación en flujos permite implementar *ingeniería de tráfico* en forma flexible. Ejemplos de este tipo de agregación, con diversos objetivos, son las redes de área local virtuales *(Virtual Local Area Networks, VLANs)*, las redes privadas virtuales *(Virtual Private Networks, VPNs)*, virtualización de enlaces mediante conmutación multi-protocolo *(Multi-Protocol Label Switching, MPLS)*, y recientemente las redes definidas por software *(Software Defined Networks, SDNs)*.

La definición de flujos permite gestionar los recursos de procesamiento eficientemente. Para ello, sólo el primer paquete de un flujo se procesa completamente para determinar su tratamiento; los paquetes siguientes correspondientes al mismo flujo requieren procesamiento más simple y reciben el mismo tratamiento. En general, el procesamiento más complejo y lento del primer paquete se realiza en plataformas software; mientras que el procesamiento de los paquetes siguientes se descarga a plataformas hardware especializadas.

1.1.2. Virtualización de recursos

Otro problema que limita el procesamiento de paquetes en redes es la arquitectura propia de los equipos involucrados. En los últimos años, el hardware electrónico involucrado en las redes de datos así como los medios de enlace utilizados en tales redes han progresado notablemente, ofreciendo capacidades de transmisión y procesamiento adecuados a las necesidades de los servicios soportados. Sin embargo, las arquitecturas de gestión y control de tales redes limitan la explotación efectiva de tales recursos. Las técnicas de virtualización surgen como una solución a esta limitación, permitiendo adaptar dinámicamente el uso de los recursos y compartir los mismos entre múltiples aplicaciones. Esta última característica es comúnmente conocida como *virtualización de recursos*. Como una de tales técnicas, podemos mencionar la virtualización de servidores [5], la cual permite mejor aprovechamiento de tales equipos definiendo múltiples máquinas virtuales *(Virtual Machines, VMs)* sobre una única máquina física. Adicionalmente, los servidores virtualizados suelen utilizar placas de interfaz de red *(Network Interface Cards, NICs)* especiales a fin de agilizar la entrada/salida de tráfico entre las NICs virtuales de las VMs *(Virtual NICS, vNICs)* y la red a la que se encuentra conectado el servidor físico [6]. La virtualización de recursos también ha sido utilizada en el caso de los enrutadores, donde las vNICs ofrecen igualmente aceleración en hardware [7]. Estos enrutadores físicos alojan así varios enrutadores virtuales aprovechando características como el empleo de tablas de enrutamiento compartidas, migración de funciones entre VMs o entre hardware y software, u optimización del consumo energético [8], [9]. Como resultado de la integración creciente de estas técnicas en redes de datos, surge el concepto de configurar y administrar en forma flexible múltiples redes virtuales *(Virtual Networks, VNs)* sobre una o múltiples infraestructuras, provistas por uno o múltiples proveedores de infraestructura, siendo esto transparente para el proveedor de servicios

[10], [11]. Estas técnicas en conjunto brindan soporte a lo que se conoce actualmente como Redes Definidas por Software *(Software Defined Networks, SDNs)* [12]. Este concepto es amplio y promete aportar mejoras en diversos aspectos. Por un lado, se busca facilitar el ensayo de nuevos protocolos y arquitecturas que puedan aportar a la evolución de las redes de datos. Actualmente, debido a que las redes de datos deben prestar servicio sin interrupciones a la sociedad, esto es difícil de realizar sin interferir la operación normal de tales redes; esto es un fenómeno conocido como la "osificación de internet". Además, los dispositivos que procesan y enrutan los paquetes de datos contienen individualmente tanto las funciones de control como las de procesamiento, comúnmente conocidas como *planos de control y de datos*, en un esquema altamente distribuido. Esto también contribuye a la osificación de las redes de datos ya que dificulta mucho la configuración conjunta de los dispositivos para realizar investigación y desarrollo. Si bien existen soluciones para problemas particulares, tales como firewalls, Traductores de dirección de red *Network Address Translators, NATs)*, Redes Privadas Virtuales *Virtual Private Networks, VPNs)*, Circuitos virtuales en redes IP *(Multiprotocol Label Switching, MPLS)*, etc., todos ellos se incorporan en definitiva a redes que carecen de este problema de osificación, y sus objetivos pueden ser mejor implementados en redes concebidas con mayor flexibilidad. Las SDNs buscan dejar el plano de datos a cargo de cada dispositivo, e implementar el plano de control en forma centralizada. De esta forma, los dispositivos de procesamiento de paquetes pueden ser mas simples y veloces, mientras que el control puede ser más flexible y ágil. El protocolo OpenFlow [13] juega un rol fundamental en este paradigma, posibilitando la efectiva comunicación entre ambos planos y aportando gran flexibilidad en la especificación de tablas de ruteo; mientras que la Open Networking Foundation se encarga de su promoción, nucleando a número creciente de empresas [14].

1.1.3. Tecnologías

Para la implementación efectiva de los conceptos analizados, se cuenta en la actualidad con una variedad de tecnologías; ellas van desde las opciones de mayor desempeño y alto costo de implementación como los circuitos integrados de propósito específico *Application-Specific Integrated Circuits, ASICs)*, pasando por opciones de mejor compromiso desempeño/costo/flexibilidad tales como los procesadores orientados a aplicaciones de redes *(Network Processors, NPs)* hasta los procesadores de uso general *(General-Purpose Procesors, GPPs)* de bajo costo y moderado desempeño. Una tecnología que en los últimos años ha ganado popularidad, tanto para aplicaciones de computación de alta performance *(High Performance Computing, HPC)* como para aplicaciones en redes de datos, son los dispositivos de lógica programable *(Programmable Logic Devices, PLDs)*, concretamente las matrices de lógica reconfigurable *Field-Programmable Logic Arrays, FPGAs)*. Estos dispositivos ofrecen un compromiso muy conveniente entre flexibilidad, desempeño y costo. Ellos no sólo pueden contener uno o múltiples procesadores programables como lo sería una arquitectura basada en GPPs o NPs, sino que el hardware mismo puede adecuarse completamente a las necesidades de la aplicación y reconfigurarse incluso durante su funcionamiento. Al tratarse de dispositivos que no dependen exclusivamente de la ejecución secuencial de un programa, ofrecen desempeño no lejano al de un ASIC; mientras que su costo es mucho menor al de ellos ya que se reutiliza un mismo chip para gran variedad de aplicaciones. Cuando la velocidad no es un factor determinante, las plataformas basadas en GPPs

son una opción interesante por su bajo costo y buenas prestaciones para realizar simulaciones e implementaciones de moderado desempeño; mientras que las plataformas basadas en FPGAs permiten mejorar el desempeño de las arquitecturas propuestas así como proponer mejoras mediante diseños altamente paralelizables y optimizaciones en hardware. Como ejemplos sobresalientes de la aplicación de GPPs en plataformas flexibles de enrutamiento se puede mencionar el enrutador modular Click! [15] y el reciente OpenVRoute [16]; mientras que en el caso de FPGAs una plataforma sobresaliente es NetFPGA [17]. Al momento, existe abundante trabajo de I+D respecto a virtualización sobre estos dispositivos, entre ellos [18] y [19]. Finalmente, existen plataformas experimentales a nivel mundial implementadas en base a una combinación de las diversas tecnologías mencionadas [20] [21]. En el ámbito académico, a nivel de trabajos de grado y posgrado, se encuentra abundante actividad relacionada con esta temática, se pueden citar [22] y [23] como ejemplos. En nuestro país, si bien se registran trabajos de I+D en las áreas de diseño en FPGAs y redes de datos en forma separada, son escasas las iniciativas que integran ambas tecnologías como pretende hacerlo la presente propuesta. Como ejemplos de trabajo local sobre FPGAs podemos mencionar [24], [25], y [26]. Vistos estos antecedentes, junto con la relevancia de las redes de datos a nivel mundial, se pretende con el presente trabajo realizar un aporte en un área de reciente desarrollo en nuestro país.

1.2. Problemas de Investigación

Según lo expuesto, se observa que las redes actuales están experimentando una evolución hacia arquitecturas más flexibles, capaces de adaptarse a nuevos y más variados servicios. Sin embargo, para que estas implementaciones sean exitosas es necesario que las tecnologías asociadas sean suficientemente configurables y provean el desempeño necesario. En la actualidad, los dispositivos FPGA ofrecen un compromiso conveniente entre la flexibilidad propia del software y el desempeño inherente al hardware de propósito específico, por lo que son ampliamente adoptados en aplicaciones de red. Es así que, adoptando esta tecnología, el presente trabajo de Tesis Doctoral busca estudiar las necesidades de estas aplicaciones y, sobre esta base, realizar contribuciones en el área.

En primer término, se definen funciones de red genéricas y se analizan sus requerimientos mediante pruebas de concepto en FPGAs, integrando luego estas funciones a un caso real de procesamiento en redes virtualizadas. Sobre esta base, se identifican las funciones críticas para el desempeño global de las redes; profundizando sobre la de *clasificación*, la cual es especialmente exigente por la flexibilidad y desempeño requeridos. Se identifican tres casos particulares: clasificación mediante memorias ternarias accesibles por contenido *(Ternary Content-Addressable Memories, TCAMs)*, clasificación uni-dimensional, y clasificación multi-dimensional. Para cada uno de ellos, se analiza exhaustivamente el estado del arte y se proponen nuevas arquitecturas especialmente orientadas a implementación eficiente en FPGAs. Además, teniendo en cuenta las exigencias de las redes actuales y futuras, se considera para todas ellas el caso general de clasificación multi-coincidencia sobre rangos arbitrarios de valores. De esta forma, se realizan aportes que tienen el potencial de contribuir a la implementación efectiva de nuevas redes virtualizadas.

La presente Tesis de Doctorado busca responder al siguiente interrogante general:

¿Se pueden diseñar arquitecturas eficientes y flexibles de procesamiento para redes de datos en hardware con consumo moderado de recursos y elevadas prestaciones?

Como interrogantes específicos abordados, podemos mencionar:

¿Qué etapas básicas involucra una plataforma de procesamiento en redes?

¿Cuál es la etapa crítica en una plataforma de procesamiento en redes?

¿Existen implementaciones suficientemente eficientes para dicha etapa crítica?

¿Cómo se puede optimizar esta etapa crítica para las necesidades actuales y futuras?

1.3. Publicaciones relacionadas

En relación con el presente trabajo de Tesis, se generaron las siguientes publicaciones:

I. J.M. Finochietto, S. Paz, and C. Zerbini, "Hardware primitives for packet flow processing architectures, " VII Southern Conference on Programmable Logic (SPL2011), pp. 37-43, 2011.

II. C. Zerbini and J.M. Finochietto, "Reconfigurable Network Processing: The FPGA Case, " 12th Argentine Symposium on Technology (AST), 40th JAIIO, 2011.

III. C.A. Zerbini and J.M. Finochietto, "Performance evaluation of packet classification on FPGA-based TCAM emulation architectures, " 2012 IEEE Global Communications Conference (GLOBECOM 2012), pp. 2766-2771, 2012.

IV. C. Zerbini and J.M. Finochietto, "Multi-match packet classification on memory-logic trade-off FPGA-based architecture, " IEEE 14th International Conference on High Performance Switching and Routing (HPSR), pp. 121-127, 2013.

V. Carlos A. Zerbini and Jorge M. Finochietto, "Optimization of Lookup Schemes for Flow-Based Packet Classification on FPGAs, " International Journal of Reconfigurable Computing, vol. 2015, Article ID 673596, 31 pages, 2015.

En particular, la publicación III. fue citada en los siguientes trabajos:

I. W. Jiang, "Scalable ternary content addressable memory implementation using FPGAs, " 2013 ACM/IEEE Symposium on Architectures for Networking and Communications Systems (ANCS), pp. 71-82, 2013.

II. T. Ganegedara, W. Jiang, and V. K. Prasanna, "A scalable and modular architecture for high-performance packet classification, " IEEE Transactions on Parallel and Distributed Systems, vol. 25, no. 5, pp. 1135-1144, 2014.

III. S. S. Cercós, R. M. Ramos, A. C. Ewald Eller, M. Martinello, M. RN Ribeiro, A. M. Fagertun, and I. T. Monroy, "Design of a stateless low-latency router architecture for green software-defined networking, " International Society for Optics and Photonics SPIE OPTO, pp. 93880I-93880I, 2015.

IV. Z. Qian and M. Margala, "Low power RAM-based hierarchical CAM on FPGA, " IEEE International Conference on ReConFigurable Computing and FPGAs (ReConFig), pp. 1-4, 2014.

V. S. S. Cercós, R. E. Oliveira, R. Vitoi, M. Martinello, M. R. N. Ribeiro, A. M. Fagertun, and I. T. Monroy, "Tackling OpenFlow power hog in core networks with KeyFlow, " Electronics Letters vol. 50, no. 24, pp. 1847-1849, 2014.

1.4. Contribuciones realizadas

1.4.1. Hardware primitives for packet flow processing architectures

En este trabajo se identifican y analizan las funciones de procesamiento básicas presentes en arquitecturas de procesamiento en redes de datos. En particular, se definen las funciones necesarias para procesamiento de *flujos* de datos en tales redes. Con base en este estudio, se discute el grado de flexibilidad e interconexión deseable para utilizarlas en plataformas de procesamiento actuales. En particular, se proponen y describen en detalle arquitecturas básicas para su implementación en plataformas de hardware reconfigurable (FPGAs). A fin de evaluar su desempeño en un ejemplo de aplicación real, se implementa un caso de conmutación en redes LAN virtuales. Los resultados obtenidos permiten validar las funciones propuestas e identificar las etapas de procesamiento más críticas. En el contexto de la presente Tesis, esta contribución se trata en el Cap. 2.

1.4.2. Reconfigurable network processing: the FPGA case

Este trabajo amplía el análisis realizado en el anterior. Para cada primitiva de procesamiento definida anteriormente, junto con otras más generales introducidas en este trabajo, se aborda un estudio cualitativo enfocándose en las alternativas tecnológicas de implementación actuales. De particular interés es el caso de FPGAs, para el cual se presenta un ejemplo de implementación en redes LAN. En el contexto de la presente Tesis, esta contribución se trata en el Cap. 2.

1.4.3. Performance Evaluation of Packet Classification on FPGA-based TCAM Emulation Architectures

En los trabajos anteriores se identificó a la primitiva de *Clasificación* como la más crítica para el desempeño de las arquitecturas de procesamiento, por lo que en este trabajo se aborda esta función y su implementación de FPGAs. En particular, se analizan las opciones de implementación de TCAMs

mediante emulación en FPGAs. Durante esta implementación se identificaron características propias de la arquitectura FPGA que permiten, más allá de meramente emular el comportamiento de una TCAM, mitigar las limitaciones de las memorias TCAM nativas. A fin de demostrar estas ventajas, se analiza el problema de codificación de rangos, el cual es el principal limitante actual de las TCAMs. Finalmente, se validan y evalúan las arquitecturas propuestas mediante implementaciones en FPGAs. En el contexto de la presente Tesis, esta contribución se trata en el Cap. 3.

1.4.4. Multi-match Packet Classification on Memory-Logic Trade-off FPGA-based Architecture

En este trabajo se aborda el caso de clasificación multi-campo, para el cual los esquemas TCAM se vuelven ineficientes. Como se comprobó en los trabajos anteriores, las FPGAs son una plataforma muy conveniente para implementación de funciones de clasificación. Si embargo, las propuestas existentes se basan generalmente en el uso intensivo de recursos lógicos o de memoria, sin ser capaces de explorar compromisos entre ambos. Además, la mayor parte de ellos se orientan al caso de clasificación por *mejor coincidencia*, mientras que las aplicaciones actuales exigen cada vez más clasificación por *múltiples coincidencias*. Finalmente, muchas de ellas no son apropiadas para *actualización incremental* de reglas. En este trabajo se propone una nueva arquitectura que permite explorar diversas combinaciones de recursos lógicos en FPGAs según las necesidades particulares. Sobre la base de propuestas previas, se ubica en un caso intermedio no explorado anteriormente. En particular, nuestra arquitectura es naturalmente apropiada para implementar el caso de clasificación multi-campo por múltiples coincidencias y su actualización incremental es inherentemente simple. En el contexto de la presente Tesis, esta contribución se trata en el Cap. 4.

1.4.5. Optimization of lookup schemes for flow-based packet classification on FPGAs

En arquitecturas de clasificación por descomposición es muy importante la interacción entre las etapas de lookup y agregación, es decir que los resultados entregados por las etapas de lookup se encuentren en el formato más eficiente para ser procesados por la etapa de agregación. En trabajos anteriores esta interacción entre etapas no ha sido considerada, desarrollando las etapas de lookup y agregación como elementos auto-contenidos y optimizados sin tener en cuenta el contexto general de clasificación. Por otro lado, el procesamieto por flujos y la creciente complejidad de las redes actuales exige que dichas etapas cubran nuevos casos de clasificación no previstos en propuestas anteriores. En este trabajo se analizan estos factores y se propone una base general sobre la cual comparar objetivamente los esquemas de lookup existentes a fin de identificar sus compromisos de diseño, en especial para plataformas FPGA. Sobre esta base, y a partir de nuestras experiencias en implementación de TCAM emulada y arquitecturas de agregación en FPGA, se proponen esquemas de lookup modificados para su óptima interacción con las arquitecturas de agregación existentes. A fin de proveer criterios de selección, se brindan asismismo resultados estimados y de implementación en FPGAs para todos los esquemas propuestos. Finalmente, se identifica un esquema especialmente

conveniente para procesamiento de flujos, el cual actualmente plantea serios inconvenientes de implementación; se proponen entonces optimizaciones particulares a este respecto con lo cual facilitar su adopción en aplicaciones actuales. En el contexto de la presente Tesis, esta contribución se trata en el Cap. 5.

1.5. Estructura general

La Tesis ha sido dividida en 6 capítulos, de acuerdo al orden de las contribuciones. En el Capítulo 1 se presenta un estudio general de las necesidades de procesamiento en redes actuales, así como las soluciones propuestas a los problemas que éstas plantean. En el Capítulo 2, en tanto, se definen y analizan las funciones necesarias para implementar este procesamiento, introduciendo y comparando las tecnologías de implementación disponibles. El Capítulo 3 se enfoca en la función de clasificación, en particular mediante memorias TCAM emuladas en FPGAs. La extensión eficiente de estas arquitecturas al caso de clasificación multi-campo se explora en el Capítulo 4, proponiendo una nueva arquitectura enfocada a este caso de clasificación. En el Capítulo 5 se proponen esquemas eficientes y generales de lookup en un campo, los que pueden utilizarse para maximizar el desempeño de un esquema multi-campo. Finalmente, el Capítulo 6 reúne las contribuciones realizadas y presenta las conclusiones generales obtenidas, proponiendo otras optimizaciones como trabajo futuro.

Capítulo 2

Procesamiento en redes de datos

2.1. Motivación

En los últimos años, a partir de las necesidades prácticas de las redes de datos, múltiples trabajos han evaluado las limitaciones de los equipos de enrutamiento tradicionales y la posibilidad de agregar nuevas funciones de procesamiento en ellos. Los factores principales considerados son costo, desempeño, y flexibilidad[1]. Considerando estas exigencias, se abordan en este capítulo dos aspectos determinantes del procesamiento de paquetes en redes de datos: por un lado, se analizan las tecnologías disponibles en el presente y cómo ellas pueden combinarse de la forma más conveniente. Por otro lado, se analiza cuáles son las funciones de procesamiento necesarias en una red de datos y qué exigencias particulares plantea cada una de ellas a las tecnologías de implementación mencionadas en primer término. Con base en esta discusión, se presenta la primera contribución de la presente Tesis, la que permite apreciar y evaluar estos conceptos sobre una tecnología moderna.

2.2. Tecnologías

En esta sección, se repasan y comparan las tecnologías disponibles para realizar procesamiento en redes de datos, analizando los trabajos más relevantes para cada una de ellas. En la Fig. 2.1 se presentan tres de las tecnologías más relevantes a analizar en los próximos párrafos.

[1]Estos términos se mencionan aquí en un sentido general. En la secciones de evaluación de arquitecturas incluidas en esta Tesis, el término *costo* se refiere puntualmente al uso de recursos computacionales, mientras que el término *desempeño* se refiere a la cantidad de paquetes procesados por unidad de tiempo.

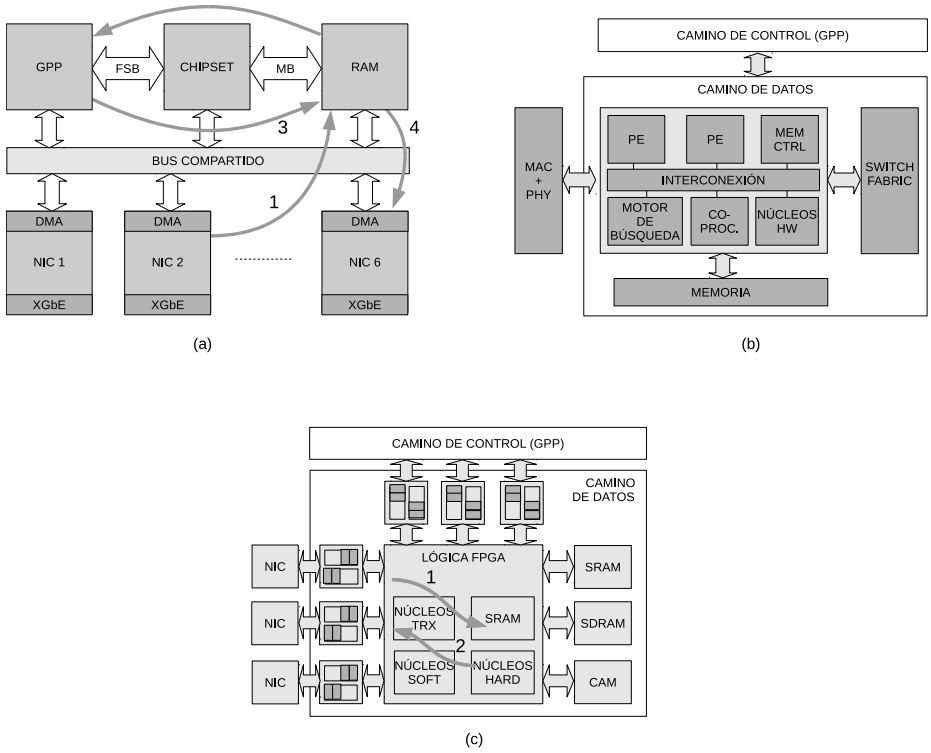


Figura 2.1: Tecnologías de procesamiento en redes de datos: (a) Procesador de propósito general, (b) procesador de redes, (c) lógica reconfigurable

2.2.1. Procesadores de Propósito General (GPPs)

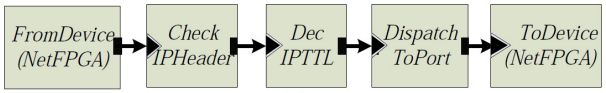
Un primer conjunto de trabajos, analizados al comienzo de la Tesis, evalúan la conveniencia de utilizar computadoras personales *(Presonal Computers, PCs)* para implementar enrutadores de datos. Estos equipos, basados en procesadores de propósito general (GPPs) utilizando en su mayoría en la arquitectura Von Newmann[2], son muy atractivos por su bajo costo debido al volumen de fabricación. Las implementaciones de routers sobre estas plataformas estaban originalmente basadas en software; podemos citar como ejemplos el router modular *Click!* para el plano de datos [15] y *Xorp* para el plano de control [27]. Debido a esta característica, sin embargo, dichas implementaciones están limitadas en su desempeño. La plataforma Click! [15] merece especial atención; desarrollada en MIT y originalmente destinada a la implementación de routers modulares, permite implementar una serie de configuraciones de procesamiento. Esta plataforma es una librería de objetos software codificados en lenguaje C++, los cuales pueden ser configurados e interconectados mediante el lenguaje de scripting *ned*. Click! puede ejecutarse como un proceso de usuario, para configuraciones generales; o como un proceso del kernel Linux, para aplicaciones que requieren buen desempeño. En la Fig. 2.2 se aprecian dos configuraciones típicas implementadas mediante módulos Click!.

Varios trabajos, como [28] y [29], establecieron los límites reales de desempeño de las plataformas basadas en GPPs. El primero de los trabajos citados utiliza el stack de enrutamiento de linux para

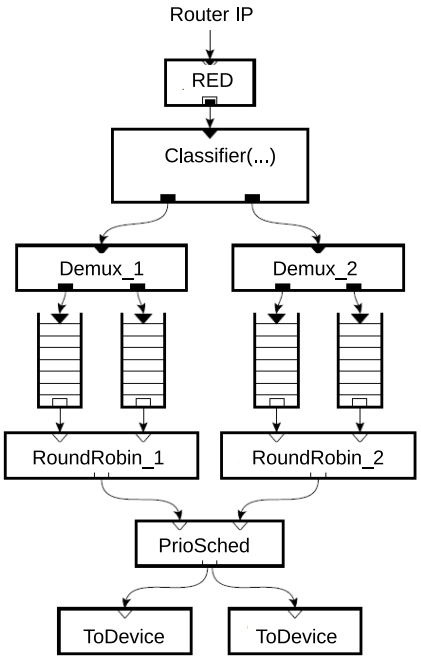
[2]La arquitectura de Von Neumann es una familia de arquitecturas de computadoras que utilizan el mismo dispositivo de almacenamiento tanto para las instrucciones como para los datos (a diferencia de la arquitectura Harvard).

```
//Instantiate components
src::FromDevice(NetFPGA);
checkip::CheckIPHeader();
ttl::DecIPTTL();
dispatch::DispatchToPort();
sink::ToDevice(NetFPGA);

//Interconnect component instances
src[out]->[in]checkip[out] ->[in]ttl[out]->
[in]dispatch[out]->[in]sink;
```



(a)



(b)

Figura 2.2: Módulos Click!: (a) camino simple de procesamiento, (b) balanceo de carga mediante combinación de módulos de *descarte (Random Early Detection, RED), encolado (Demux), y planificación (RoundRobin, PrioSched)*

realizar los ensayos, mientras que el segundo utiliza Click!. El procesamiento evaluado es el lookup de paquetes según prefijos en direcciones IPv4. Como se muestra en la Fig. 2.1(a), un sistema basado en un GPP consta de cuatro elementos principales: la unidad central de proceso *(Central Processing Unit, CPU)*, memoria de acceso aleatorio *(Random Access Memory, RAM)*, tarjetas de interfaz de red *(Network Interface Cards, NICs)* y el chipset que une las partes anteriores. La comunicación entre estos elementos se efectúa mediante los buses de interconexión de sistema *(System Bus o Front Side Bus, FSB)*, el bus de memoria *(Memory Bus, MB)*, y bus compartido *(Shared Bus, SB)*, en este caso el bus de periféricos *(Peripheral Component Interconnect, PCI)*. La CPU, representada por el GPP, es la encargada de procesar cada paquete individualmente, por lo que puede convertirse en un limitante, especialmente en el caso que se produzcan muchos fallos de memoria cache (cache misses) durante el procesamiento. Como se ve en la Fig. 2.1(a), se requieren 4 transacciones para completar el procesamiento de un paquete desde que ingresa, por ejemplo, a la NIC 2 hasta su despacho, por ejemplo, por la NIC 6. El bus PCI/PCI-X/PCI Express es compartido entre múltiples periféricos y la memoria a través del chipset, donde sólo un dispositivo a la vez puede actuar como maestro. Además de los propios datos transferidos, este bus involucra transacciones adicionales de control, tales como latencia de arbitraje entre dispositivos, ciclos de direccionamiento/atributos, y estados de espera; esto limita notablemente su tasa de transferencia de datos, especialmente para paquetes mínimos. Las NICs poseen buffers temporales para los paquetes recibidos y transmitidos, y transfieren datos mediante acceso directo a memoria *(Direct Memory Access, DMA)* desde y hacia las colas de recepción reservadas por el CPU en RAM; sus limitaciones residen principalmente en la heterogeneidad entre distintos fabricantes y su incapacidad de procesar el contenido de los pa-

quetes antes de enviarlos a RAM. Un problema común en versiones antiguas de los controladores asociados a las NICs consiste en su funcionamiento exclusivamente basado en interrupciones desde la NIC, esto producía el problema de *livelock* donde el proceso que atiende las interrupciones está tan ocupado haciéndolo que no puede leer efectivamente los paquetes almacenados en la NIC [30]. Este problema se mitiga notablemente si, a partir de una interrupción por paquetes disponibles, se deshabilitan las interrupciones por hardware y se comienza un ciclo de polling hasta descargar el buffer de la NIC; este método fue introducido en versiones mas recientes de Linux. Click!, por su lado, permite asimismo utilizar polling puro, el cual en casos de arribo frecuente de paquetes mejora aún más el rendimiento de transferencia y permite implementar el mecanismo de reserva de RAM llamado *buffer recycling*, más eficiente que el utilizado por el stack de red de Linux. [29], en particular, exploró diferentes combinaciones de transmisión, recepción y forwarding de paquetes sobre diferentes configuraciones de Click!. Como resultados, se observó que el GPP era capaz de sostener holgadamente el procesamiento de paquetes Ethernet mínimos a velocidad de línea en redes Gigabit Ethernet; sin embargo el bus PCI-X y las tarjetas NIC no soportaban la transferencia de paquetes mínimos a esa velocidad transformándose en los principales cuellos de botella de la arquitectura. Trabajos posteriores [31] buscan integrar servidores y enrutadores en plataformas GPP, así como explotar múltiples servidores y múltiples GPPs (cores) por servidor para paralelizar la tarea de estas plataformas software, alcanzando velocidades de 35 Gigabits por segundo (Gbps). En este caso, los principales cuellos de botella se detectaron en el Front Side Bus que conecta el GPP con la memoria RAM, así como dentro del GPP mismo.

Otro conjunto de trabajos sobre arquitecturas de conmutación y procesamiento de paquetes basada en software surgió junto con la amplia difusión de las técnicas de *virtualización* que tuvo lugar en los pasados años. Tales técnicas introdujeron un nuevo escenario en los esquemas de red, creando una nueva capa de acceso que interconecta las múltiples máquinas virtuales (VMs) que residen en un host físico. La virtualización de recursos introduce gran flexibilidad a las arquitecturas de red al estar éstas ahora definidas mediante software, independientemente de los hosts físicos. Sin embargo, estas nuevas características no pueden ser aprovechadas en redes tradicionales basadas en protocolos ethernet y TCP/IP, debiendo definirse nuevos criterios de enrutamiento. Uno de los primeros trabajos que consideran estos criterios es OpenVSwitch [32]; OpenVRoute [33], en tanto, se concentra en la implementación de *routers virtuales* utilizando la base de OpenVSwitch. OpenVRoute implementa las conexiones necesarias entre los recursos lógicos y físicos de un router físico para montar sobre él múltiples routers virtuales en forma eficiente. La conexión entre los componentes se realiza mediante el protocolo *OpenFlow*. De este modo, se logra un plano de forwarding dividido en dos capas; una de ellas transporta flujos de alto volumen *(elephant flows)* a alta velocidad mediante un switch externo interconectado mediante OpenFlow, mientras que la otra transporta flujos de bajo volumen *(mice flows)* mediante OpenVSwitch en software. En la actualidad, ambas arquitecturas son de gran importancia en un contexto donde la virtualización de recursos es ampliamente adoptada.

2.2.2. Circuitos integrados de aplicación específica (ASICs)

Los equipos de enrutamiento comerciales utilizan en su mayoría este tipo de tecnología, ya que ofrece óptima relación entre desempeño y costo de hardware. Los ASICs cuentan con gran cantidad de bloques lógicos, extensivamente optimizados para el menor costo de hardware y máxima velocidad con mínimo consumo. Sin embargo, por lo general este tipo de tecnología requiere extensos ciclos de desarrollo, lo que dificulta su uso en plataformas experimentales y su adaptación a nuevas funciones de una red de datos. Su alto costo de desarrollo debe ser amortizado por grandes volúmenes de producción y comercialización a costos reducidos. Típicamente estos dispositivos ofrecen oportunidades de programación bastante reducidas, pudiendo ajustarse solo algunos parámetros de funcionamiento.

2.2.3. Procesadores para aplicaciones de redes (NPs)

Los NPs buscan integrar el alto desempeño de los ASICs con la programabilidad propia de los GPPs [34]. Estos procesadores son arquitecturas especializadas, de muy alto paralelismo, generalmente basadas en múltiples procesadores de tipo RISC. En base al estudio de las propiedades de las funciones de red, estas arquitecturas las distribuyen entre múltiples cores o elementos de procesamiento *(Processing Elements, PEs)*. Algunas de las técnicas comúnmente utilizadas son el pipelining intensivo (deep pipelining), sets de instrucciones especializados, unidades especializadas en tareas de redes, y técnicas multi-hilo (multi-threads). Como ejemplos de estos dispositivos podemos citar los llamados Network Flow Processors NFP-6xxx de Netronome (2013). Estos NPs integran mas de 3B de transistores de tecnología Intel 3-D en 22nm, trabajando a 1.2 GHz para una tasa de procesamiento de 200 Gbps. Entre otras características [35], estos dispositivos cuentan con 120 núcleos de procesamiento de flujos *(Flow Processing Cores, FPCs)*, 96 núcleos de procesamiento de paquetes *(Packet Processing Cores, PPCs)* y memoria de 400/7400 Gbps de velocidad de acceso. Si bien su desempeño es muy bueno, la desventaja de estos dispositivos es la dependencia que se establece respecto al fabricante cuando se adopta uno de ellos para la implementación particular. Las interfaces de programación ofrecidas suelen ser propietarias y dependen del dispositivo en particular, por lo que la eventual migración de un dispositivo a otro puede ser un inconveniente. Esto limita mucho la portabilidad de los diseños basados en NPs.

2.2.4. Circuitos integrados de lógica programable (FPGAs)

En los últimos años, la tecnología de hardware reconfigurable ha sido ampliamente adoptada en diversos campos de aplicación tales como comunicaciones digitales, sistemas de control, sistemas de medición, procesamiento de imágenes, y hasta sistemas de radio-frecuencias [36]. El hardware reconfigurable abarca diversos dispositivos, tales como los dispositivos lógicos programables complejos *(Complex Programmable Logic Devices, CPLDs)* y los arrays de compuertas programables en campo *(Field-Programmable Logic Arrays, FPGAs)*; siendo estos últimos los más potentes. El desempeño de un FPGA se aproxima mucho al de un ASIC, mientras que sus posibilidades de re-programación/reconfiguración son mucho mayores. Los FPGAs han avanzado notablemente en escala de integración,

Cuadro 2.1: Comparación de tecnologías

	GPPs	FPGAs	ASICs	NPs
Costo	Bajo	Medio	Alto	Medio
Paralelismo	Limitado	Alto	Alto	Variable
Programabilidad	Buena	Medio	Baja/nula	Media
Desempeño	Bajo	Bueno	Muy bueno	Variable
Tiempo de desarrollo	Corto	Medio	Largo	Medio
Consumo	Alto	Medio	Bajo	Alto
Lenguajes	Standard	Standard/Ad Hoc	Ad Hoc	Standard/Propietario
Interface	Abierta	Abierta	Propietara	Propietaria

en velocidad de procesamiento, y en optimización del consumo de energía. Estos dispositivos integran, además de los bloques lógicos configurables, múltiples núcleos de propósito específico tales como bloques DSP, transceptores seriales, interfaces serie/paralelas de memoria, múltiples núcleos ARM; de esta forma un FPGA es capaz de alojar un System-on-chip (SoC) completo. La propiedad más destacable de los FPGAs es su posibilidad de *re-configuración*, es decir cambio del hardware instanciado; y *re-programación*, es decir cambio del comportamiento de dicho hardware. Combinando estas técnicas, se pueden realizar cambios tanto al momento de compilar un nuevo diseño como durante el funcionamiento, logrando un balance muy conveniente entre flexibilidad de modificación y desempeño.

Los principales problemas que enfrentan los FPGAs son su alto consumo de energía, por un lado; y la necesidad de contar con competencias especiales para su programación, por el otro. El consumo de los FPGAs ha sido muy optimizado en los ultimos años; por ejemplo los últimos FPGAs Stratix 10 de Altera han logrado reducir el consumo hasta un 70 % respecto a sus predecesores. La complejidad de uso, por otro lado, se relaciona con la necesidad de dominar Lenguajes de Descripción de Hardware *(Hardware Description Languages, HDLs)* específicos tales como Verilog o VHDL para codificar la arquitectura diseñada. Esto dificulta mucho la interacción entre especialistas en software, electrónica y redes. Los lenguajes de alto nivel *(High Level Languages, HLLs)* buscan solucionar este problema, adoptando dialectos de lenguajes populares en programación como por ejemplo C para describir la funcionalidad deseada. Dicha descripción, junto con un conjunto de directivas de compilación, es procesada por herramientas de síntesis de alto nivel a partir de las cuales se obtiene automáticamente la descripción en HDLs. Estas herramientas, si bien se encuentran en desarrollo, prometen mejorar el problema de programación en FPGAs [37]. Finalmente, existen actualmente librerías que extienden el concepto de *open software* mediante el de *open hardware*; el código sintetizable disponible en ellas se conoce como *gateware*. Estas librerías ofrecen gran cantidad de módulos especializados; asimismo existen proyectos que ofrecen el código de implementación en forma abierta.

2.2.5. Procesamiento asistido por hardware

Uno de los primeros proyectos en considerar FPGAs para procesamiento en redes fue el Extensor de Puertos Programable *(Field Programmable Port Extender, FPX)*, desarrollado a partir de de 2000 por la Universidad de Washington en St. Louis [38]; este proyecto derivó en el desar-

rollo de módulos más generales llamados extensiones dinámicas de hardware *(Dynamic Hardware Plugins, DHPs)* [39]. FPX es una plataforma de hardware abierta y reconfigurable, utilizada para implementar funciones de procesamiento de paquetes a velocidades OC-48 (2488,32 Mbit/s). Su objetivo es experimentar con nuevas funciones de procesamiento de paquetes, permitiendo construir un enrutador que evolucione junto con las aplicaciones. Las placas utilizadas incluyen dos FPGAs, una de ellas *(Reprogrammable Application Device, RAD)* utilizada para las funciones propiamente dichas, y la otra *(Network Interface Device, NID)* utilizada como auxiliar para reprogramar la RAD a través de la red. Estos dispositivos se incluyen en tarjetas que se interconectan a los puertos de un Switch *(Washington University Gigabit Switch o WUGS)*, formando en conjunto un enrutador. Este proyecto generó abundante trabajo sobre arquitecturas de lookup, protocolos, reconfiguración parcial (DHPs), así como aplicaciones para obtención de estadísticas de tráfico, Deep Packet Inspection (DPI), firewalling, streaming de video, etc.. La Fig. 2.3 muestra la interconexión física de múltiples placas FPX al switch WUGS.

Otro de los primeros trabajos en adoptar FPGAs en redes fue el router MIR *(Mixed-version IP Router)* [40], el cual utilizaba un FPGA Xilinx Virtex-II Pro conteniendo hasta cuatro núcleos PowerPC y lógica de propósito general para implementación de funciones hardware. El objetivo básico del diseño era permitir que funciones muy frecuentes y poco complejas sobre paquetes IPv4 pudieran ejecutarse en lógica de alto desempeño, mientras que casos especiales como paquetes IPv6 se derivaban directamente a los procesadores PowerPC. De esta manera, se demostró una interacción software/hardware que no había sido posible previamente.

Los trabajos mencionados establecieron las bases para desarrollos posteriores, que adoptarían un esquema donde el software (sobre un GPP) y el hardware (sobre FPGAs) interactúan, dando lugar a arquitecturas de procesamiento mixtas. En general, la sección basada en GPP ofrece máxima flexibilidad con performance limitada, por lo que es apta para funciones de ejecución poco frecuente y tareas de gestión que requieren procesamientos complejos. La sección basada en FPGAs, en tanto, ofrece máxima performance para tareas que deben realizarse sobre cada paquete entrante, con complejidad de procesamiento moderada y determinística. Además, el GPP tiene la capacidad de migrar tareas a hardware en caso de volverse éstas muy frecuentes; y el FPGA puede hacer lo propio hacia el GPP cuando una función ya no es necesaria en hardware. Finalmente, al aliviar la carga del GPP, éste puede soportar otras tareas como por ejemplo la implementación de servidores; todo ello sobre la misma plataforma hardware.

Posteriormente, la Universidad de Rice desarrolló una NIC basada en hardware reconfigurable [41], particularmente la FPGA Virtex II Pro de (en ese momento) Avnet, con arquitectura abierta y orientada al uso en investigación y educación. Su objetivo principal es la evaluación de nuevas arquitecturas en servidores sobre prototipos reales en lugar de simulación. La simulación en arquitecturas de computación suele ser imprecisa debido a la interacción entre múltipes sistema asícronos entre sí. Las NICs disponibles hasta ese momento carecían de las capacidades de procesamiento, memoria y flexibilidad para ensayar esquemas de procesamiento en redes, por lo que este trabajo significó un importante paso como plataforma de soporte. Paralelamente, y con el surgimiento de GPPs con múltiples núcleos y las técnicas de virtualización, los principales fabricantes también desarrollaron múltiples modelos de NICs incorporando funciones de procesamiento y múltiples colas de espera

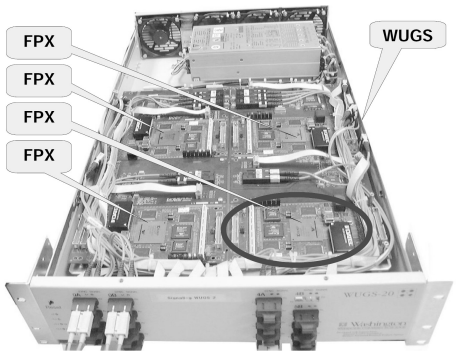


Figura 2.3: Múltiples Field-programmable Port Extenders conectados al switch WUGS

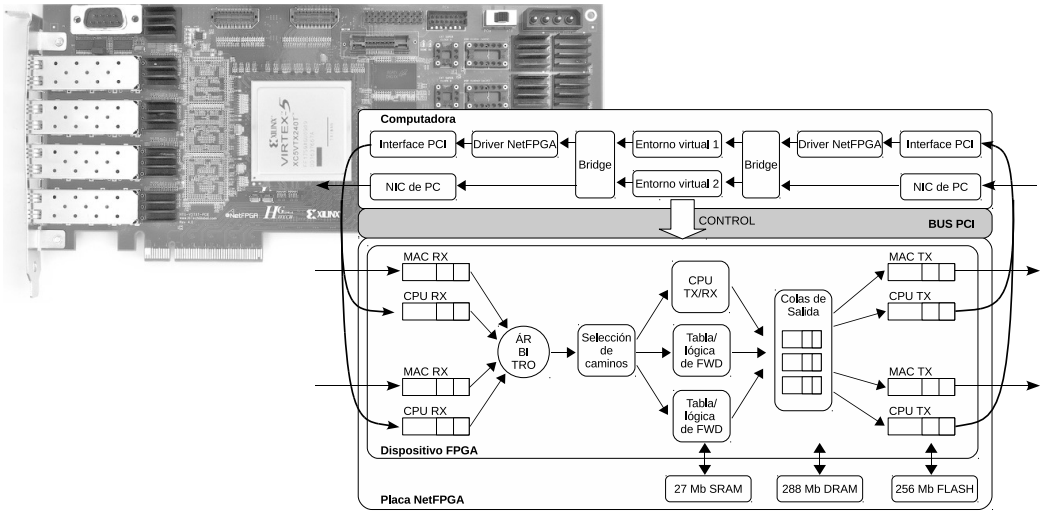


Figura 2.4: Plataforma reconfigurable para aplicación en redes NetFPGA

(First In, First Out, FIFO) con sus interrupciones *(Interrupt Requests, IRQs)* asociadas [42] [43]. El objetivo de estos recursos es asociar CPUs a colas de espera en las mismas NICs, aliviando la carga sobre las memorias caché de los núcleos y la memoria RAM del sistema.

Otra platafoma que integra FPGAs en placas de red para fines de desarrollo y educación es NetFPGA [44]. Este proyecto surgió en la Universidad de Stanford en 2008, implementando una plataforma con 4 interfaces Gigabit Ethernet y FPGA Xilinx Virtex II Pro, y actualmente se encuentra en plena vigencia con una versión de 4 puertos 10 Gigabit Ethernet y FPGA Xilinx Virtex 5. Esta plataforma también se implementó en FPGAs de Altera en el proyecto DE4 NetFPGA [45]. Consta de la plataforma hardware, el código RTL del datapath básico para interconectar puertos y una amplia librería de funciones de procesamiento aportadas por diferentes grupos de investigación, a las que se pueden sumar los propios. En la Fig. 2.4 se puede apreciar el aspecto y arquitectura base de este proyecto.

El proyecto HERO *(High-speed Enhanced Routing Operation)* [7], por su parte, desarrolló un IP core para FPGAs Altera Stratix GX montadas en NICs que maneja tanto la interfaz PCI-X hacia el

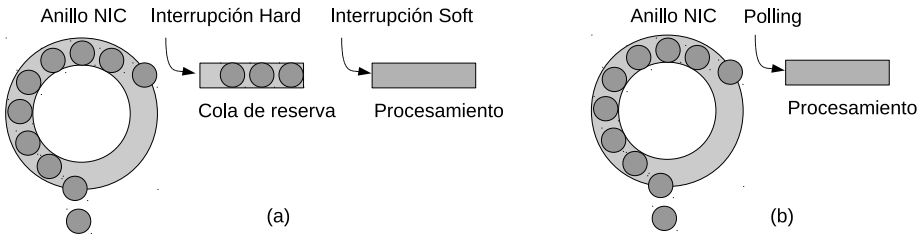


Figura 2.5: Interfaz NIC-sistema: (a) procesamiento por interrupciones, (b) procesamiento por interrupciones + encuesta (polling)

GPP como la interfaz Ethernet hacia la red de interconexión. En particular, implementa mecanismos para definir dos caminos de datos; el *slow path* que siguen todos los paquetes a ser procesados en el GPP, y el *fast path* destinado a transacciones entre NICs a través del bus PCI-X sin pasar por la memoria del sistema. Para ello, utiliza colas múltiples en cada NIC y funciones de *clasificación* para definir el destino de los paquetes entrantes. Este trabajo se destaca por lo detallado de su análisis y la extensiva evaluación de su desempeño en el contexto de un sistema mixto completo.

Otro trabajo que estableció los nuevos límites de desempeño de un enrutador basado en GPPs asistidos por hardware es [46]. En este trabajo, se utiliza Linux sobre una arquitectura PC con múltiples canales de DMA, buses PCI Express duales y NICs 10 GbE de dos puertos basadas en el chipset Intel 82598 [43]. Durante estos experimentos se comprobó que, aún sin utilizar las características especiales de las NICs, se puede soportar velocidad de 10 Gbps para un único flujo de paquetes de tamaño ≥ 250 bits. Utilizando las características del procesamiento asistido por hardware, se pueden soportar múltiples flujos a 10 Gbps con tráfico de características reales. Mediante este trabajo se comprobó que, utilizando las nuevas capacidades incluidas en las NICs y los múltiples núcleos de los GPPs actuales, un router basado en plataformas de costo moderado puede competir con equipos comerciales de alto desempeño basados en ASICs. Algunos conceptos clave para aprovechar estas características, que no profundizaremos aquí, son el correcto manejo de afinidad de interrupciones (interrupt affinity), asignación conveniente de canales DMA (DMA allocation), y drivers optimizados para las nuevas NICs con soporte de nueva API *(New API, NAPI)* que soluciona el efecto de Receive Livelock de drivers previos, como se ilustra en la Fig. 2.5. Es asimismo importante la correcta gestión de los cores del GPP, la selección de las NICs, memoria, y buses de interconexión. El software, en tanto, debe mantener accesos a memoria locales evitando el uso de bloques compartidos entre núcleos que llevarían a fallos de cache degradando el desempeño. Otro aspecto clave es la correcta distribución de la carga de datos entre las diversas memorias y núcleos del GPP; aquí juega un rol fundamental el soporte de clasificación de paquetes y múltiples colas de espera (rings) en TX/RX provisto por las NICs (por ejemplo mediante hashing basado en Receive Side Scaling o RSS). Como se menciona en [46], mediante las múltiples colas de las NICs la carga de tráfico entrante es distribuida via canales DMA separados hacia diferentes memorias y núcleos; esta distribución se hace en base a hashing de los encabezados de los paquetes entrantes. Esto provee soporte para funciones de virtualización y es esencial para el desempeño del sistema, ya que permite paralelizar el procesamiento de paquetes. Como ya comentamos, esta clasificación mediante hashing, si bien es básica, permite ya descargar decisiones de forwarding a hardware. Inclusive algunas NICs, como Neptune 10Gb de Sun Microsystems (actualmente licenciada a Marvell e incluida como

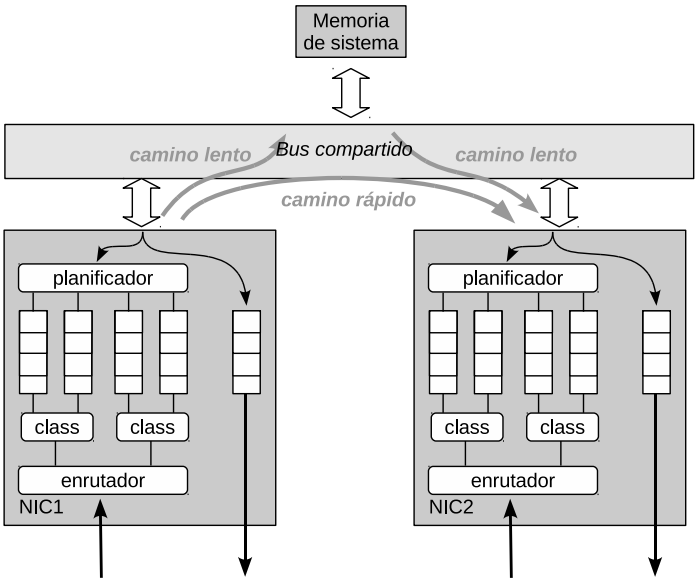


Figura 2.6: Aceleración del procesamiento mediante NICs inteligentes

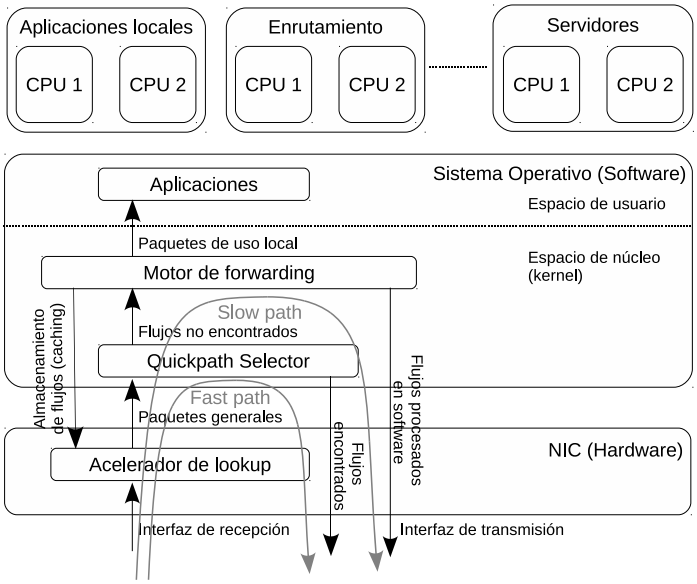


Figura 2.7: Arquitectura asistida por hardware para distribución de carga en un router/server combinado

Network Interface Unit o NIU en el procesador UltraSPARC T2), proveen memoria TCAM la que como veremos permite clasificación mucho más granular y configurable que la provista por las funciones hash de las NICs Intel. Mediante las funciones de clasificaión también se puede implementar calidad de servicio (QoS), filtrado de tráfico, captura de tráfico y gestión de flujos con estado, como se verá más adelante al abordar las funciones de procesamiento.

Dada la flexibilidad y desempeño demostrados por las plataformas GPP asistidas por hardware inteligente, en conjunto con las técnicas de virtualización, algunos trabajos se han concentrado en implementaciones donde se combinan enrutadores con otras funciones en la misma plataforma.

En particular [47] propone la implementación de un router y servidor combinado basándose en su trabajo previo [48]. Una combinación router/server es útil por ejemplo en datacenters, donde cada servidor actúa a la vez como host y como relay de otros servidores; o en una red comunitaria donde un servidor local almacena contenido de uso frecuente para optimizar la velocidad de acceso. Este trabajo utiliza placas Intel 82599 con múltiples colas de TX/RX así como un clasificador on-board llamado Flow Director. Mediante un módulo especialmente diseñado e incluido en el kernel de Linux, llamado Quickpath Selector, implementa forwarding de paquetes directamente entre NICs, en lo que se llama el *camino rápido (fast path)*. Los flujos que no pueden ser derivados hacia el fast path son procesados mediante el conmutador *Open vSwitch* basado puramente en software [49], definiendo lo que se llama el *camino lento (slow path)*. Finalmente, los paquetes destinados a servidor(es) alojados en la PC host se derivan al espacio de usuario para su procesamiento local.

Un aspecto importante de este esquema reside en lograr el correcto balance entre las tareas de servidor y enrutamiento, sin que una interfiera en la otra. Para ello, los núcleos se asocian a las interrupciones de las NICs pero sólo algunos de ellos se encargan de hacer ruteo, mientras que los demás cubren tareas del servidor. A medida que la carga de ruteo aumenta, se redistribuyen las tareas de los núcleos logrando el balance deseado. En la Fig. 2.6 se observa una arquitectura de conmutación mixta que combina tráfico entre NICs inteligentes (fast path) y tráfico derivado para procesamiento en software (slow path); en ella se observan los módulos típicos necesarios así como la presencia de múltiples colas de espera en cada NIC. La Fig. 2.7, en tanto, ilustra en mayor detalle los módulos involucrados en la implementación router/servidores de [47].

2.3. Esquemas mixtos

Con el advenimiento de las redes virtualizadas, los planos de control y de datos que anteriormente se hallaban presentes como partes integrantes de cada equipo de conmutación, se pueden separar totalmente e implementarse en forma distribuida sobre plataformas diferentes. En general, el plano de control se implementa comunmente en software dada la complejidad de las operaciones involucradas, mientras que el plano de datos se diseña mediante hardware específico para máximo desempeño; esto puede verse como una generalización del procesamiento asistido por hardware. El proyecto OpenFlow [13], iniciado en la Universidad de Stanford, fue una de las primeras iniciativas en este sentido. OpenFlow busca combatir la *osificación de internet*, facilitando la experimentación sobre redes existentes sin afectar su servicio normal. Para ello, explota las tablas de flujos que ya contienen la mayoría de los equipos de conmutación y propone una interfaz standard, implementada mediante software, que permite definir y actualizar flujos experimentales sin afectar los ya presentes. La nueva interfaz sería la base para la separación de planos de control y de datos adoptada en SDNs. En este trabajo, los equipos de conmutación podían ser switches comerciales, basados en ASICs, o NPs; o plataformas diseñadas en torno a FPGAs.

Posteriormente, Casado et. al. [50] adoptan estas ideas y estudian cómo debería ser la interfaz entre software y hardware. En el mecanismo propuesto, el componente de software toma las primeras decisiones sobre paquetes no identificados y almacena los resultados en hardware; así, todos los

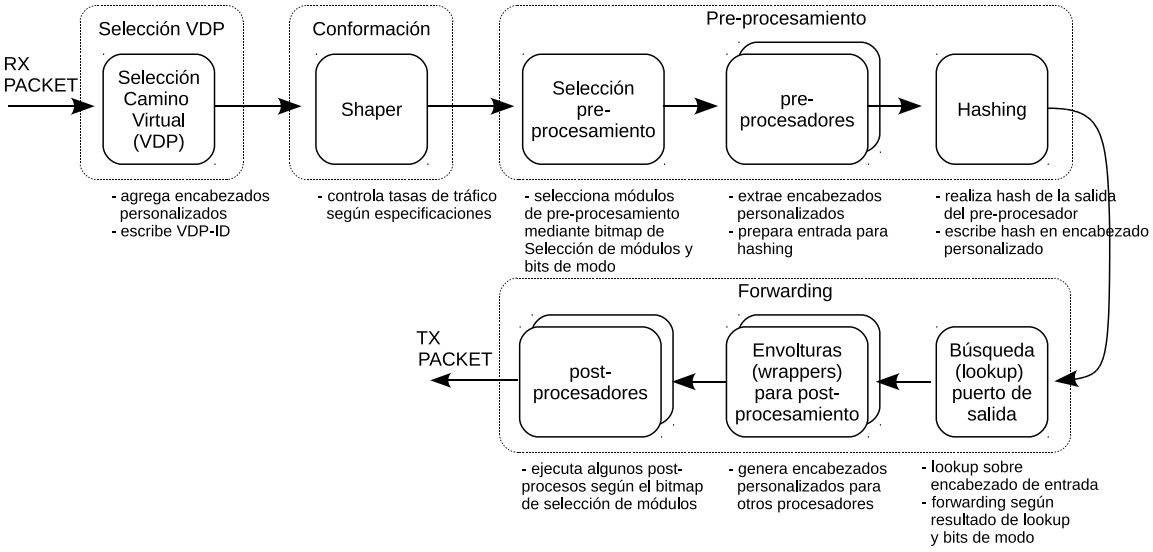


Figura 2.8: Switchblade: pipeline general de procesamiento

paquetes futuros que corresponden con esta decisión serán directamente procesados en hardware. La ventaja de este método reside en que las decisiones sobre paquetes desconocidos suelen ser complejas, para lo que el software es lo más apropiado. Una vez que este paquete es procesado, se asocia a él una determinada *acción* y se almacenan en hardware (a) una regla de clasificacion, y (b) la acción determinada en software. Posteriormente muchos otros paquetes pueden arribar compartiendo sus características, por lo que son tratados en hardware mediante la rápida consulta de la tabla de flujos y tratados según su acción asociada. En definitiva, el software captura la complejidad de determinar acciones (en base a métricas pre-definidas para el tratamiento del flujo), mientras que el hardware se mantiene simple y veloz. SwitchBlade [51] lleva estas ideas a la práctica mediante una implementación sobre NetFPGA. Para ello, define las llamadas *software exceptions*; es decir, el tráfico entrante por los puertos de NetFPGA es procesado en hardware; de no identificarse como perteneciente a algún flujo, se deriva hacia un esquema software donde se toma la decisión correspondiente. El diseño hardware se basa en *planos virtuales de datos (VDPs por su nombre en inglés)* por donde se derivan los paquetes, definiendo así *caminos de procesamiento*, todos ellos implementados mediante pipelines para obtener máximo desempeño. Los VDPs son definidos dinámicamente, habilitando las funciones de procesamiento correspondientes mediante un módulo llamado *VDP selector*. El pipeline general de procesamiento definido se ilustra en la Fig. 2.8. Este último trabajo nos lleva a nuestro próximo análisis en el contexto más general de procesamiento de paquetes.

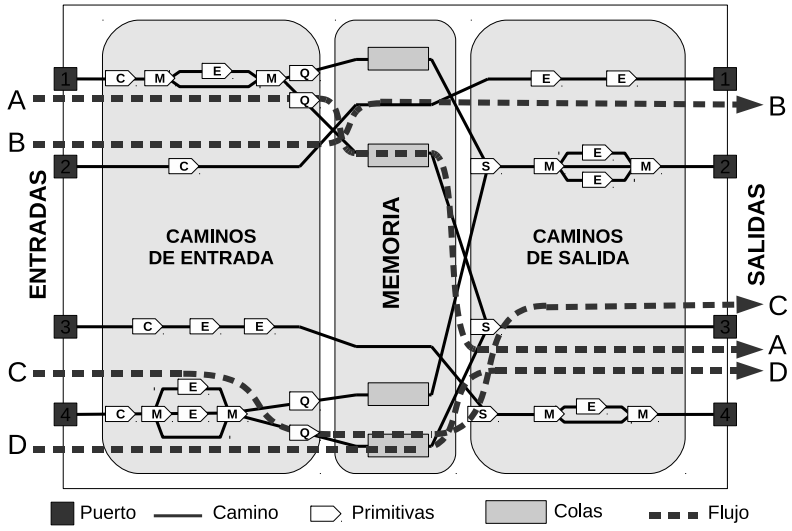
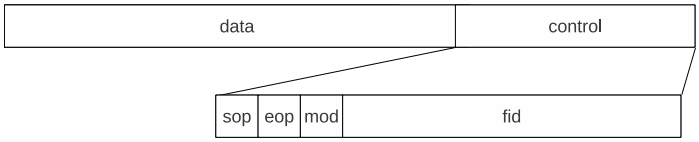
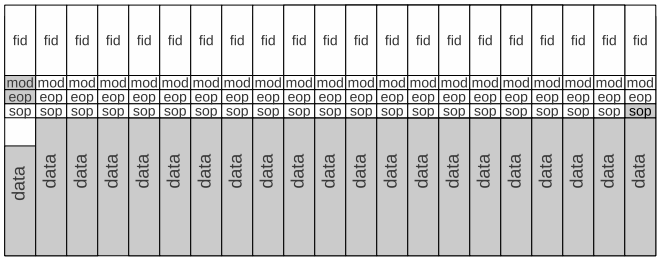


Figura 2.9: Modelo de caminos de procesamiento



(a)



(b)

Figura 2.10: Estructura de un paquete para procesamiento de flujos

2.4. Arquitectura genérica para procesamiento de paquetes

2.4.1. Caminos de procesamiento en redes de datos

A fin de abordar las necesidades de procesamiento en las redes de datos e identificar claramente las funciones necesarias, planteamos en esta sección un modelo general ilustrado en la Fig. 2.9. En este esquema, se consideran cuatro flujos *A, B, C* y *D* arribando a los puertos de entrada *1, 2, 3* y *4* respectivamente. Cada flujo se procesa primero en una etapa de entrada y pasa por una etapa de conmutación, para ser procesado nuevamente en una etapa de salida y finalmente abandonar el sistema a través de un puerto de salida. En este proceso, puede darse el problema de *contención*

donde dos paquetes llegan desde la etapa de conmutación al mismo puerto de salida en el mismo instante. Para solucionar este problema se incluye una etapa de memoria, en la forma de colas de espera (Queue, First-In First-Out memory o FIFO), que desacopla efectivamente las etapas de entrada y salida. En la Fig. 2.9, por ejemplo, los flujos A, C y D podrían competir por el mismo puerto de salida, por lo que sus paquetes se mantienen en la etapa de memoria hasta que puedan ser efectivamente transmitidos. El flujo B, en tanto, no sufre contención ya que es el único flujo dirigido al puerto de salida 1; por ello no es necesaria una cola de espera para él.

Las etapas de entrada y salida interconectan los puertos de entrada/salida con la etapa de memoria mediante distintos *caminos de procesamiento*. Cada puerto puede conectarse a uno o más caminos de procesamiento. Por ejemplo, en la Fig. 2.9 el puerto de entrada 3 se conecta a un camino de entrada, mientras que el puerto de salida 3 está asociado a múltiples caminos de salida. En cada camino de procesamiento existen operaciones específicas, llamadas *primitivas* de procesamiento, aplicadas a cada paquete que sigue ese camino. Los caminos de procesamiento pueden compartir algunas primitivas y/o colas de espera pero no todas ellas; en tal caso se trataría en realidad de un único camino de procesamiento. Para compartir ciertas primitivas y/o colas, los caminos pueden ser combinados en ciertas secciones y separados en otras. Durante el multiplexado necesario para compartir primitivas y/o colas, se debe siempre evitar la colisión de datos. En la etapa de entrada, esto se logra imponiendo que en los puntos de agregación los caminos involucrados tengan el mismo retardo de datos. Como consecuencia de esto, en la etapa de entrada no es posible agregar datos provenientes de diferentes puertos ya que podrían presentarse juntos en el punto de agregación con la consecuente colisión. En la etapa de salida esto es posible almacenando momentáneamente los datos en colas de espera y agregandolos sucesivamente mediante mecanismos de arbitraje (scheduling).

Un *flujo de paquetes* se asocia fundamentalmente a un conjunto de paquetes que comparten hosts de origen y destino. En la actualidad, esto no necesariamente implica que compartan puertos, direcciones IP, o incluso máquinas físicas de origen y destino, ya que las técnicas de virtualización de recursos pueden alterar estos parámetros durante la vida de un determinado flujo. En el esquema de la Fig. 2.9, cada flujo se asocia fundamentalmente a un camino de ingreso y a un camino de egreso. Por ejemplo, los flujos C y D están relacionados con dos caminos de ingreso diferentes, pero comparten el mismo camino de egreso. Dos flujos pueden inclusive compartir los mismos caminos de ingreso/egreso y la misma cola de espera en la etapa de memoria, pero aún así ser procesados de forma diferente.

El concepto de flujo es muy conveniente ya que permite abstraerse de la topología de red física y concentrarse en el tratamiento que recibe el tráfico según las necesidades; sin embargo el desempeño de la plataforma física de base está aún dado por la cantidad de *paquetes* que es capaz de procesar en una unidad de tiempo, junto con la cantidad de puertos que es capaz de soportar. El concepto de paquete permite abstraer la red del contenido transportado por ella; por este motivo, la unidad utilizada en las arquitecturas de procesamiento de flujos es el paquete. Un paquete involucra fundamentalmente una sección de *datos*, que es la información transportada (una imagen, un texto, etc.); y un *encabezado* donde se incluye información sobre su trayectoria en la red. Para nuestros fines de procesamiento de flujos, definimos en este encabezado una sección específica de *control* que indica el estado del paquete en lo que a su flujo asociado concierne. Esta estructura general se ilustra en

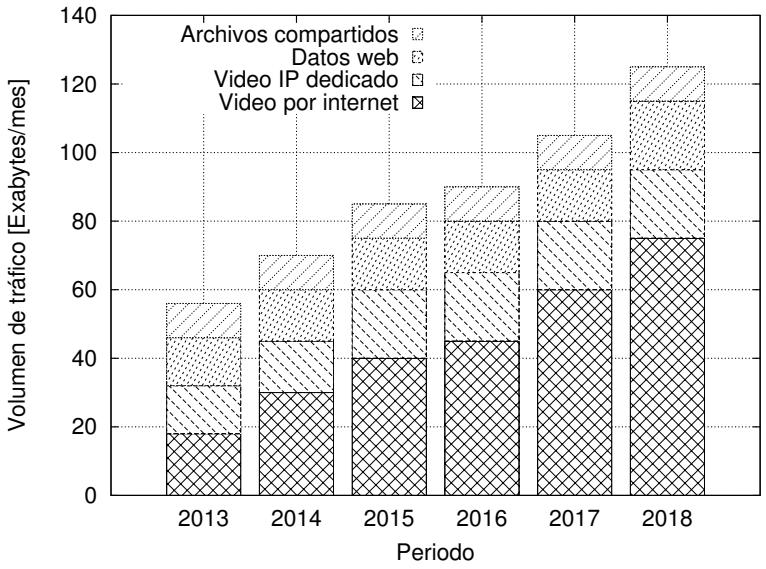


Figura 2.11: Evolución del tráfico IP y su distribución según servicio (Fuente: Cisco Visual Networking Index 2014)

la Fig. 2.10. En dicho esquema, los delimitadores de inicio de paquete *start of packet, sop)*, fin de paquete *(end of packet, eop)*, y módulo *(module, mod)* sirven al procesamiento digital específico de un paquete. Dentro de una primitiva de procesamiento, los datos son en general tratados en bloques o palabras de ancho mucho menor al tamaño de un paquete. Por otro lado, los paquetes tienen un tamaño desconocido al momento de su arribo a cada etapa de procesamiento. Los indicadores *sop* y *eop*, asociados a cada palabra procesada, indican cuándo esta palabra es la primera o la última del paquete. En el caso de que sea la última palabra del paquete, no necesariamente todos los bits de ella son válidos, por lo que el campo *mod* indica los que no lo son a fin de descartarlos. Finalmente, el identificador de flujo *(flow identifier, fid)* indica el flujo al que pertenece el paquete como consecuencia del camino de procesamiento asociado en el contexto del esquema planteado.

2.4.2. Primitivas de procesamiento de paquetes

En esta sección, analizaremos las funciones presentes en una red de *conmutación por paquetes*. En este tipo de redes la información es segmentada y encapsulada en unidades llamadas paquetes, cada una de las cuales deben ser procesadas para determinar su trayectoria en la red. Esta granularidad de procesamiento es lo que las diferencia fundamentalmente de las redes *conmutadas por circuitos*, donde se establece un canal origen-destino por donde se cursa toda la información intercambiada entre ellos. Cada uno de estos métodos presenta sus ventajas e incovenientes; por ello actualmente ambos se suelen utilizar en conjunto como veremos más adelante.

Los mecanismos de procesamiento básico de paquetes surgen originalmente ante la necesidad de interconectar un conjunto de computadoras, o *hosts*, a una red compartida para el intercambio de información. El standard predominante en tales redes es Ethernet, donde la identificación de hosts

de origen/destino se realiza en base a direcciones de acceso (direcciones MAC) de 6 bytes. Con el surgimiento de redes que interconectan a su vez tales redes Ethernet entre sí, en sus orígenes la ARPANET y actualmente Internet, se hizo necesario utilizar direcciones que fueran capaces de representar una red completa o conjunto de MACs; surge así el uso del direccionamiento por *prefijos* en base a direcciones de Internet (direcciones IP), de 32 bits en el caso de IPv4. Como veremos más adelante, dichas direcciones involucran un *número de red* así como una *máscara*; ambos posibilitan agrupar varios hosts bajo una dirección común y escalar el intercambio de tráfico más allá de una red local.

Con estos dos esquemas de redes, surgen los dos casos elementales de procesamiento en redes conmutadas por paquetes: comprobación de direcciones MAC *exactas* y comprobación de *prefijos* IP. El primero consiste en comparar el valor exacto de la dirección MAC del paquete contra una lista de valores exactos de MACs, y normalmente se ejecuta en equipos llamados *switches de capa de enlace* que interconectan una red Ethernet segmentada. El segundo es algo más complejo ya que involucra comparar tanto direcciones IP como sus longitudes, y es llamado *ruteo*. Ambos son utilizados durante el encaminamiento de paquetes desde su host origen hasta su destino, en los equipos de *conmutación*. Estos equipos cuentan con múltiples puertos de entrada/salida por donde se interconectan los hosts y redes; realizan búsqueda o *lookup* a muy alta velocidad y de acuerdo a alguna tabla de direcciones, conmutando el paquete al puerto desde donde pueden alcanzar su destino, y almacenandolo transitoriamente en colas de espera en caso de ser necesario.

Los servicios transportados por Internet han evolucionado notablemente desde estos dos esquemas básicos, dando lugar a una gran cantidad de protocolos y tipos de tráfico; esta evolución debe ser acompañada por las correspondientes arquitecturas y tecnologías para su soporte agregando permanentemente nuevas funciones de procesamiento. Dos características fundamentales marcan la evolución de tales funciones: por un lado, el procesamiento es cada vez más complejo debido a la *heterogeneidad del tráfico* soportado; ya no sólo se encargan de conmutar paquetes, sino que deben modificar dinámicamente el plano de datos definido para cada flujo. Por otro lado, las funciones se deben ejecutar más rápidamente ya que el *volumen de tráfico* es creciente y así también lo es la velocidad requerida a las redes. Los caminos de datos se modifican en base a métricas tales como carga de los equipos, latencia, nivel de seguridad, contenido del paquete, etc.. Las velocidades, por otro lado, han evolucionado desde 1Mbit/s hasta los actuales 100Gbit/s en el caso de redes Ethernet, mientras que el tráfico global transportado ha pasado de aproximadamente 100 gigabytes por día en 1992 a 30000 gigabytes por segundo (GBps) en 2013 [2]. Esto se observa en la Fig. 2.11, donde se muestran además los aportes según el tipo de tráfico.

Como una primera aproximación, las funciones utilizadas en redes de datos se pueden dividir de la siguiente forma:

- *Funciones de capa física*, las cuales convierten señales transportadas por el medio físico de enlace en una secuencia de bits (bitstream) con cierto formato de trama (frame).
- *Funciones de camino rápido (Fast Data Path o Plane)*, que deben operar a *velocidad de línea* sobre cada uno de los paquetes.

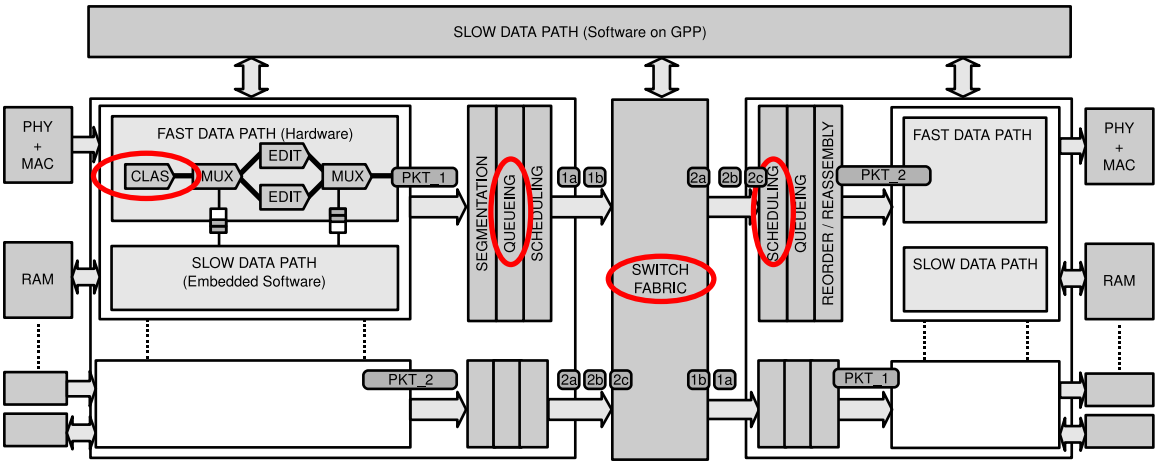
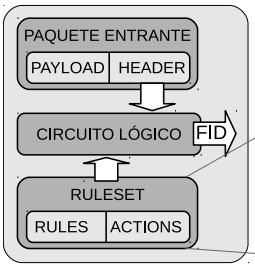


Figura 2.12: Equipo genérico de conmutación con sus funciones asociadas

- *Funciones de camino lento (Slow Data Path o Plane)*, que operan sobre ciertos paquetes e involucran comúnmente operaciones más complejas o impredecibles que aquéllas del camino rápido.
- *Funciones de conmutación*, que tienen como objeto planear y ejecutar la transferencia de paquetes entre puertos de la forma más rápida y efectiva posible.
- *Funciones del camino de control (Control Path o Plane)*, que se ocupan de tareas de configuración y gestión de la red y sus equipos asociados; son en general de alta complejidad computacional.

A fin de evaluar las implementaciones más convenientes de estas funciones, a continuación estudiaremos las más difundidas y pertinentes a esta Tesis junto con los compromisos que implica su implementación en las tecnologías analizadas. Este análisis forma parte del primer aporte de la presente Tesis [52]. En la Fig. 2.12 se observa la arquitectura de un equipo de conmutación genérico, junto con la ubicación típica de las funciones que analizaremos. En este esquema se distinguen claramente el plano de datos, con sus etapas de entrada/salida de paquetes; la matriz de conmutación entre los puertos de entrada/salida; así como el plano de control que se encuentra claramente separado por algún tipo de bus propio del equipo o compartido entre varios equipos (como es el caso de OpenFlow). Obsérvese que este esquema puede involucrar implementaciones tanto en software como en hardware según lo más conveniente en cada caso.

Volviendo al ejemplo de la Fig. 2.9, se observa que los paquetes entrantes necesitan fundamentalmente mapearse a flujos; para ello se utiliza la primitiva de *clasificación* (C) de paquetes en flujos. Esta función es responsable de asignar un valor *fid* al campo de control correspondiente a cada paquete como se muestra en la Fig. 2.10. Una vez que los paquetes son clasificados en flujos, éstos deben ser enrutados en consecuencia a través de sus caminos de procesamiento asociados mediante primitivas *multiplexers* (M). Los paquetes pueden además requerir modificación de su encabezado de protocolo, a fin de propagar información de control entre primitivas; a esto lo realiza un *editor* (E). Ejemplos de esta edición son el decremento del tiempo de vida de un paquete *(Time To Live,*



No.	IP(64bits)		Port (32bits)		Protocol(8bits)
	Source (32bits)	Destination (32bits)	Source (16bits)	Destination (16bits)	
1	0.0.0.0/0	0.0.0.0/0	0 : 65535	21 : 21	0/ff
2	0.83.1.0/24	0.0.4.6/32	0 : 65535	20 : 30	17/ff
3	0.83.4.0/22	0.0.0.0/0	0 : 65535	21 : 21	0/0
4	0.0.9.0/24	0.0.0.0/0	0 : 65535	0 : 65535	0/ff
5	0.83.0.77/32	0.0.4.6/32	0 : 65535	0 : 65535	17/ff

Figura 2.13: Clasificador: (a) esquema general, (b) tabla de reglas típica

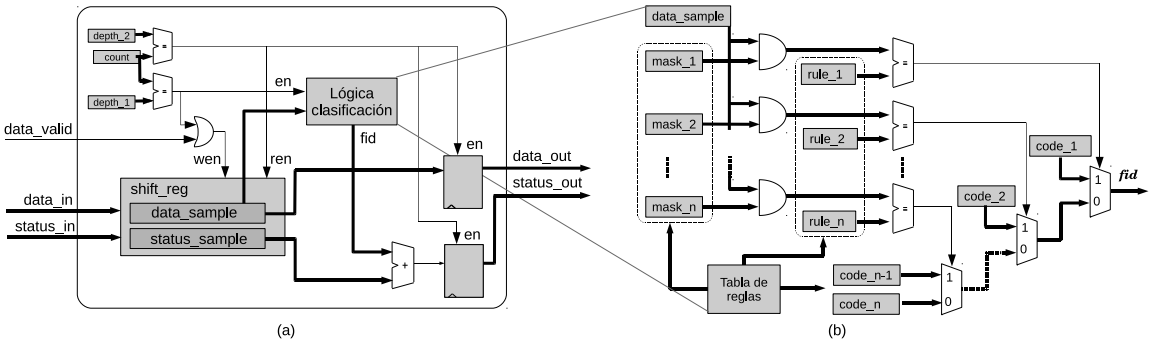


Figura 2.14: Implementación básica: (a) esquema general, (b) proceso de clasificación

TTL), la actualización de etiquetas (tags) de LAN virtual *(Virtual LAN, VLAN)*, actualización de las etiquetas en conmutación multi-protocolo (MPLS switching),etc.. Por otro lado, según la Fig. 2.9 se requiere una etapa de colas de espera para resolver problemas de contención entre las etapas de salida y de entrada. Un requerimiento especial de estas colas es que ellas deben almacenar *paquetes completos*; vale decir que si la cola se llena durante el almacenamiento de un paquete éste debe ser descartado completamente. Con este objeto, las colas deben incorporar lógica que las haga sensibles a los indicadores *sop, eop* y *mod*; esta lógica forma la primitiva de *almacenamiento o queueing* (Q). Finalmente, los paquetes exitosamente almacenados en las colas de espera deben ser atendidos evitando colisiones, en base a un cierto criterio o política, esto se realiza mediante la primitiva de *planificación o scheduling* (S). Esta primitiva debe incorporar suficiente flexibilidad para adoptar diferentes políticas de planificación basadas en asignación de prioridades a las colas de espera.

A continuación se analizan las primitivas planteadas; primero desde un punto de vista general analizando opciones tecnológicas, luego mediante ejemplos concretos de implementación propia en FPGAs. El primer análisis fue publicado en [52], mientras que la implementación de primitivas es la segunda contribución de esta Tesis [53]. Si bien en trabajos relacionados como [17] y [31] se informan resultados de implementación específicos, no se encontraron módulos de código abierto que cumplieran con la flexibilidad necesaria para nuestras objetivos; por ello esta primera implementación conceptual fue fundamental para evaluar y validar nuestras ideas.

2.4.3. Clasificación (Classification)

Esta primitiva será objeto de análisis exhaustivo a lo largo de esta Tesis, por lo que nos limitaremos aquí a introducirla en el contexto del camino de procesamiento. Los clasificadores son bloques esenciales en cualquier arquitectura de procesamiento de redes, y como veremos son unas de las etapas más críticas en lo que respecta a implementación. En un clasificador, el encabezado de un paquete se analiza para decidir sobre su camino de procesamiento. Esta decisión impacta en una palabra de control asociada a cada palabra del paquete, que las demás primitivas interpretarán a lo largo del camino para tomar sus propias decisiones. De esta forma, el clasificador determina el procesamiento que sufrirá el paquete impartiendo *órdenes diferidas* a las demás primitivas en al camino de procesamiento. Algunas aplicaciones típicas de este mecanismo son Calidad de Servicio *(Quality of Service, QoS)*, Clase de Servicio *(Class of Service, CoS)*, facturación de tráfico (traffic billing), filtrado de paquetes (firewalling), detección de intrusiones *(Network Intrusion Detection Systems, NIDS)*, entre muchas otras. En la Fig. 2.13(a) se muestran los componentes generales de un clasificador, mientras que la Fig. 2.13(b) muestra una pequeña tabla de reglas típica de un clasificador basado en quíntuplas IP formadas por los campos Dirección IP de origen(SrcIP) / Dirección IP de destino (DstIP) / Puerto de origen (SrcPort) / Puerto de destino (DstPort) / Protocolo (Protocol o Prot).

El clasificador normalmente considera múltiples campos en el encabezado del paquete y los compara contra múltiples reglas de clasificación, por lo que su complejidad aumenta con el espacio de búsqueda definido por el número de campos y el número de reglas. Además, se hace necesario un mecanismo para, a partir de la comparación contra cada una de las reglas, asociar un identificador de flujo *fid* al paquete. En la Fig. 2.14(a) se observa la implementación conceptual realizada. Un contador *count* lleva registro de la cantidad de palabras leídas a medida que éstas se cargan en un registro de desplazamiento *data_sample* junto con sus palabras de control en *status_sample*. A medida que este contador pasa por los campos de interés a profundidades *depth_1, depth_2*, etc., se activa la lógica de clasificación asociada. Las primeras palabras del paquete contienen su encabezado, por lo que se puede decidir sobre él a medida que se almacena en los buffers internos. En la Fig. 2.14(b) se detallan las operaciones lógicas básicas de clasificación. El encabezado capturado en *data_sample* hasta la profundidad de interés se filtra mediante máscaras configurables *mask_1,...,mask_n* para extraer la información considerada para clasificar. Este filtrado, en su forma más simple, consiste en operaciones lógicas AND bit a bit. Los datos filtrados se comparan entonces contra las reglas *rule_1,...,rule_n* utilizando en este caso conceptual un mecanismo de búsqueda lineal. La comparación contra cada regla se realiza mediante operaciones XOR. En este ejemplo, cada regla se asocia a un determinado valor de *fid* (*code_1,...,code_n*), donde dos reglas pueden tener asociado el mismo *fid*. Las reglas no son en general mutuamente exclusivas, por lo que el paquete puede satisfacer múltiples reglas; se debe establecer entonces de antemano una cierta prioridad entre ellas, la que determinará el *fid* final como se ilustra en la Fig. 2.14(b).

Tanto las profundidades de captura como las reglas y las máscaras (los *parámetros* del clasificador) se pueden almacenar bajo la forma de (a) funciones lógicas combinacionales, (b) en los registros síncronos o (c) en los bloques de memoria SRAM disponibles en el FPGA. En el primer caso,

Cuadro 2.2: Clasificador: opciones de implementación

GPPs	FPGAs	ASICs
Muchas reglas en SDRAM lenta	Registros dedicados; SRAM interna y limitada de baja latencia; SDRAM externa de latencia media	memoria ad-hoc interna limitada; SDRAM externa (si se cuenta con interfaz)
Algoritmos complejos a baja velocidad	Arquitecturas paralelas a alta velocidad	Arquitecturas paralelas a alta velocidad
Reglas programables	Reglas programables/reconfigurables	Reglas parcialmente programables

el compilador de hardware toma las especificaciones de entrada (desplazamientos, reglas y máscaras) y genera las salidas deseadas (fid) para cada posible encabezado; se llega así a una expresión en forma de suma de productos (Sum of Products, SOP) minimizada que se almacena en las tablas de lookup (Look-Up Tables, LUTs) del FPGA, con mínimo consumo de hardware y alto desempeño. Esta primera opción, sin embargo, se basa puramente en lógica combinacioal por lo que no permite modificar los mencionados parámetros en tiempo de ejecución. Para ello se pueden utilizar los registros síncronos del FPGA en un *banco de registros (regfile)*, o bien los bloques de memoria SRAM disponibles. El uso de registros permite acceso totalmente concurrente, por ejemplo a todas las reglas y máscaras de la Fig. 2.14(b) con óptima velocidad de operación, sin embargo no escala mas allá de pocas reglas debido a la complejidad de ruteo necesaria para interconectar registros individuales. La memoria SRAM, finalmente, permite almacenar grandes rulesets pero plantea compromisos de desempeño ya que su ancho de palabra es limitado. Nuestra primera implementación básica presenta escalabilidad limitada por la necesidad de acceso concurrente a todas las reglas/máscaras y por la disposición de codificadores de prioridad en cascada con el consiguiente retardo de propagación.

2.4.4. Conmutación (Switching)

El sistema de conmutación juega, al igual que el clasificador, un rol determinante en la implementación de las redes de datos. Mientras que el clasificador determina los caminos de procesamiento, el conmutador debe trazar eficientemente parte de esos caminos. En particular, el conmutador absorbe los problemas de contención debidos a la utilización de recursos compartidos. En los enrutadores IP, por ejemplo, el sistema de conmutación debe tener inteligencia suficiente para enrutar paquetes hacia los puertos de salida con máxima utilización de los recursos disponibles. Algunas de las características importantes en esta primitiva son el soporte de multicast/broadcast hacia varios puertos a la vez, acelaración (speedup) utilizada, posibilidad de bloqueo interno y trabajo con celdas fijas o paquetes variables.

Esta primitiva puede trabajar esencialmente por división de tiempo *(Time Division Switching, TDS)* o de espacio *(Space Division Switching, SDS)*. Los conmutadores por división en el tiempo utilizan un bus o una memoria compartidos entre los puertos de entrada/salida (E/S). En el caso de un bus compartido *(Shared Bus, SB)*, tal como PCIe en arquitecturas GPP, éste debe garantizar

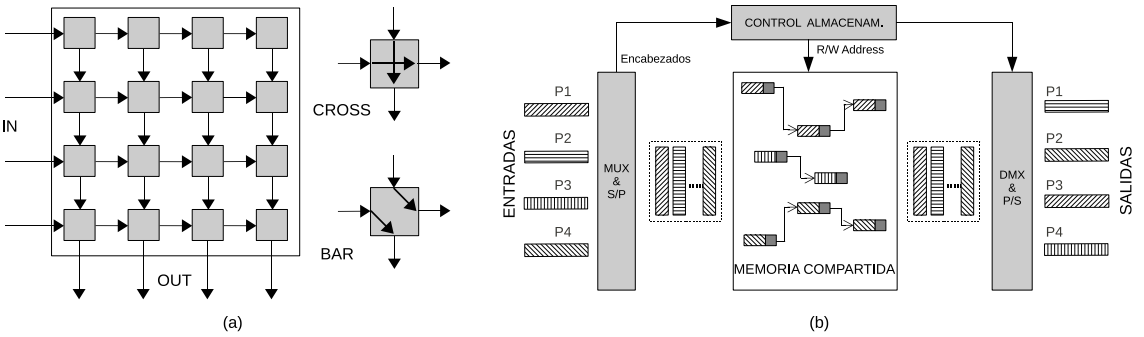


Figura 2.15: Conmutación: (a) SDS (Crossbar), (b) TDS (Memoria compartida)

ancho de banda suficiente para la transferencia agregada de paquetes entre múltiples puertos. En el caso de ser una memoria compartida *(Shared Memory, SM)*, ésta se asigna dinámicamente a múltiples colas de espera, cada una de ellas asociada a puertos de entrada y salida; por lo que debe soportar N escrituras/ 1 lectura por slot de tiempo en el peor caso (N = puertos de E/S). El diseño de esquemas de asignación que permitan adecuada explotación de la capacidad de memoria en este tipo de conmutación es un área de intensa investigación, podemos citar [54] como ejemplo. La conmutación por bus compartido se utiliza típicamente en arqutecturas basadas en GPPs, por lo tanto se puede ilustrar por las Figs. 2.1 y 2.6. La conmutación por memoria compartida se abordará en detalle en el contexto de los esquemas de almacenamiento.

Los conmutadores por división de espacio, también llamados matrices de conmutación o *switch fabrics*, ofrecen en general mayor desempeño y son los preferidos en aplicaciones de mayores exigencias. La principal ventaja de estos conmutadores es que permiten múltiples transferencias *simultáneas* entre puertos de E/S; mientras que su limitación está dada por efectos físicos en las interconexiones de *backplane* y el efecto de *bloqueo interno* propio de sus arquitecturas. Algunos ejemplos son los switches Crossbar, los totalmente interconectados, Clos, Banyan, o Multiplano [55]. Estos esquemas son implementados tradicionalmente mediante ASICs, mientras que los FPGAs actuales son también capaces de integrarlos a velocidades de 10/100 Gbps [56]. En el Cuadro 2.3 se observan los compromisos de implementación de un conmutador en las tecnologías actuales. En la Fig. 2.15, en tanto, se muestran una arquitectura Crossbar y una de memoria compartida como ejemplos típicos de SDS y TDS respectivamente.

Finalmente, cabe mencionar que la comutación se puede realizar en base a celdas de longitud fija, como en el caso de conmutadores de modo de transferencia asíncrona *(Asynchronous Tansfer Mode, ATM)*; o paquetes de longitud variable como es el caso de redes Ethernet o IP. Ya que las redes actuales pueden incorporar ambos casos, las funciones de conmutación deberían tener la flexibilidad suficiente para soportarlos.

Cuadro 2.3: Conmutador: opciones de implementación

GPPs	FPGAs	ASICs
Memoria/bus compartidos	Division por tiempo/espacio	Division por tiempo/espacio
Desempeño bajo/medio	Alto desempeño	Muy alto desempeño
Optimización muy limitada	Optimización limitada por la familia	Optimización en tiempo de diseño

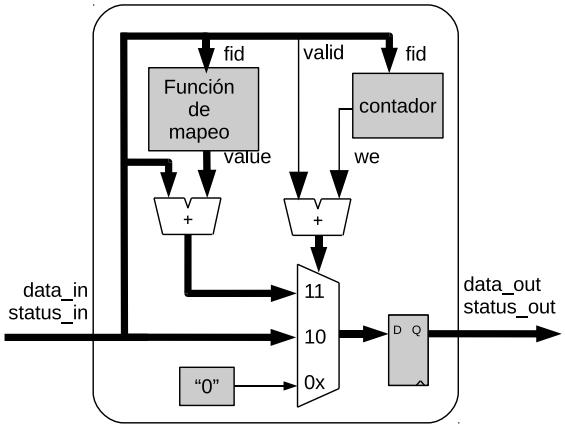


Figura 2.16: Primitiva de edición

2.4.5. Edición (Tagging)

La primitiva de edición, no obstante su simplicidad, es fundamental para la definición de caminos de procesamiento. El clasificador, como vimos, toma decisiones sobre los paquetes y las escribe como etiquetas (fid) en un campo de control fijo reservado para ello. El editor sirve, en cierta forma, para traducir las decisiones del clasificador hacia valores interpretables más allá de los puertos de salida. El editor lee el fid en la sección de control y en base a él cambia valores en el propio encabezado del paquete, los que pueden ser interpretados en otros equipos de la red. En la Fig. 2.16 se observa la implementación general de esta primitiva. A fin de modificar un campo específico del encabezado, se utiliza un contador de palabras. Al llegar un nuevo paquete (detección de *sop*) se extrae el fid del campo de control. Cada fid se asocia con un valor y desplazamiento específicos y configurables de edición. Un multiplexor se encarga de reemplazar los valores entrantes con el valor nuevo sólo al momento de cumplirse el offset, mientras el resto del paquete es re-transmitido sin modificación. La salida de palabras, ya sean éstas modificadas o no, se controla mediante el bit *valid*, utilizado a lo largo de toda la arquitectura y controlado por las banderas *sop* y *eop*. Durante la ausencia de paquetes, se transmiten palabras nulas de relleno a fin de mantener el flujo de datos.

2.4.6. Multiplexado

Esta primitiva se encarga de agregar y desagregar paquetes de diferentes flujos, permitiendo la co-existencia de múltiples caminos de procesamiento dentro de las etapas de etapa/salida de las Figs. 2.9 y 2.12, y es controlada por las decisiones del clasificador mediante el fid. Para cumplir su función, los multiplexores deben mantener una tabla relacionando fids con entradas/salidas. La *desagregación* de caminos se implementa leyendo el valor fid del paquete y destinando las palabras a la salida asociada hasta tanto llegue el próximo *eop*. Cabe observar que por las salidas restantes se transmiten datos nulos de relleno, garantizando que sólo una salida es activa para cada paquete. Esta característica se utiliza en la *agregación*, implementándola mediante simple operación XOR de las entradas.

2.4.7. Encolador (Queuer) y Segmentación/re-ensamblado (Segmentation/re-assembly)

Estas primitivas se asocian comúnmente a otras para posibilitar su correcto funcionamiento. El encolador, en particular, asegura el almacenamiento de paquetes válidos en las colas de espera. Esto es debido a que los paquetes tienen en general longitudes no predecibles, por lo que se podría dar el caso en que la cola de espera se llene durante el almacenamiento del mismo. En este caso, el encolador se encarga de descartar la porción y almacenada, descartando efectivamente todo el paquete asociado y habilitando la escritura del siguiente.

Las funciones de segmentación se aplican, por ejemplo, en las interfaces de entrada de un switch para facilitar las conmutaciones realizándolas a intervalos regulares (duración de una celda), esto se llama funcionamiento en *modo de celda*. La conmutación directa de paquetes presenta gran dificultad ya que a la complejidad del planeamiento se suma la variación en los momentos de dicha conmutación para cada puerto. Como se muestra en la Fig. 2.12, la segmentación consiste en dividir los paquetes entrantes en *celdas* de tamaño fijo que atraviesan el switch a intervalos bien definidos y regulares. Al abandonar los puertos de salida del switch, las celdas son re-ensambladas en el paquete original. Los inconvenientes de esto son el desperdicio de ancho de banda cuando el paquete no es un número entero de celdas, el re-ordenamiento necesario en la salida ya que las celdas pueden llegar fuera de orden, y el overhead necesario para relacionar celdas con sus respectivos paquetes. Esto también requiere almacenamientos temporales de celdas durante su procesamiento, operando en modo *store-and-forward*. Un switch también puede utilizar *modo de paquetes* directo, en cuyo caso puede operar en modo *cut-through* donde las conexiones E/S se establecen durante la duración de un paquete completo.

En nuestra arquitectura se utilizaron primitivas de encolamiento para garantizar al eliminación de paquetes no válidos. Las funciones de segmentación/re-ensamblado, en tanto, no llegaron a utilizarse ya que el caso de ensayo no lo requirió.

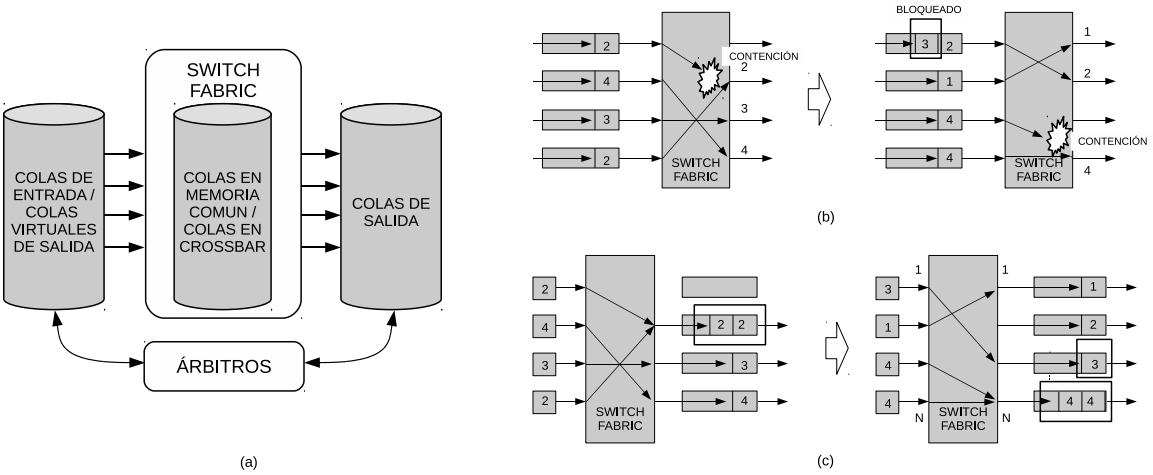


Figura 2.17: Almacenamiento: (a) casos generales IQ/OQ/SQ/XQ/VOQ, (b) HOL blocking en IQ, (c) Solución de HOL en OQ

2.4.8. Almacenamiento (Queueing o Buffering)

Las estrategias de almacenamiento transitorio de paquetes están íntimamente ligadas a la arquitectura de conmutación utilizada; y en conjunto definen desempeño, aplicaciones y requerimientos tecnológicos. Haciendo referencia a la Fig. 2.12, los buffers de almacenamiento pueden ubicarse a la entrada, a la salida, en entrada y salida, o dentro de la matriz de conmutación; estos casos se muestran en el esquema general de la Fig. 2.17(a). El almacenamiento en la entrada sólo requiere escritura/lectura a velocidad de línea, sin embargo sufre el problema conocido como *Head-Of-Line (HOL) blocking*. Éste se produce cuando la palabra al incio del buffer no puede atenderse por efectos de contención en un puerto de salida; bloqueando palabras posteriores que de no mediar este efecto podrían ser atendidas; esto es ilustrado en la Fig. 2.17(b). El esquema de almacenamiento en puertos de salida *(Output Queueing, OQ)* soluciona este problema, sin embargo requiere en el peor caso velocidad de escritura igual a N veces la velocidad de línea, como se ilustra en la Fig. 2.17(c). El esquema de colas virtuales de salida *(Virtual Output Queueing, VOQ)* combina las ventajas de IQ y OQ pero requiere algoritmos complejos de planeamiento, como se verá más adelante. El sistema de colas compartidas *(Shared Memory, SM)*, muy popular en la última década, integra buffering y switching en un solo esquema con gran flexibilidad en cuanto a asignación de recursos a cada puerto; sin embargo dados N puertos de E/S puede llegar a requerir N escrituras y N lecturas para cada operación de conmutación/almacenamiento. También presenta otros problemas como segmentación de memoria para el caso de paquetes y algoritmos complejos de planeamiento. El esquema de almacenamiento en punto de cruce *(Crosspoint Queueing, XQ)* mitiga los problemas de todos los esquemas anteriores, requiriendo velocidad de línea para escritura/lectura y planeamiento simple; sin embargo requiere N^2 colas independientes y no es muy flexible para adaptarse a variaciones de carga entre los diferentes puertos. Tanto VOQ como SM se discuten junto con las primitivas de planeamiento en este trabajo. Otros esquemas, específicamente destinados al caso FPGA y estudiados al iniciar el trabajo de Doctorado, son los basados en redes-en-chip *(Networks-on-Chip, NoC)*; se pueden citar [57] y [58] como ejemplos de estos esquemas.

Cuadro 2.4: Almacenamiento: opciones de implementación

GPPs	FPGAs	ASICs
SDRAM	SRAM interna o SDRAM externa	SRAM interna o SDRAM externa
Alta capacidad y latencia	Baja latencia (interna)/Alta capacidad (externa)	Baja latencia (interna)/Alta capacidad (externa)
Centralizada	Centralizada/distribuida	Centralizada

Cuadro 2.5: Planificación: opciones de implementación

GPPs	FPGAs	ASICs
Programable	Configurable/ Programable	Fijo
Lento	Medio	Rápido

El Cuadro 2.4 muestra las opciones de almacenamiento en las tecnologías consideradas. En particular, se observa que las arquitecturas basadas en GPPs sólo permiten implementar almacenamiento compartido en la memoria RAM del sistema; esto es un cuello de botella importante ya que como vimos los esquemas SM pueden requerir N escrituras/lecturas en cada ciclo, lo que limita mucho la velocidad máxima alcanzada.

2.4.9. Planificación (Scheduling)

La primitiva de planificación determina la atención de paquetes en los puertos de E/S del switch. Cada puerto de salida cuenta con un planificador para evitar colisiones. De este modo, los caminos de procesamiento asociados a un mismo puerto de salida compartirán un mismo planificador. En algunos casos, como los esquemas de buffering virtual de salida (VOQ), se incluyen también planificadores en los puertos de entrada; éstos coordinan acciones con los de salida a fin de lograr las transferencias más eficientes a través de la matriz de conmutación. Los planificadores controlan la configuración de la matriz de conmutación para que ésta ejecute efectivamente la transferencia de paquetes acordada.

En su forma más general, un planificador consta de un multiplexor controlado por un mecanismo de arbitraje. El multiplexor puede estar distribuido en los elementos de conmutación de la matriz, o centralizado y asociado a su árbitro. El mecanismo de arbitraje utilizado, por otro lado, recibe requerimietos de atención *(Service Requests, SRs)* desde múltiples demandantes *(REQuesters, REQs)* y determina su orden de atención, el cual depende de las prioridades relativas asignadas a cada demandante. El esquema de planificación básico asigna prioridades fijas a cada demandante, otorgando servicio al demandante de mayor prioridad. Sin embargo, cuando se trabaja con flujos muy activos, los demandantes de alta prioridad pueden absorber todos los servicios llevando a los demás a estado de inanición (starvation). Otro esquema, que atiende equitativamente a todos los demandantes involucrados, es el basado en el algoritmo *round-robin*. En este algoritmo, las prioridades se asignan en

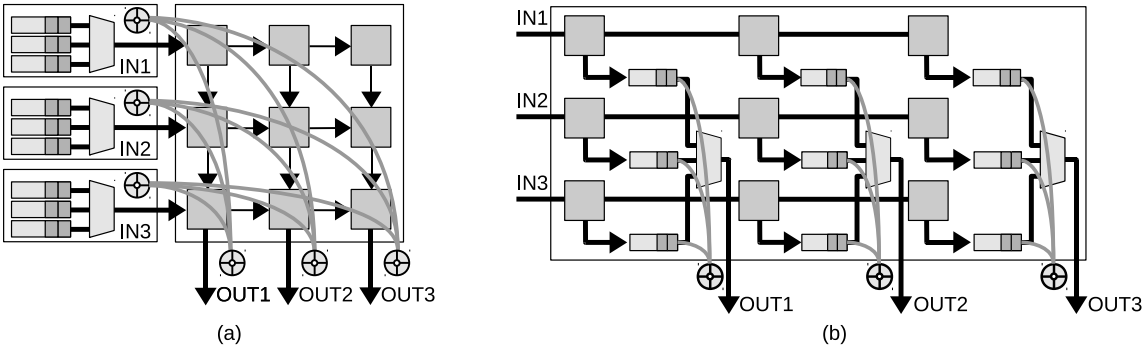


Figura 2.18: Scheduling: (a) aplicación en esquema VOQ (b) aplicación en esquema XQ

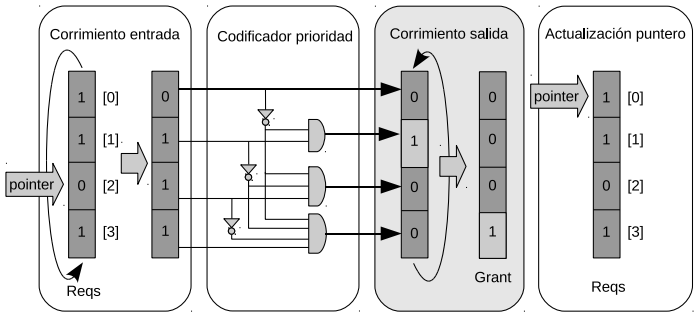


Figura 2.19: Scheduling: esquema implementado en FPGA

turnos, asignando al demandante atendido la mínima prioridad para la próxima ronda de arbitraje. Otros algoritmos son asismismo aplicables cuando se busca establecer otras prioridades, por ejemplo para satisfacer requerimientos de calidad de servicio (QoS). Finalmente, cabe mencionar que el planificador puede tomar decisiones a nivel de paquetes (por ejemplo IP) o a intervalos regulares (por ejemplo celdas ATM). Adicionalmente, el scheduler se puede combinar con funciones de supervisión (policing) y conformación (shaping) para satisfacer requerimientos más complejos. Los supervisores *descartan* paquetes en los puertos de entrada del switch, mientras que los conformadores *retardan* paquetes en los puertos de salida antes de propagarlos más adelante.

El Cuadro 2.5 muestra las opciones tecnológicas para esta función. Las arquitecturas basadas en GPPs requieren planificación centralizada e implementada mediante software. La eficiencia de las demás tecnologías, en tanto, depende fuertemente del esquema de almacenamiento adoptado. Por ejemplo, el esquema de colas virtuales de salida (VOQ) es uno de los más eficientes, pero para ello requiere algoritmos de planificación complejos a alta velocidad. Esto se ilustra en la Fig. 2.18(a) para el caso de tres puertos de E/S, donde se observan tres planificadores de entrada y tres de salida que deben tomar decisiones coordinadas. Los esquemas de XQ también tienen buena eficiencia con algoritmos de planificación simple y distribuida; esto tiene como contrapartida mayor consumo de recursos hardware. En la Fig. 2.18 se ilustra la situación de los planificadores en tal esquema; como se observa sólo es necesario uno de ellos para cada puerto de salida lo que reduce notablemente la complejidad del sistema.

Durante las implementaciones de planificación round-robin realizadas en FPGAs, se observó que

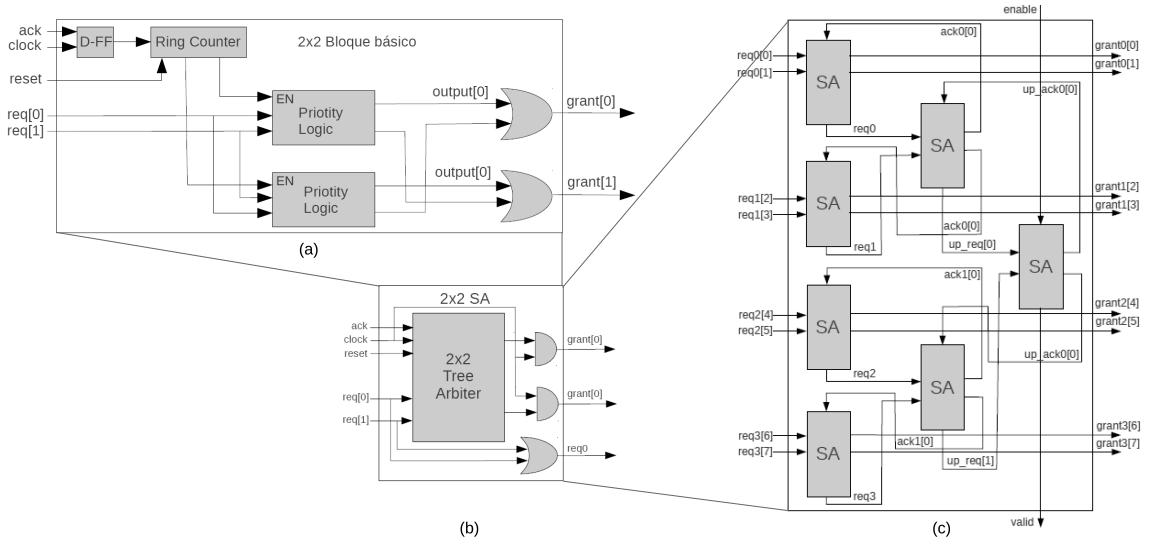


Figura 2.20: Scheduling: esquema round-robin en árbol de desempeño mejorado

existen múltiples estilos de codificación que llevan a resultados de muy diversos rendimientos [59]. En la Fig. 2.19 se ilustra el esquema implementado para un balance complejidad vs. desempeño. Un vector *Reqs* almacena las solicitudes para cada demandante, mientras que un puntero mantiene la posición del demandante actualmente atendido. En base a este puntero, se rota el vector en una primera etapa para implementar la rotación de prioridades. Posteriormente, este vector rotado ingresa a un codificador de prioridad que selecciona el primer requerimiento activo; luego se rota nuevamente el vector para atender al solicitante adecuado. La decisión del planificador genera un vector *Grant* que indica el demandante asignado para atención. Otros diseños más complejos y eficientes se encuentran en [60], cabe mencionar entre ellos la implementación en árbol, ilustrada en la Fig. 2.20 que fue asimismo evaluada durante la presente Tesis con mejores resultados en cuanto a escalabilidad.

2.5. Procesamiento en redes mediante FPGA: caso de estudio

El hardware reconfigurable, en particular las FPGAs, permiten gran flexibilidad en el procesamiento en redes. Por un lado, la posibilidad de separar primitivas individuales permite implementar sólo el procesamiento *necesario* para cada aplicación, sin tener que utilizar equipamiento comercial de mayor complejidad y costo como routers, switches, *middleboxes* (firewalls, NATs, etc.) que incluyen otras funciones que quedan ociosas. Aún más, los recursos que quedan disponibles en la FPGA luego de nuestra implementación pueden ser aprovechados para otro tipo de aplicaciones, sin quedar necesariamente ligados a la misma arquitectura de procesamiento. Por otra parte, al definir las funciones de procesamiento en forma independiente y vincularlas dinámicamente, los dispositivos de red dejan de ser equipos físicamente centralizados para convertirse en una *combinación*

particular de funciones de red. De esta manera, dichos dispositivos virtuales pueden adaptarse más efectivamente a las necesidades de procesamiento, implementando las primitivas individuales en el exacto *lugar* donde son necesarias y más efectivas. Como se introdujo al hablar de FPX [38], también la configuración de las funciones individuales se puede realizar en forma remota. Estos conceptos están íntimamente ligados al de *virtualización de redes*. Asimismo, el uso de FPGAs permite a la vez *re-configuración* del hardware, *re-programación* de cómo dicho hardware se comporta; así como *combinación* de este hardware con plataformas software, tanto basadas en GPPs como en procesadores embebidos, lo que le aporta la flexibilidad propia de los sistemas mixtos (Sec. 2.3).

En base a las funciones de procesamiento introducidas en la Sec. 2.4.2, y a las características tecnológicas analizadas en la Sec. 2.2, en esta sección se aborda un caso de estudio real de complejidad moderada, que permite visualizar las posibles aplicaciones. Se considera el caso de un switch de capa 3 (Layer 3 o L3 switch), el cual además de manejar paquetes basándose en su dirección exacta de control de acceso al medio *(Media Access Control, MAC)*, permite agrupar puertos en *Redes LAN Virtuales (VLANs)* y enrutar paquetes entre ellas basándose en sus direcciones IP. Se definió este escenario ya que representa una tendencia más general hacia agregación de paquetes en flujos. El etiquetado VLAN (VLAN tagging), definido en la norma IEEE 802.1Q, permite compartir una red Ethernet física entre múltiples flujos lógicos independientes que representan redes Ethernet virtuales. Estas redes virtuales, a pesar de compartir un switch, no se ven entre sí en la capa 2; una de las ventajas directas de esto reside en la segmentación de los dominios de difusión (broadcasting)[2] mejorando el aprovechamiento de recursos. Un switch de capa 2 es capaz de manejar VLANs cuando, además de realizar la conmutación característica por direcciones MAC exactas, puede agrupar sus puertos según etiquetas VLAN *(VLAN tags o VIDs)*. Así, el forwarding interno a cada VLAN es implementado por simple comprobación del correspondiente campo *VID* previsto en el encabezado para soporte de VLANs. El enrutamiento *entre* VLANs, en tanto, requiere procesamiento externo al switch, a nivel de capa de red (L3), y consecuente edición del VID. Cuando un paquete entrante a un puerto del switch no está destinado a la VLAN a la que pertenece dicho puerto, el switch no puede determinar su destino. En consecuencia, el paquete se envía a un puerto troncal conectado a un enrutador de un puerto *(Router on a Stick, RoS)* que se encarga de procesar el tráfico inter-VLAN. Esta configuración se ilustra en la Fig. 2.21(a). El procesamiento inter-VLAN se basa en la inspección de la dirección L3 (IP) para determinar la nueva VLAN de destino, y la consecuente edición de la etiqueta VID. Si bien en este caso se trata de un solo switch, cabe aclarar que la VLAN de destino podría también encontrarse en otro switch. En el simple esquema planteado, por ejemplo, el tráfico entre 4 VLANs es agregado en un enlace troncal para ser procesado; en particular la VLAN 4 constituye una puerta de enlace (Gateway) hacia la internet. Los encabezados involucrados en las redes de datos consideradas se muestran en la Fig. 2.22, donde se destacan aquéllos de mayor relevancia para el presente ensayo. El campo *VLAN Tag* en particular, que posibilita el uso de redes VLAN, no es de uso completamente general, requiriendo de equipamiento adecuado para su correcto procesamiento.

En el esquema implementado, se utilizó un L2 Switch de la empresa 3COM con 24 puertos GbE,

[2] Un dominio de difusión es un área lógica en una red de datos, donde cualquier terminal puede comunicarse con otro sin precisar para ello de un dispositivo de encaminamiento o ruteo. El tamaño de dichos dominios está limitado por la cantidad de terminales interconectados, requiriendo enrutadores para segmentarlos a partir de cierto tamaño.

mientras que el L3 RoS se implementó mediante la plataforma de desarrollo *Altera PCI Express Development Kit*, basada en un FPGA Altera Stratix GX II. Para generación de tráfico se utilizó el software *Packeth*, mientras que para captura se recurrió al software *Wireshark*. En la Fig. 2.21(b) se muestran los caminos de procesamiento definidos dentro del RoS. Como se observa, en este caso se incluyen la mayor parte de las primitivas analizadas. El tráfico proveniente del puerto troncal ingresa al RoS pasando por una primitiva de *Clasificación*; en ella se comparan campos configurables del paquete contra un conjunto de reglas también configurable para decidir en consecuencia su identificador de flujo *fid*. Este identificador, local a la arquitectura, se propaga a través del campo de control actuando sobre una primitiva de *multiplexación*. Este módulo divide el tráfico entre flujos conocidos o *clasificados*, que permanecerán dentro de la arquitectura, y aquéllos desconocidos que son direccionados a una plataforma sofware para decidir sobre ellos en base a otras funciones. Los paquetes clasificados se dirigen a un *editor*, que basado en el valor *fid* propagado desde el clasificador escribe un valor de VID identificable por el L2 switch externo a la arquitectura.

Por otro lado, los paquetes procedentes del software son considerados como clasificados, por lo que ingresan directamente a una segunda primitiva de edición como se muestra en la Fig. 2.21(b). Tanto los paquetes destinados a software, como los provenientes del software y de hardware se encolan en etapas de *almacenamiento temporal* (FIFO_ext y FIFO_in respectivamente) mediante sus correspondientes módulos auxiliares de encolamiento *Q*. La primitiva *FIFO_int*, en particular, fue diseñada para alojar múltiples FIFOs en memoria compartida (SM), pudiendo soportar de este modo múltiples puertos por sí misma (suponiendo velocidades de escritura/lectura apropiadas). Finalmente, los paquetes encolados son atendidos mediante módulos *Planificadores (Schedulers)*. Un planificador implementado sobre FPGA atiende paquetes clasificados tanto en software como en hardware y los despacha hacia el L2 Switch a través del puerto troncal. Por otro lado, los paquetes no clasificados en hardware son atendidos por un scheduler software, no incluido aquí.

La Fig. 2.23 muestra en mayor detalle la interconexión interna del RoS junto con sus módulos auxiliares. Los archivos de prueba (*test data file/wave file*) permiten capturar todas las señales de datos y control a nivel de capa 1 para su posterior análisis. Los adaptadores MAC sirven para adaptar señales de la interfaz 10GbE prevista en nuestra arquitectura a la interfaz 1GbE conque cuenta el L2 switch disponible; estos módulos asimismo agrupan todas las señales de control en un único bus *status* que prevee otros campos de control internos a la arquitectura. En el presente caso de ensayo se puede observar que todo el tráfico se cursa por un único puerto troncal, esto en parte fue definido por el hardware disponible. Sin embargo, se desea también evaluar el comportamiento de las primitivas diseñadas en un contexto de múltiples puertos de E/S, donde se producen problemas de contención. Para simular esta situación se utilizaron frames especiales de pausa según la norma IEEE 802.3x, y control de flujos con prioridad IEEE 802.1qbb. El módulo *pause* detecta los frames de control de pausa a través del editor, e inhibe en consecuencia los servicios correspondientes en el módulo de planificación; de este modo se regula la carga de las colas de espera para evaluar su correcto funcionamiento ante situaciones de contención. Finalmente, cabe mencionar que se cuenta con interfaz hacia una plataforma software a fin de implementar arquitecturas mixtas; esta se utilizó momentáneamente en modo *loopback* como ilustra la Fig. 2.23.

En este contexto, merece especial mención el esquema de memoria implementado; éste permite

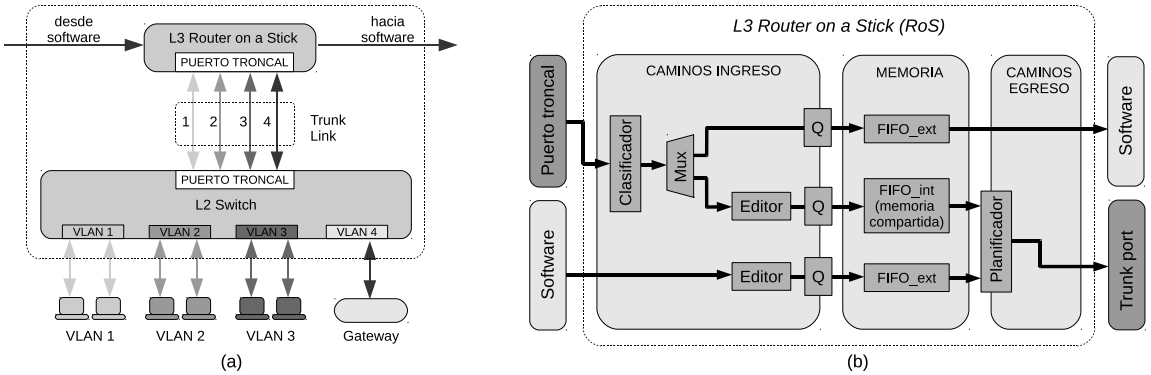


Figura 2.21: Caso de estudio: (a) escenario de aplicación, (b) caminos de procesamiento

combinar los casos de memoria independiente y de memoria compartida. En la Fig. 2.24(a) se muestra la arquitectura de memoria compartida *sm_fifos*, la que consta esencialmente de un bloque de memoria *memory* y un bloque *control* de gestión de punteros para definir las FIFOs dentro de dicha memoria. En su forma más simple, este bloque contiene sólo una fifo mediante la instanciación de dos punteros de read/write respectivamente y un comparador, vinculados a *FIFO_1* en el bloque de memoria. A partir de este caso, y a medida que se agregan mas punteros de read/write, se crean nuevas zonas en el mismo bloque de memoria representando nuevos arreglos de tipo FIFO. El bloque de control genera los requerimientos de servicio (*req*) y recibe los otorgamientos de servicio mediante el puerto *raddr*; y se vincula con el bloque de memoria mediante los vectores de escritura y lectura *waddr_array* y *raddr_array* respectivamente. Para nuestros propósitos, se asigna un espacio máximo a cada nueva FIFO mediante un decodificador *dec*; otros esquemas de asignación dinámica son posibles pero bastante más complejos [54]. Múltiples bloques independientes del tipo *sm_fifos* pueden combinarse mediante un bloque de mayor jerarquía *im_fifos* como muestra la Fig. 2.24(b). Este bloque combina los requerimientos de cada *sm_fifos* y los reporta el planificador; luego divide el vector de otorgamiento de servicios *grant* procedente de éste y lo direcciona al bloque *sm_mem* adecuado mediante codificadores $2^n \rightarrow n$. Asimismo genera los señales de habilitación de lectura *ren* según el bloque *sm_fifos* asociado al grant otorgado. Del lado de escritura, en tanto, la FIFO a activar está directamente dada por el valor de VID del paquete ingresante a *data_in*, activando la posición correspondiente dentro del vector *waddr_array*. Los esquemas de las Figs. 2.24(a) y 2.24(b), además de servir a nuestro caso de estudio, aportan la flexibilidad necesaria para implementar los casos de almacenamiento estudiados en la Sec. 2.4.2.

Las primitivas necesarias fueron implementadas mediante el lenguaje de descripción de hardware *(Hardware description Languaje, HDL)* Verilog, simuladas mediante Mentor ModelSim y sintetizadas mediante la herramienta Altera Quartus II para FPGAs Stratix IIGX, optimizando para un compromiso velocidad/área durante el proceso de compilación. Los reportes corresponden a resultados post-fitting, es decir, considerando los retardos de ruteo resultantes del emplazamiento en el FPGA. En primera instancia se comprobó el uso de recursos y la máxima velocidad de operación. La unidad básica para implementación de lógica combinacional en FPGAs es la tabla de lookup *(Look-Up Table, LUT)*, mientras que los registros son la unidad básica de memoria para pequeños módulos de almacenamiento e implementación de pipelines. Cuando se requiere mayor capacidad de

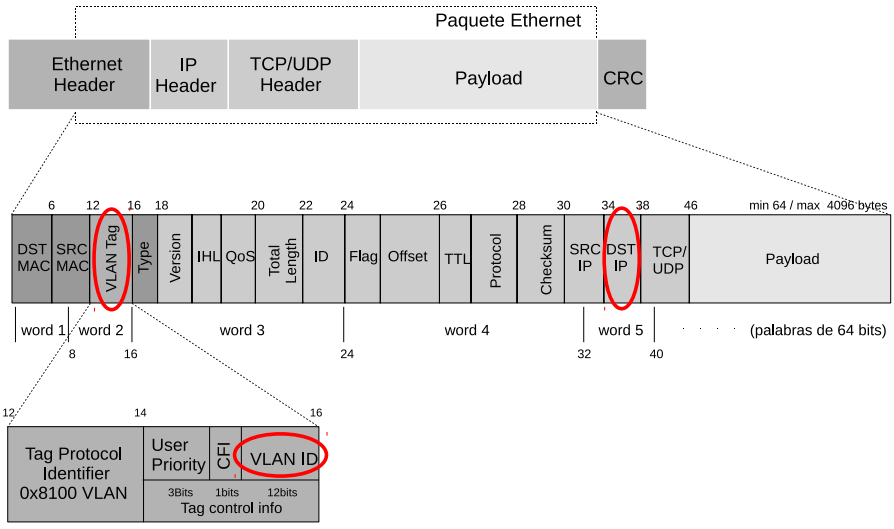


Figura 2.22: Caso de estudio: encabezados involucrados

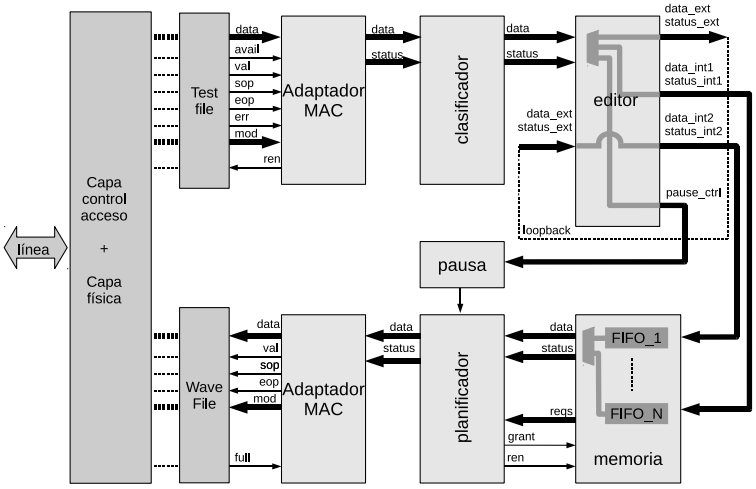


Figura 2.23: Caso de estudio: arquitectura hardware

almacenamiento, los registros no son convenientes y en su lugar se utilizan bloques de memoria RAM (BRAM). En los presentes ensayos interesa evaluar el consumo de LUTs y registros; el uso de BRAM será dependiente del tamaño de FIFOs deseado y puede obtenerse directamente. En cuanto a la velocidad requerida, se resumen en el Cuadro 2.6 las relaciones involucradas. El tamaño mínimo de un paquete Ethernet es de 64 bytes, mientras que en algunos casos se lo lleva a 40 bytes; considerando 8 bits/byte se obtienen las velocidades especificadas. En el Cuadro 2.7 se muestra el uso de recursos y la velocidad alcanzada para cada primitiva en su configuración más simple. El bus utilizado es de 96 bits, 64 de ellos para datos y 32 bits para control. En el Cuadro 2.8, en tanto, se reporta el consumo de recursos para la combinación de estas primitivas en nuestro caso de ensayo, es decir un RoS L3. Además, se considera el consumo de los dos módulos auxiliares *mac_adapter* (Fig. 2.23) y el IP Core MAC para implementación de la capa 2 Ethernet. El RoS, implementado sobre la plataforma de desarrollo, fue ensayado en conjunto con el Switch L2, implementando exitosamente ruteo entre

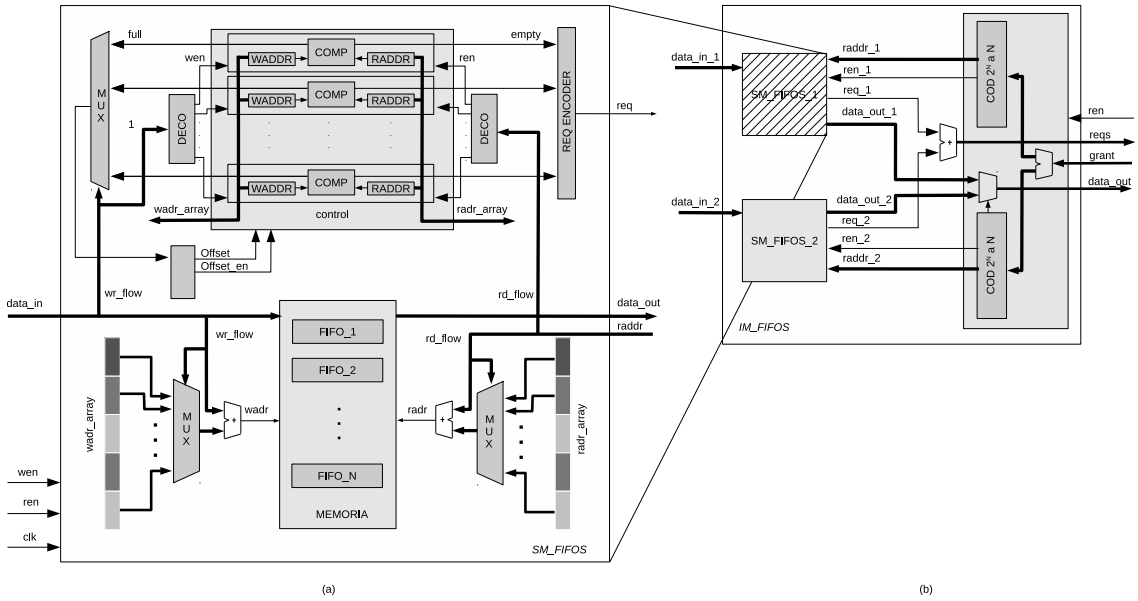


Figura 2.24: Almacenamiento: (a) FIFOs en memoria compartida, (b) FIFOs independientes

Cuadro 2.6: Velocidades máximas Ethernet

	1 GbE	10 GbE	40 GbE	100 GbE
40 bytes	3,125 MHz	31,25 MHz	125 MHz	312,5 MHz
64 bytes	1,953125 MHz	19,53125 MHz	78,125 MHz	195,3125 MHz

múltiples VLANs a velocidades 1GbE. Asimismo, en base a los resultados de síntesis, se observa que también puede soportar operación en redes 10/40 GbE para paquetes mínimos de 40 bytes (31,25 y 125 MHz respectivamente) y en redes 100 GbE para paquetes mínimos de 64 bytes (195 MHz).

En base a los ensayos realizados, se determinó que los principales factores que determinan la escalabilidad de esta arquitectura son (a) el número de *flujos*, y (b) el número de *reglas* de clasificación. En general, cada regla tiene una *acción* asociada que en este caso es el identificador *VID*; mientras que múltiples reglas pueden tener asociado el mismo *VID*.

La cantidad de flujos impacta principalmente en el número de FIFOs necesarias, así como en la complejidad del Planificador; mientras que el número de reglas afecta principalmente al módulo Clasificador. En el Cuadro 2.9 se muestra el consumo particular del esquema de la Fig. 2.24(b) para dos instancias de *sm_fifos* con n_1 y n_2 colas respectivamente de profundidades dadas por *depth*. En nuestro escenario, por ejemplo, n_1 FIFOs se asignarían a paquetes clasificados en hardware, mientras que n_2 FIFOs recibirían paquetes clasificados en software. Como se muestra, este módulo soporta velocidades mayores a 10 GbE, sin embargo su desempeño se ve afectado por la cantidad de FIFOs. Esto puede ser optimizado mediante descripciones HDL de más bajo nivel (más cercanas al hardware final utilizado), sin embargo esto disminuye la flexibilidad de configuración. El Cuadro 2.10, en tanto, muestra el desempeño de la primitiva de planificación al aumentar el número de FIFOs (y de VIDs relacionadas); en esta implementación se utilizó el codificador round-robin basado en árbol de la

Cuadro 2.7: Primitivas de procesamiento: consumo de recursos

Primitiva	Comb. LUTs	Registros	f_{max}[MHz]
Clasificador (8 reglas)	75/72768	238/727183	249.63
Encolador	36	28	196.16
Planificador	185	165	180.25
Editor	37	197	> 300

Cuadro 2.8: Caso de estudio: consumo de recursos

Primitiva	LUTs	Registros
Clasificador (8 reglas)	70	231
Encolador (3)	179	84
Planificador	180	139
Editores (2)	77	327
Sub-total	506	781
IP Core MAC y lógica asociada	1906	1792
Total	2412	2573

Fig. 2.20. Este módulo presenta mejor escalabilidad que el de almacenamiento, en parte debido al uso de descripción HDL de más bajo nivel. Aun para 129 FIFOS (128+1), la velocidad de trabajo supera ampliamente los $31,25$MHz necesarios para trabajar en redes 10 GbE con paquetes mínimos de 40 bytes, llegando incluso a velocidades aptas para trabajo en redes 100 GbE (312,5 MHz@40 bytes).

En cuanto a la cantidad de reglas, el módulo más afectado en su escalabilidad es el *Clasificador*. Si el conjunto de reglas es conocido al momento de realizar la síntesis y no es necesario re-programarlo, se puede optimizar mucho el consumo implementando las reglas como funciones booleanas. De esta forma, la clasificación se implementa en lógica mediante LUTs. Una primera implementación, basada en LUTs y un codificador de prioridad en árbol basado en la implementación utilizada en NetFPGA, arrojó los resultados mostados en el Cuadro 2.11. Mediante esta implementación sumamante básica se pueden soportar hasta 128 reglas a velocidades 40 GbE; sin embargo la flexibilidad y escalabilidad de la misma es sumamente acotada.

2.6. Conclusiones

En el presente Capítulo se plantea una arquitectura general de procesamiento de paquetes en redes de datos, la cual permite abstraerse de aplicaciones particulares y abordar los problemas generales que este procesamiento plantea. En particular, se definen funciones básicas de procesamiento, llamadas *primitivas*, que combinadas eficientemente pueden satisfacer necesidades puntuales de aplicación. Se analizan las tecnologías disponibles y sus compromisos de diseño para la implementación de cada primitiva, abordando luego una prueba de concepto sobre plataformas de lógica reconfigurable

Cuadro 2.9: Primitiva de Almacenamiento: consumo de recursos

$n_1, n_2, depth$	LUTs	Registros	BRAM[Bits]	f_{max}[MHz]
1, 1, 256	171/72768	56/72718	98304/4520448	270
4, 1, 256	272	110	147456	195,16
16, 1, 256	745	326	442368	137,02

Cuadro 2.10: Primitiva de planificación: consumo de recursos

n_1, n_2	LUTs	Registros	f_{max}[MHz]
4, 1	14/72768	111/727183	483,56
8,1	26	123	475,51
16,1	55	147	300,75
32,1	103	210	339,9
64,1	206	291	219,64
128,1	415	483	255,36

Cuadro 2.11: Primitiva de clasificación: consumo de recursos

Reglas	LUTs	Registros	f_{max}[MHz]
16	84/72768	200/72718	351,86
32	93	201	297
64	157	202	202,84
128	221	203	147,73

(FPGAs). Se diseñan e implementan primitivas para el caso de *Redes LAN Virtuales (VLANs)*, comprobando su desempeño en un escenario real. Finalmente se analizan los resultados de síntesis, los cuales permiten apreciar la escalabilidad de estas primitivas. Sobre la base de estos resultados, se pueden apreciar las etapas más críticas para una implementación eficiente, respondiendo los primeros interrogantes de investigación planteados.

El trabajo desarrollado en este Capítulo se vio plasmado en las dos primeras contribuciones de la Tesis Doctoral. La primera de ellas, titulada *Hardware primitives for packet flow processing architectures*, fue presentada en la VII Southern Conference on Programmable Logic SPL 2011. La segunda contribución, titulada *Reconfigurable network processing: the FPGA case*, se realizó en el contexto del 12th Symposium of Technology (AST) de las 40th Jornadas Argentinas de Informática JAIIO 2011.

Capítulo 3

Clasificación mediante TCAMs

3.1. Motivación

En la actualidad, las arquitecturas de red se definen comúnmente en base a los *servicios* soportados sobre la red, más que en base a su topología física. A fin de llevar este concepto a implementaciones eficaces, se deben agregar nuevos esquemas de forwarding al ya tradicional esquema de enrutamiento basado en el prefijo más específico *(Longest Prefix Matching, LPM)* utilizado en redes IP. El elemento crítico en estos esquemas es la función o primitiva de *clasificación*, ya introducida en capítulos previos. La clasificación de paquetes se suele referir también como *layer 4 switching*, enfatizando que las decisiones de enrutamiento se toman considerando capas superiores a la 3 o de *inter-red* donde se encuentran las direcciones IP. Algunos ejemplos de los campos adicionales examinados pueden ser la dirección IP de origen (para proveer servicios diferenciados según la red que realice el *requerimiento*), puertos de origen/destino (para diferenciar tipos de tráfico tales como streaming de audio/video, mails, transacciones seguras, etc.), o Flags TCP (para diferenciar si una conexión se inicia desde dentro o fuera de una red). La función de clasificación posibilita un sinnúmero de nuevos servicios, tales como enrutamiento basado en políticas específicas (policy-based routing), contabilización de volumen de tráfico (traffic accounting), balanceo de carga, control de acceso, o detección de intrusiones (network intrusion detection systems, NIDS). Actualmente, las funciones de clasificación están presentes en la mayoría de los dispositivos de una red de datos, tales como enrutadores, conmutadores, servidores, etc. [55].

La función de clasificación evoluciona a partir de los primitivos cortafuegos o *firewalls*, utilizados originalmente en las interfaces inter-redes a fin de proveer *políticas de seguridad*, eliminando el tráfico de paquetes indeseados entre estas redes. Los firewalls se introducen en los puertos de un equipo de conmutación, e involucran un número reducido de reglas, en el orden de las centenas. Una política típica de un firewall sería permitir requerimientos a cierto recurso de una red desde la misma red, pero no desde fuera de ella. Otra función más reciente de los clasificadores es la de proveer

calidades de servicio diferenciadas dentro de enlaces comunes. Esta función surge de la necesidad de proveer servicio predecible y garantizado sobre una red basada en el mejor esfuerzo tal como es Internet. Para ello, surgen *protocolos de reserva* tales como DiffServ, los cuales permiten reservar anchos de banda y asociarlos a pares origen/destino. Más recientemente, han surgido protocolos tales como la *Conmutación por etiquetas multi-protocolo (Multi-Protocol Label Switching, MPLS)*, que permiten establecer rutas persistentes sobre un esquema de rutas dinámicas tal como el de Internet. Finalmente, la actual migración hacia redes virtuales (Virtual Networks, VNs) que sacan provecho de la virtualización de recursos, y las Redes Definidas por Software (Software Defined Networks, SDNs) como generalización de este concepto, integra todas las técnicas mencionadas, con lo que se observa claramente la relevancia que adquiere el esquema de clasificación. En particular, como se profundizará más adelante, estos clasificadores deberán soportan *más reglas*, considerando a la vez *más campos*.

En trabajos recientes [61] se observa que, si bien este tópico se halla muy desarrollado, los esquemas tradicionales de clasificación se orientan mayormente a aplicaciones de control de acceso y firewalls, con grupos de reglas relativamente pequeños y estáticos [62]; mientras que su desempeño para aplicaciones emergentes en internet es incierto. Las exigencias actuales planteadas al módulo de clasificación provienen de dos fuentes principales, a saber, *velocidad de procesamiento* y *complejidad de procesamiento*. Por un lado, las capacidades de ruteo y conmutación se encuentran en el orden de los Terabits por segundo (Tbps) en datacenters y de cientos de Gigabits por segundo (Gbps) en redes LAN/MAN/WAN, velocidades muy elevadas para los esquemas de clasificación existentes. Por otro lado, a medida que surgen nuevos protocolos y aplicaciones, los rulesets tienden a ser cada vez más complejos y grandes, lo que plantea nuevas exigencias de procesamiento. Las redes existentes en el contexto de data centers como de redes empresariales de mayor tamaño requieren control de flujo de granularidad fina (fine-grained flow control) para enrutamiento entre miles de máquinas físicas y virtuales, mientras que los nuevos protocolos tales como OpenFlow [13] utilizan reglas complejas definidas sobre 10 campos o más.

En este capítulo se aborda la técnica más difundida a nivel industrial para casos donde se requieren garantías de desempeño en clasificación: las Memorias Ternarias Accesibles por Contenido *(Ternary Content-Addressable Memories, TCAMs)*. En particular, se analizan opciones de emulación de TCAM para plataformas FPGA, y sobre esta base se propone una arquitectura genérica. Dicha arquitectura sirve para demostrar las ventajas que puede aportar la emulación de TCAMs contra una implementación nativa de esta memoria, en particular respecto al problema conocido como *expansión de rangos*. Finalmente, se valida el análisis realizado mediante resultados de síntesis sobre FPGAs.

3.2. Aspectos generales

En la Sec. 2.4.2 se analizó la primitiva de clasificación en el contexto de un camino general de procesamiento. En este capítulo, en cambio, se profundizará sobre la función de un clasificador y cómo éste puede implementarse eficientemente.

Un clasificador utiliza el encabezado *(Header)* de un paquete como clave *(Key)* de búsqueda, analizándola contra un conjunto de filtros o *reglas* pre-determinadas, cada una de ellas asociada a un determinado *identificador*. Como consecuencia de este procesamiento, se toma una determinada *acción* sobre el paquete. Comúnmente se asocian bi-unívocamente *reglas* con *acciones*; tal concepto proviene de las reglas IP donde éste es efectivamente el caso. En un contexto general, sin embargo, pueden surgir relaciones más complejas entre reglas y acciones a tomar, como se verá más adelante. El encabezado del paquete o *Key* puede dividirse en múltiples *campos* independientes, en el sentido de que se pueden obtener resultados para cada uno de ellos sin recurrir a los demás, por ello se dice que dichos campos son *ortogonales* unos de otros. Sobre cada uno de estos campos se aplica clasificación uni-dimensional o *lookup*; los resultados de lookup son válidos dentro de los respectivos espacios de búsqueda uni-dimensionales y pueden definir por sí mismos decisiones sobre el paquete. En un contexto de clasificación multi-campo, sin embargo, se deben considerar las *combinaciones* entre tales resultados a fin de tomar una acción.

3.2.1. Requerimientos de clasificación

Actualmente, los requerimientos más importantes impuestos a un sistema de clasificación son, en términos generales:

- procesamiento determinístico a velocidad de línea
- adaptabilidad para búsqueda según valores exactos, prefijos o rangos arbitrarios
- adaptabilidad para entrega de resultados best-match y multi-match
- soporte de actualización incremental (o dinámica)

Procesamiento determinístico a velocidad de línea significa que se debe tomar una nueva decisión y despachar un nuevo paquete con el mismo ritmo que arriban paquetes al sistema, sin depender esta velocidad del camino de búsqueda seguido. Esto no implica necesariamente que el procesamiento de deba realizar en un ciclo de reloj; este factor se aprovecha para aplicar esquemas de pipelining que, si bien introducen cierta latencia entre el ingreso y egreso de paquetes, permiten mayor tasa de procesamiento. Por supuesto que durante este tiempo los paquetes que están siendo procesados se deben almacenar dentro del sistema. Ya que el tamaño de paquetes es variable, se toma el peor caso que es el paquete de mínimo tamaño. Para el caso de redes ethernet abordado en esta tesis, este caso corresponde a un paquete de 40 bytes, o sea 3.2 ns/paquete para redes Ethernet de 100 Gbps.

En general, la operación de lookup se define en base a *rangos generales (General Range, GR)* de valores del campo considerado. Entre estos rangos generales, se pueden identificar cuatro casos especiales: *prefijos (PrefiX, PX), valores exactos (EXact value, EX), rangos arbitrarios (Arbitrary Range, AR), y wildcards (WildCard, WC)*. Los prefijos son rangos que se extienden entre dos potencias de dos, por lo que se pueden representar mediante cadenas de bits de la forma $b_{m-1}, b_{m-2}, ...b_p, X_{p-1}, X_{p-2}, ..., X_0$, donde $b_{m-1}, b_{m-2}, ..., b_p$ son bits especificados y $X_{p-1}, X_{p-2}, ..., X_0$

son bits no especificados o *don't cares*. Los bits de don't care se encuentran siempre en las posiciones menos significativas del prefijo. Los valores exactos son rangos donde los límites superior e inferior son iguales, mientras que los wildcards son rangos que cubren todo el espacio de búsqueda. Los rangos arbitrarios, en tanto, pueden abarcar cualquier intervalo dentro del espacio de búsqueda. Estos casos de rango han encontrado tradicionalmente aplicación para clasificación en base a *quíntuplas IP* que abarcan los campos SrcIP/DstIP, SrcPT/DstPT y Protocolo en un encabezado IP. Los prefijos son extensivamente utilizados en la capa 3 (L3 en el stack TCP/IP) para enrutamiento CIDR (Classless-InterDomain Routing). Los valores exactos se utilizan típicamente para especificar hosts y filtrado basado en protocolos en L3, así como para filtrado basado en puertos en la capa de transporte (L4 en el stack TCP/IP). Los wildcards especifican decisiones por defecto para paquetes que no satisfacen ninguna de las restantes reglas del ruleset. Los rangos arbitrarios, en tanto, se utilizan generalmente para identificar sesiones entre determinados puertos de capa L4. Como veremos, cada caso plantea distinta dificultad; en este trabajo se analiza en particular el caso de AR que es capaz de cubrir todos los demás.

Las aplicaciones actuales, por otro lado, suelen requerir resultados basados en múltiples coincidencias *(Multi-Match, MM)*, a diferencia de las tradicionales aplicaciones para ruteo IP que requerían resultados del tipo mejor coincidencia *(Best Match, BM)*. Es decir, la acción a tomar sobre el paquete depende de la *combinación particular* de reglas que se cumplen para ese paquete. La clasificación MM permite, por ejemplo, mantener múltiples contadores con el fin de relevar volumen de tráfico y realizar la correspondiente facturación de servicio. Otro ejemplo son los sistemas de detección de intrusiones (NIDS) como SNORT [63], los que controlan el tráfico de una red para detectar intrusiones maliciosas y ataques del tipo de denegación de servicio *(Denial of Service, DoS)*. De este modo, si bien los esquemas BM encuentran abundante aplicación en la actualidad, es muy deseable que las nuevas propuestas sean capaces de entregar resultados MM, y BM como un caso particular de aquéllos. Algunos trabajos previos [64] [65] ofrecen una completa discusión de las aplicaciones de la clasificación MM, así como técnicas especialmente orientadas a implementación en TCAM nativa, la cual intrínsecamente entrega resultados BM. En particular, como se discute en los Capítulos 4 y 5, la obtención de resultados multi-match es indispensable para la implementación de clasificación multi-dimensional por descomposición.

La *actualización dinámica* del ruleset, en tanto, no era un requerimiento en el pasado, ya que las reglas de ruteo eran relativamente estáticas. Las aplicaciones actuales y la utilización de plataformas virtuales altamente dinámicas [66], sin embargo, requieren que este aspecto se considere en nuevos diseños. En un esquema de clasificación general, el ruleset resulta afectado en dos momentos principales:

1. durante la construcción del ruleset con un conjunto inicial de reglas, llamaremos a esta operación *almacenamiento*

2. durante la operación normal, al agregar o borrar reglas; llamaremos a esta operación *actualización incremental o dinámica*

La operación de almacenamiento se realiza sólo una vez por lo que no tiene requerimientos especiales, siempre que su complejidad permita su implementación. La operación de actualización dinámica, en cambio, puede requerirse frecuentemente sobre el sistema en operación, en especial considerando los dinámicos esquemas de red actuales. La complejidad de ambas operaciones depende fundamentalmente del método de búsqueda empleado y sus estructuras de datos asociadas. La actualización incremental consiste esencialmente en la adición o sustracción de una regla, la cual puede ocasionar una o múltiples escrituras en las estructuras utilizadas. Las posibles operaciones de actualización son *set, clear y modification*; en la primera de ellas se adiciona una nueva regla, en la segunda se elimina una regla existente, mientras que en la tercera se modifica la especificación de una regla ya existente en el ruleset. El procesamiento asociado a una operación de actualización se implementa comúnmente en plataformas GPP de menor desempeño pero con gran capacidad de cómputo y recursos de hardware/memoria, mientras que los resultados se almacenan en arquitecturas de máximo desempeño y recursos limitados como son las FPGAs.

3.2.2. Clasificación mediante TCAMs

A continuación se investiga una primera opción para realizar clasificación multi-match sobre rangos arbitrarios en FPGAs, basada en técnicas de *búsqueda exhaustiva*. En particular, se evalúan los compromisos de implementar arrays asociativos, se aporta un análisis y comparación de las opciones existentes para emular TCAM en FPGAs, y se obtienen resultados de implementación a fin de evaluar objetivamente sus capacidades.

Las técnicas de búsqueda exhaustiva evalúan cada regla en forma independiente, por lo que tienen requerimiento de memoria predecible e igual al de las reglas originales, que aumenta linealmente con el número de reglas N. La implementación puramente secuencial de esta técnica es la *búsqueda lineal*, cuyo tiempo de búsqueda es directamente proporcional a N. La versión puramente concurrente de la búsqueda exhaustiva, en el extremo opuesto, es la *búsqueda asociativa*, que minimiza el tiempo de búsqueda aumentando notablemente los recursos lógicos requeridos. En particular, las memorias accesibles por contenido *(Content Addressable Memories, CAMs)* y su versión ternaria *(Ternary CAMs, TCAMs)* implementan búsqueda de múltiples coincidencias en tiempo constante mediante la comparación de cada bit del encabezado contra cada bit de las N reglas. Las CAMs incorporan memoria SRAM para el almacenamiento de las reglas individuales, así como comparadores de igualdad para cada bit almacenado. Las TCAMs, en tanto, requieren almacenamiento adicional para enmascarar ciertos bits que no son relevantes para la búsqueda (don't care). En la Fig 3.1 se muestra la arquitectura general de una memoria TCAM nativa, conteniendo 3 reglas de 3 bits cada una. Las celdas de TCAM C_{ab} se disponen en un arreglo de a filas por b columnas. A cada fila le corresponde una *línea de coincidencia* ML_a, la cual alimenta a su salida un amplificador de sensado. Asimismo, a cada columna b le corresponde un par de líneas de búsqueda $SL_b, /SL_b$ conteniendo respectivamente los valores original y negado de un bit del key. Cada celda de TCAM contiene (a) el bit de estado, (b) el bit de *don't care*, y (c) un comparador de igualdad. Los bits son almacenados comunmente en celdas de memoria SRAM; el comparador, que básicamente implementa una operación XOR, puede por su lado implementarse mediante compuertas NAND o NOR. La operación de búsqueda

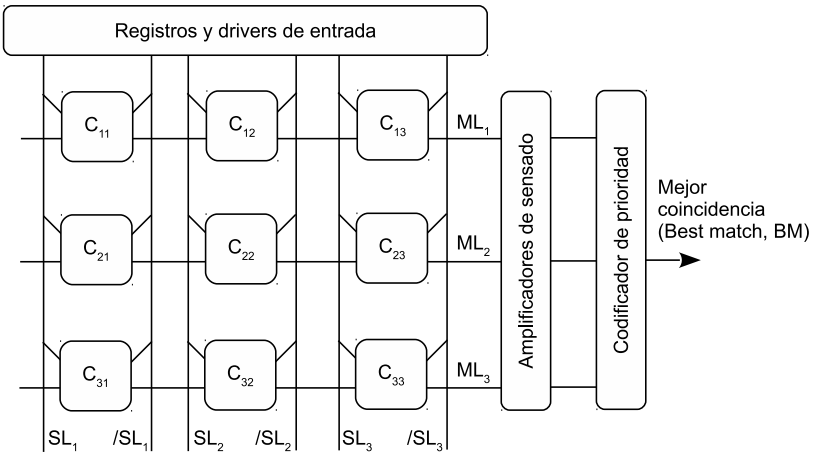


Figura 3.1: Arquitectura de TCAM nativa

comienza pre-cargando las líneas de coincidencia a estado alto, mientras que los valores del key se capturan en los registros de entrada. Luego, los drivers de entrada propagan el valor del key hacia las correspondientes columnas, por lo que las celdas afectarán sus respectivas líneas de coincidencia según los resultados de sus comparaciones. De este modo, las líneas de coincidencia donde todas las celdas arrojen resultado positivo permanecerán en estado alto, mientras que aquéllas donde al menos una celda arroje resultado negativo serán descargadas pasando a estado bajo. El resultado de cada línea es detectado por los amplificadores se sensado, mientras que el codificador de prioridad de salida selecciona la línea de mayor prioridad que haya resultado positiva.

De las múltiples coincidencias presentes, los chips de TCAM entregan en general sólo una; ésta se selecciona internamente según el orden en que se encuentra almacenada (first match) o en base a algun orden pre-establecido (weighted match) [67]. Para el primer caso, el más común, la TCAM incluye un codificador de prioridad. De este modo, se observa que no es directamente posible extraer las múltiples coincidencias (vector de coincidencias) de un chip TCAM. Para ello se debe recurrir a técnicas que re-organizan las entradas de memoria de acuerdo a los solapamientos de reglas [67], modifican los circuitos internos de la TCAM o generan grupos disjuntos de reglas para mapearlos a TCAMs independientes [68]. Estas técnicas requieren de múltiples ciclos de búsqueda o utilizan codificaciones basadas en el patrón de reglas, lo que actúa a su vez en contra de las ventajas de una TCAM. Asimismo se requiere re-ordenamiento de reglas durante la operación de actualización, lo que puede resultar muy costoso; a este respecto existe asimismo abundante trabajo [69].

Las memorias TCAM son actualmente la tecnología preferida en equipos comerciales cuando se requieren garantías de desempeño sin depender del ruleset y actualización de reglas en servicio. Sin embargo, las TCAMs son costosas y su escalabilidad es limitada al aumentar la cantidad de reglas N o el ancho de campos M. A diferencia de las SRAMs, donde sólo una fila del array de memoria se encuentra activa a la vez; en una TCAM todas las posiciones de memoria se encuentran activas para cada búsqueda, ello ocasiona alto consumo de energía. Además, las celdas de TCAM demandan mayor área de silicio que una celda equivalente de SRAM, por lo que su escalabilidad es limitada [70] [71]. Las TCAM son efectivamente aplicadas para LPM basado en PX; sin embargo para el caso

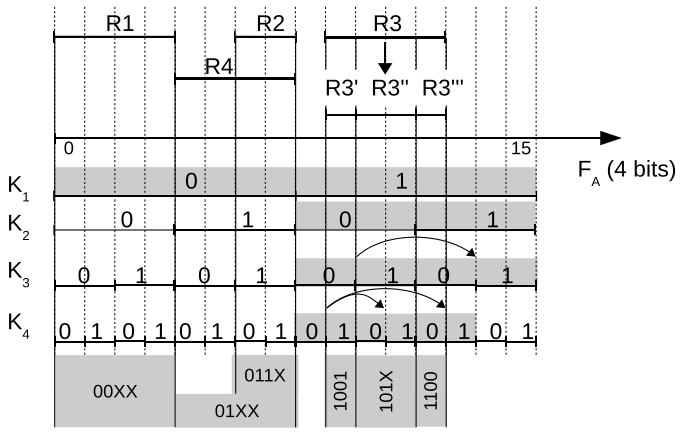


Figura 3.2: Caso de expansión de rango en una TCAM

de ARs sufren el problema de *expansión de rangos*. Esto es debido a que una fila de TCAM puede almacenar sólo un valor exacto o un prefijo; un AR debe en cambio implementarse expandiéndolo previamente en múltiples prefijos que se almacenarán en múltiples filas de TCAM. De este modo, una regla que involucre ARs en alguno de sus campos puede requerir hasta w filas o *entradas* de TCAM, donde w es el ancho de direccionamiento que involucra el AR (por ejemplo, $w = 4$ si un AR abarca el intervalo de 4 bits $[0001, 1100] = [1, 12]$). Para mitigar estas desventajas se han introducido mejoras de diseño a nivel de circuito [71] [72], así como optimizaciones basadas en análisis de reglas reales [68] [73] [74] [75] [76].

En la Fig. 3.2 se ilustra claramente el problema de expansión de rangos. En este caso, se desea almacenar en una TCAM cuatro reglas R1, R2, R3 y R4 con ancho de key $M = 4$ $(K_1, ..., K_4)$. R1, R2 y R4 son reglas basadas en prefijos y por lo tanto cada una de ellas ocupa una fila de TCAM. La regla R3, sin embargo, es un AR por lo que presenta expansión de rango. Ésta se puede apreciar al considerar los espacios de búsqueda cubiertos por $K_1, ...K_4$ según su peso en el key. Un rango puede ser totalmente cubierto por una entrada de TCAM si, para cada bit del key, este rango abarca (a) un espacio menor al cubierto por cualquier bit del key (prefijo de longitud M, es decir EX), o (b) un espacio múltiplo exacto del espacio cubierto por cualquiera de los bits del key (por ejemplo la regla R4 cubre múltiplos exactos de los espacios de K_4 y K_3, por lo que estos bits son don't care). La regla R3, en cambio, cubre la mitad del rango de K_4 en sus extremos. Ya que las coincidencias se resuelven mediante AND entre los resultados de comparación de cada bit, R3 se debe sub-dividir en tres prefijos 1001, $101X$, y 1100 como se muestra, expandiendo efectivamente en tres reglas y sumando un total de 6 reglas para implementar las 4 originales.

3.3. Diseño propuesto

3.3.1. Recursos tecnológicos

En general, una FPGA ofrece los siguientes recursos de procesamiento para implementar clasificación de paquetes:

- Tablas de búsqueda (lookup): estos elementos, llamados comunmente *Lookup Tables (LUTs)*, permiten implementación de funciones lógicas combinacionales. Las LUTs se implementan sobre memoria SRAM, la que se graba o *configura* al actualizar el diseño y puede luego leerse durante la operación del circuito. Si bien se contempla la posibilidad de *reconfiguración parcial* durante la operación, esta memoria no está prevista para realizar escrituras frecuentes. Las LUTs poseen líneas de direccionamiento que constituyen las variables de entrada a la función lógica implementada; el número de estas líneas depende de la familia de FPGAs y es una consideración de diseño crítica para el máximo aprovechamiento de recursos. En FPGAs actuales, la memoria que constituye la LUT es de 1 bit de ancho y almacena así el estado de la variable de salida para cada combinación de las variables de entrada según la función lógica definida. Por ejemplo, para la familia Stratix V de Altera, las LUTs pueden tener hasta 6 líneas de direccionamiento, siendo capaces de implementar funciones lógicas de 6 entradas. Para mayor cantidad de entradas, se combinan múltiples LUTs, con costo adicional en cuanto a retardo de propagación. La cantidad de LUTs en una FPGA actual de alto desempeño tal como la familia Altera Stratix V se encuentra en el orden de 370000.

- Registros síncronos (flip-flops): estos elementos constituyen la unidad básica de almacenamiento de estados del FPGAs, permitiendo cada uno almacenar un bit. Ya que se encuentran distribuidos en toda el área del chip, e íntimamente relacionados con las LUTs, permiten implementar arquitecturas altamente concurrentes mediante técnicas de pipelining, las cuales esencialmente segmentan un circuito complejo en múltiples secciones más pequeñas operando concurrentemente. Mediante diseños adecuados, esto permite aumentar notablemente la velocidad de operación de tales circuitos, donde el retardo de procesamiento queda determinado por la etapa más lenta del pipeline, llamada *camino crítico*. Los registros también se utilizan, en conjunto con circuitos de direccionamiento, para implementar memorias de tamaño reducido o bancos de registros *(regfiles)*. La cantidad de registros en FPGAs actuales se encuentra en el orden de 700000.

- Bloques de memoria SRAM: sirven a la implementación de memorias de mayor capacidad que las que se podrían implementar eficientemente mediante registros, ya que contienen circuitos de direccionamiento internos especializados y celdas SRAM de alta densidad. Los esquemas basados en registros implementan los circuitos de direccionamiento y reloj mediante los recursos de ruteo propios del FPGA, con retardos de propagación excesivos a partir de cierto tamaño de memoria. De este modo, los bloques de memoria SRAM, comunmente llamados BRAM, permiten obtener tamaños de memoria para los cuales una implementación basada en registros es impracticable o muy ineficiente. Las BRAMs tienen distintos tamaños según la familia de

Cuadro 3.1: Recursos generales disponibles en familias sucesivas de FPGAs

Familia	LUTs	Registros	Tipos BRAM	BRAM (bits)
Stratix II	106032	106032	M512/M4K/MRAM	6747840
Stratix III	270400	270400	M9K/M144K/MLAB	16662528
Stratix IV	424960	424960	M9K/M144K/MLAB	21233664
Stratix V	450800	901600	M20K/MLAB	54476800

FPGAs; comunmente se preveen bloques de dos o tres tamaños diferentes a fin de aprovecharlos correctamente segun la aplicación. Las FPGAs Altera Stratix V, por ejemplo, incorporan alrededor de 2000 BRAMs de 20Kbits cada una.

- Bloques especializados: estos elementos son básicamente ASICs embebidos, los que permiten implementación de funciones muy comunes en forma más eficiente que la que permitirían los recursos lógicos generales del FPGA. Ejemplos son los bloques DSP de precisión variable para aplicaciones en procesamiento de señales, módulos de lazo enganchado en fase *(Phase-Locked Loops, PLLs)*, bloques para comunicación PCI Express, y transceptores de alta velocidad. Actualmente también se incorporan bloques del tipo *Embedded Hardcopy*, permitiendo implementar diseños propietarios embebidos con las características de un ASIC.

En la Fig. 3.3 se muestran los componentes generales de un FPGA Altera Stratix V. Los elementos fundamentales para aplicaciones generales son los *Bloques de matriz lógica (Logic Array Block, LAB)*, los cuales se interconectan mediante recursos de enrutamiento en jerarquías según sus longitudes y correspondientes retardos de propagación, tal como se observa en la Fig. 3.3(a). Cada LAB concentra recursos lógicos bajo la forma de LUTs, multiplexores, sumadores, lógica de acarreo y registros como muestra la Fig. 3.3(b). Los detalles de esta arquitectura responden a optimizaciones orientadas a implementar las aplicaciones más comunes en forma eficiente. En el Cuadro 3.1, en tanto, se observan los recursos de propósito general disponibles en los dispositivos utilizados en nuestros ensayos. En los ensayos del Cap. 2 se utilizó la familia Altera Stratix II por ser la plataforma disponible en ese momento. En los demás ensayos de esta Tesis, en tanto, se adopta un dispositivo de la familia Altera Stratix V, por representar la generación inmediata anterior a la que se encuentra en proceso de difusión en la actualidad (Stratix 10).

Los diseños orientados a FPGAs se describen mediante lenguajes especiales, llamados *Lenguajes de Descripción de Hardware (Hardware Description Languages, HDLs)* como VHDL o Verilog, y se portan a la FPGA utilizada mediante herramientas de compilación o *síntesis* de hardware, las cuales permiten explotar eficientemente los recursos y arquitectura de cada FPGA en particular. Se pueden utilizar asimismo otros lenguajes, más cercanos a los lenguajes comunes para implementación de software, llamados *Lenguajes de Alto Nivel (High Level Languages, HLLs)*; sin embargo esta alternativa requiere más trabajo por parte de las herramientas de optimización. Para los propósitos de esta Tesis se utilizaron HDLs.

Las FPGAs pueden utilizarse en conjunto con chips TCAM externos para realizar clasificación en redes; sin embargo esta opción requiere una interfaz de comunicación entre ambos chips que limita

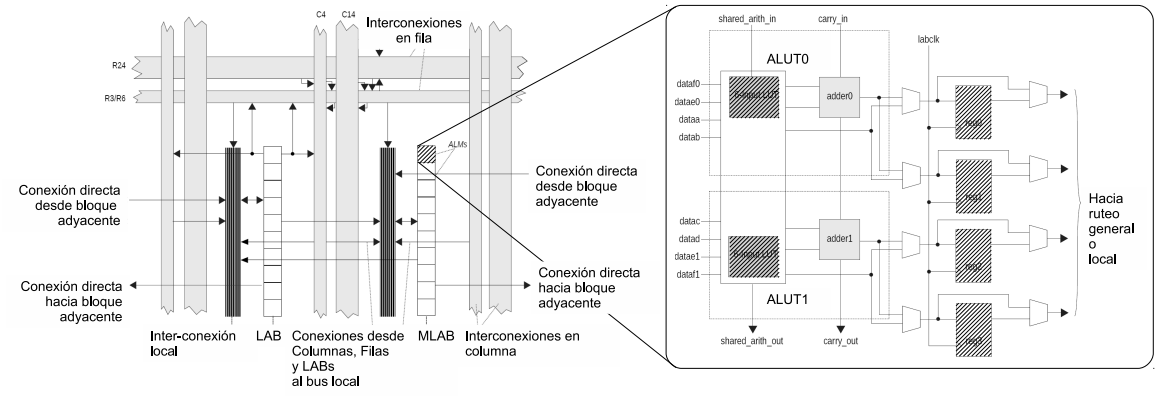


Figura 3.3: FPGAs Altera Stratix V: (a) arquitectura general, (b) módulo aritmético-lógico (ALM)

su velocidad. En antiguos FPGAs, por ejemplo la familia APEX de Altera [77] [78], se incluyeron núcleos o cores embebidos de *TCAM nativa*, es decir silicio especialmente dedicado a esta función; sin embargo estos núcleos no se incluyen actualmente por su alto costo en área de silicio y lo específico de su aplicación. Para reemplazarlos, los FPGAs ofrecen actualmente abundantes recursos lógicos y de memoria que pueden utilizarse estratégicamente en arquitecturas para *emulación de TCAM*. Estas arquitecturas ofrecen además la posibilidad de acceder a las distintas etapas internas de una TCAM más allá del resultado final de clasificación, como por ejemplo los resultados individuales de comparación, aportando mayor flexibilidad para satisfacer necesidades especiales de clasificación. Los primeros trabajos en proponer emulación de TCAM en FPGAs [79] disponían de muy limitados recursos; mientras que propuestas más recientes explotan mejores posibilidades y desempeño [80] [81] [82] [83].

3.3.2. Arquitectura de emulación de TCAMs

Las FPGAs ofrecen dos recursos principales para emular TCAM [84]. La primera opción utiliza registros, logrando funcionamiento exactamente análogo al de la TCAM nativa. A diferencia de los chips de TCAM, sin embargo, esta implementación requiere gran cantidad de registros dispersos en el área del FPGA, utilizando recursos de ruteo para su inter-conexión con altos retardos de propagación; esto limita severamente su escalabilidad. La segunda alternativa consiste en utilizar bloques SRAM del FPGA; en este caso el puerto de dirección *(address)* de lectura se conecta al puerto *key* de la TCAM emulada, mientras que el puerto *data* de lectura representa el puerto de coincidencias *(match)* con los resultados de la TCAM emulada. De esta forma, el espacio de valores del *key* resulta en el espacio de direccionamiento de la RAM, mientras que el dato almacenado en cada posición de RAM representa el vector de coincidencias para ese valor del key. En la Fig. 3.4(a) se observa la operación de búsqueda de un $key = 110$ en una TCAM nativa conteniendo las reglas $110, 10X$ y $X1X$, resultando en el vector de coincidencias 101. La Fig. 3.4(b) muestra como sería una hipotética emulación exacta de esta arquitectura mediante direccionamiento de pequeñas memorias de dos posiciones. En el caso de que esta TCAM sea emulada mediante RAM según el esquema de direccionamiento analizado, la implementación resulta en la Fig. 3.4(c). En este ejemplo, el key

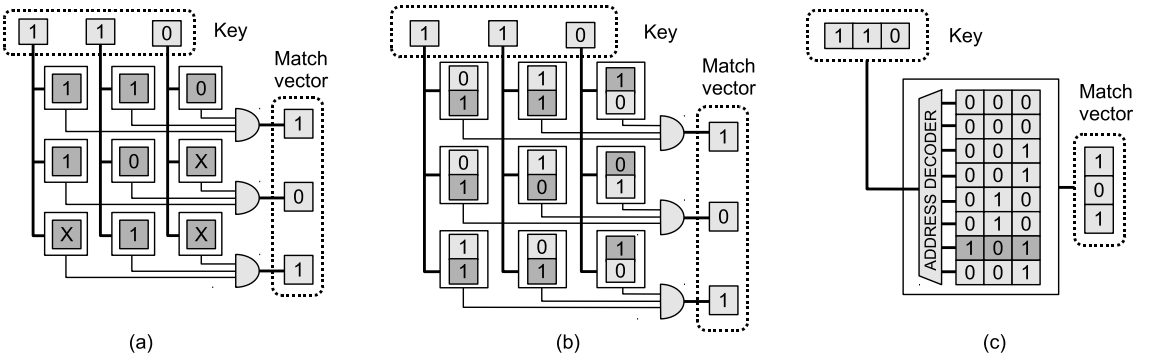


Figura 3.4: FPGAs Altera Stratix V: (a) TCAM nativa, (b) emulación exacta, (c) emulación mediante memoria RAM

110 accede una palabra de memoria 101, resultando en un comportamiento equivalente a la TCAM nativa.

En el ejemplo simple de la Fig. 3.4 se advierte el problema de la implementación mediante memoria, que denominamos *expansión de memoria*. Para analizar este problema, consideramos en la Fig. 3.5 un key de ancho M y una regla, es decir $N = 1$. Si se utiliza un solo bloque de memoria como muestra la Fig. 3.5(a), se pueden almacenar en forma independiente los vectores de match para cada valor del key. Si bien en este caso se elimina completamente la expansión de rangos presente en las TCAMs, el espacio de direccionamiento es 2^M, conduciendo a *expansión completa de direccionamiento* de M a 2^M. Esta expansión se puede reducir dividiendo el key en segmentos de ancho w, y direccionando M/w bloques de memoria independiente con espacios de direccionamiento 2^w. De este modo, se puede alcanzar el extremo opuesto donde $w = 1$, es decir se direccionan M bloques de memoria con espacio de direccionamiento 2^1 cada uno. Este último caso, ilustrado en la Fig. 3.5(c), no presenta expansión de direccionamiento pero sufre de *expansión completa de rangos* emulando exactamente el comportamiento de una TCAM. Entre ambos extremos se pueden definir casos intermedios, desde $\frac{M}{M-1}$ bloques de 2^{M-1} posiciones cada uno hasta $M/2$ bloques de 2^2 posiciones cada uno, ilustrados en conjunto por la Fig. 3.5(b). En resumen, esta *emulación con segmentación* demanda $M/w \times 2^w$ bits de memoria, por lo que la *expansión de direccionamiento* resultante es $(2^w)/w$.

Los casos de búsqueda por prefijo y exacto no se ven afectados por la configuración de segmentación adoptada, a diferencia del caso de rango arbitrario. Para este caso, surge un compromiso *rango vs. expansión de direccionamiento*. Supongamos un caso general donde cada bloque de memoria resultante de la segmentación reduce el ancho de direccionamiento M a un ancho reducido m. En general, se observa que una TCAM emulada mediante bloques de memoria de 2^m posiciones no sufre expansión de rango para rangos de ancho $w \leq m$. De este modo, cuanto mayor es m, mayor es el rango máximo soportado a costa de mayor consumo de memoria por la expansión de direccionamiento $(2^m)/m$. Por otro lado, al considerar N reglas, se requiere ancho de palabra N. Para un bloque disponible de memoria de ancho n, simplemente se concatenan $\lceil N/n \rceil$ de estos bloques. Todo esto queda resumido en los casos generales de las Figs. 3.6(a), 3.6(b) y 3.6(c).

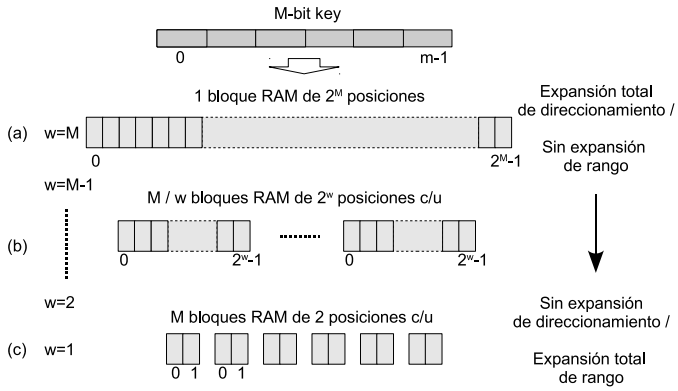


Figura 3.5: Emulación de TCAM: Casos de expansión de direccionamiento

En el caso de FPGAs, los bloques de memoria RAM se ofrecen con capacidades pre-definidas, pudiendo configurar los anchos de puerto de direccionamiento (m) y datos (n) para ellos. La familia de FPGAs *Stratix* de Altera, por ejemplo, ofrece LABs orientados a memoria (MLABs) conteniendo 640 bits para implementar memorias chicas y poco profundas, así como bloques M4K/M9K/M20K según el dispositivo para implementaciones de propósito general. Para estos bloques, se pueden utilizar diferentes *modos* definidos por su relación *profundidad* $\times$ *ancho* ($2^m \times x$), o correspondientes *factores de forma* definidos por la relación m/x. Así, por ejemplo, los bloques M9K se pueden configurar en modos desde 8Kx1 hasta 256x36; mientras que los bloques M20K pueden configurarse en modos desde 16Kx1 hasta 512x 40. Si la TCAM a emular excede la capacidad de un bloque RAM, se unen varios de ellos mediante segmentación horizontal y vertical.

En la Fig. 3.7(a) se observa el esquema de emulación general mediante RAM, con especial énfasis en el mapeo de puertos de la CAM emulada. En la Fig. 3.7(b), en tanto, se replica el esquema de la Fig. 3.7(a), enfatizando la arquitectura de implementación mediante segmentaciones horizontal y vertical. Cada BRAM en la matriz evalúa una sección del key contra una sección del ruleset, entregando como consecuencia una sección del vector de resultados. En este ejemplo, se emula una TCAM de 64 reglas de 16 bits cada una (N=64, M=16) mediante bloques BRAM M9K con factor de forma 8/32. Durante la operación de búsqueda se ingresa con el valor del encabezado en el puerto *key* y se obtienen las coincidencias en el puerto *match*, resultantes de la conjunción de los resultados a lo largo de cada fila de BRAMs. Un codificador de prioridad opcional entrega la regla de mayor peso. Cabe mencionar que, al igual que una TCAM nativa, la operación de búsqueda se realiza en un ciclo de clock.

Para el ejemplo de la Fig. 3.7(b), una implementación nativa requeriría $M \cdot N = 16 \cdot 64 = 1K$ celdas TCAM, mientras que la implementación sin segmentación de BRAM requiere $2^M \cdot N = 2^{16} \cdot 64 = 4M$ celdas SRAM. Emulando mediante cuatro bloques M9K de 256x32 se tiene un compromiso óptimo de uso de BRAM, resultando en $32Kb$ de memoria BRAM (512 bits por regla), con soporte de rangos de hasta 8 bits (intervalos de 256 posiciones).

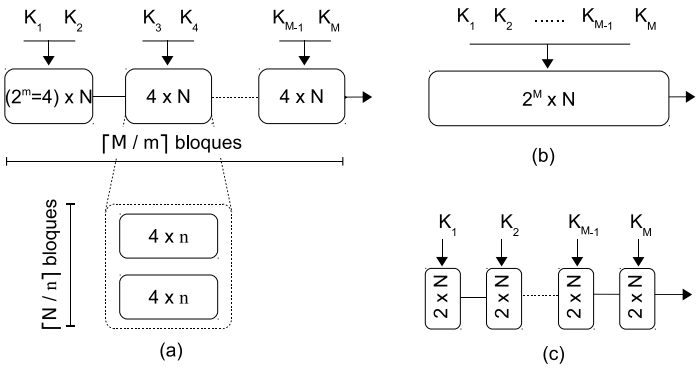


Figura 3.6: Expansión de direccionamiento: (a) caso general, (b) expansión total, (c) sin expansión (TCAM nativa)

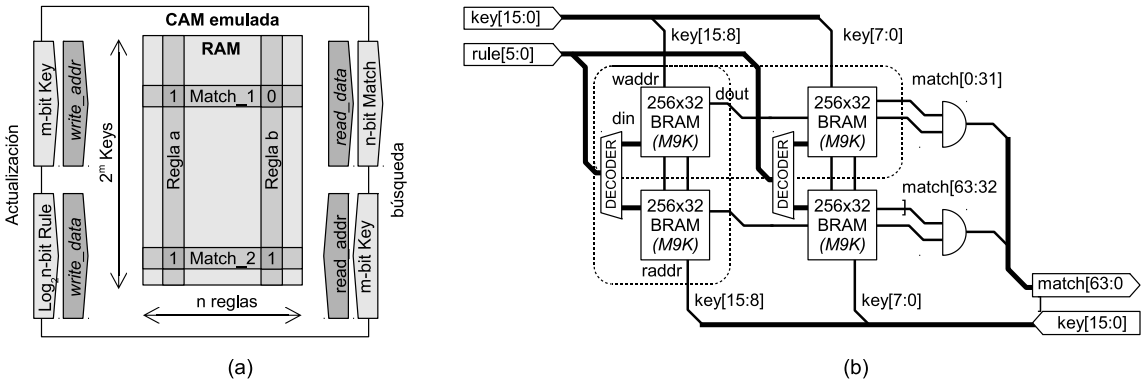


Figura 3.7: Implementación de TCAM emulada: (a) esquema general, (b) arquitectura de segmentación

3.3.3. Actualización dinámica

La actualización dinámica de TCAMs es inherentemente simple y no requiere pre-cómputo alguno, ya que cada regla posee una línea de TCAM directamente asociada. Esta característica se encuentra presente también en el caso de las TCAMs emuladas, si bien la operación puede tomar múltiples ciclos de escritura.

Durante la operación de actualización dinámica de reglas, se ingresa con el ID de regla *rule* (6 bits para nuestro ejemplo de 64 reglas) el cual es decodificado activando sólo una fila de BRAMs. A la vez, se ingresan los valores del key para los que la regla es válida, almacenando unos en dichas posiciones de memoria. Ya que las columnas de bloques BRAM se actualizan concurrentemente, esta operación requiere 2^m escrituras como máximo para un caso de actualización del ruleset completo. Existen otras estrategias para acelerar esta actualización [85] [86], sin embargo se utilizó ésta por su generalidad y por garantizar tiempo máximo 2^m en el peor caso de actualización.

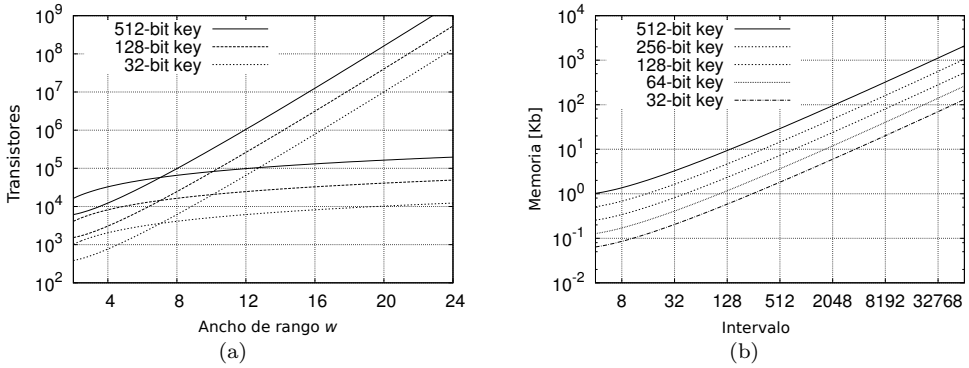


Figura 3.8: Emulación de TCAM: (a) Consumo de área, (b) Consumo de memoria RAM

3.4. Resultados

3.4.1. Consumo de recursos

Como primera estimación, se compara el consumo de área en un chip de una TCAM nativa vs. la versión emulada. Una celda de TCAM consume típicamente 16 transistores debido a la necesidad de almacenar tanto los bits especificados como los bits de don't care. Una celda de SRAM, en tanto, consume típicamente 6 transistores. Además, la codificación de un rango de ancho w en TCAM puede expandir en w reglas en el peor caso [76]. Sobre esta base, se estima el consumo de área de una TCAM en $w \times M \times 16$, mientras que el consumo para la misma TCAM emulada en SRAM es $\frac{M}{w \times 2^w \times 6}$. Estas métricas se observan para el caso de una regla (N=1) en la Fig. 3.8(a), considerando rangos $1 \leq w \leq 24$ y anchos de key $M = 32, 128, 512$. Se observa que, de acuerdo a las estimaciones realizadas, el consumo de TCAM es lineal con w mientras que el de SRAM sigue una ley exponencial. Para rangos grandes, el costo de implementación con un bloque RAM único puede ser excesivo comparado con el costo de expandir rango en TCAM, sin embargo para rangos $w \leq 8$ la expansión de direccionamiento puede resultar mas conveniente que la expansión de rangos. El ancho de key no modifica estas tendencias generales sino los valores absolutos de consumo de recursos.

En la Fig. 3.8(b) se muestra el consumo de memoria RAM para rangos típicos presentes en rulesets reales. Intervalos pequeños de hasta 512 posiciones (9 bits), suficientes para puertos, prefijos y valores exactos, consumen entre 1Kbits y 100 Kbits por regla. Para intervalos más amplios, entre 512 y 32K posiciones (9 a 15 bits), se requieren 1-100 Kbits por regla. La Fig. 3.9(a), en tanto, considera distintos anchos de key M y números de reglas N para un caso general de $w = 9$, suficiente para gran cantidad de casos reales. Las curvas mostradas muestran el consumo de memoria para $N = 256, 1024, 4096$, mientras que los puntos llenos muestran los resultados de síntesis para un FPGA Altera Stratix V utilizando M20K BRAMs en configuración 512x40. En este caso se puede apreciar la tendencia lineal con el número de reglas y el ancho de key, confirmando los resultados estimados mediante resultados post-fitting de síntesis para dispositivos actuales.

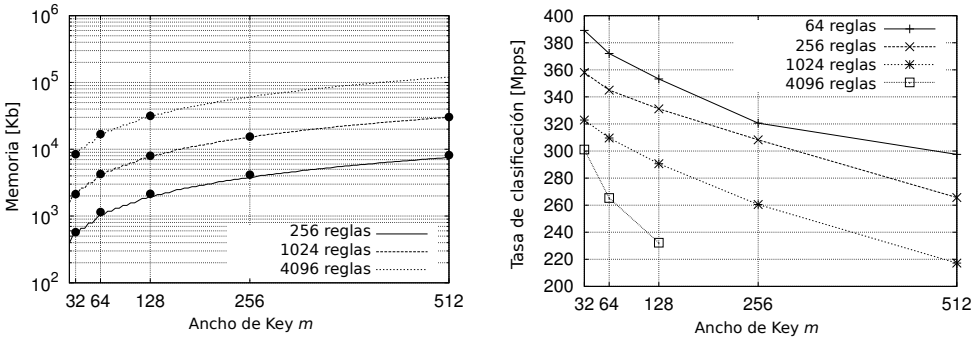


Figura 3.9: Resultados de implementación: (a) requerimientos de memoria, (b) desempeño

Cuadro 3.2: Recursos BRAM disponibles en familias de FPGAs

Familia	Tamaño BRAM (b)	# BRAMs	Total RAM (Kb)
Stratix II	4608	609	2806
Stratix III	9216	1040	9584
Stratix IV	9216	1280	11796
Stratix V	20480	2660	54476

3.4.2. Desempeño

A fin de evaluar el desempeño de arquitecturas para emulación de TCAM en FPGAs actuales, así como su evolución con la escala de integración, se realizó un grupo de ensayos en diferentes familias de FPGAs de Altera: Stratix II (65 nm, 2005), Stratix III (65 nm, 2006), Stratix IV (40 nm, 2008) y Stratix V (28 nm, 2010), seleccionando los dispositivos con mejores características de capacidad BRAM vs. speed grade como representativas de cada tecnología. Los recursos disponibles, en particular de memoria BRAM para nuestro caso, se muestran en el Cuadro 3.2.

Además del consumo de recursos, un factor determinante para evaluar estas arquitecturas de emulación de TCAM es el desempeño o velocidad de clasificación, ponderado en millones de paquetes por segundo (Mpps) o millones de lookups por segundo (Mlps). Como referencia, las redes actuales 100G pueden requerir en el peor caso un desempeño de 300 Mpps por enlace, mientras que redes 40G exigen 125 Mpps. La Fig. 3.9(b) muestra el desempeño de la implementación basada en bloques 512x40, donde se muestra que la velocidad máxima requerida de 300 Mpps se satisface en la mayor parte de los ensayos, manteniendose por encima de 200 Mpps en todos ellos.

En la Fig. 3.10, finalmente, interesa evaluar el impacto de utilizar factores de forma específicos, mas allá de su rendimiento en consumo de recursos, a fin de soportar rangos mayores que 9 bits exclusivamente mediante expansión de direccionamiento. Esta vez se fija la cantidad de reglas en $N = 256$, y se estudia el consumo de recursos y desempeño para distintos valores de w. Como se observa en la Fig. 3.10(a), los rangos de hasta 14 bits (límite de configuración de BRAM M20K) consumen por debajo de 50 Mbits de memoria. Para bloques con $w < 14$ aumenta la eficiencia de

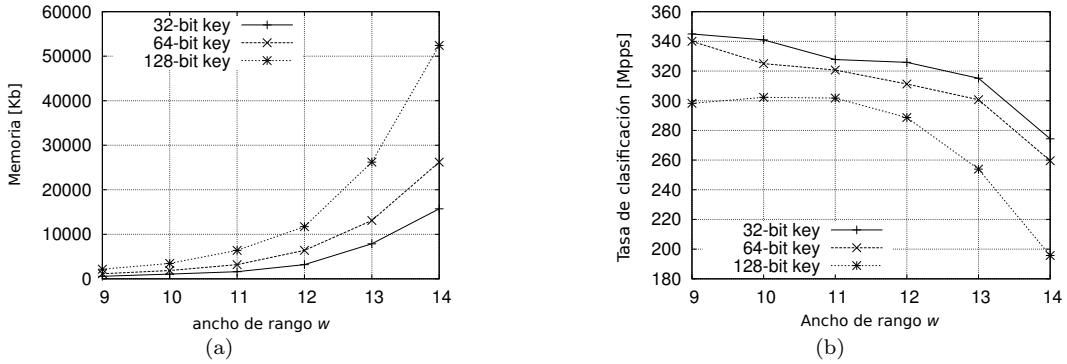


Figura 3.10: Soporte de rangos específicos (a) consumo de memoria (b) desempeño

almacenamiento, ya que i) disminuye la expansión de direccionamiento, y ii) se cuenta con mayor granularidad para ajustarse a anchos de key específicos, por lo tanto no se desperdicia tanta memoria como en el caso de w más grandes. En la Fig. 3.10(b), finalmente, se observan desempeños similares a los de la Fig. 3.9(b) para $M \leq 128$ y $w \leq 12$; sin embargo para $12 \leq w \leq 14$ tanto el desempeño como el consumo de recursos empeoran notablemente. Cabe mencionar que $w = 14$ es el límite máximo para m en bloques BRAM M20K, por lo que más allá de este ancho se debe recurrir a una combinación de espansión de direccionamiento y rangos. Esta técnica se aborda en al Cap. 5.

Finalmente, y a fin de apreciar la factibilidad creciente de estas arquitecturas con la evolución de la tecnología de FPGAs, los Cuadros 3.3, 3.4, 3.5 y 3.6 muestran los resultados post-fitting de síntesis para dispositivos Altera Stratix II, Stratix III, Stratix IV, y Stratix V respectivamente.

3.5. Conclusiones

Como conclusión del presente Capítulo, se observa que las arquitecturas de emulación de TCAM mediante memoria SRAM son convenientes y practicables en FPGAs actuales, mientras que la evolución de esta tecnología permite implementaciones cada vez más flexibles y eficaces. En particular, se observa que la emulación de TCAM es especialmente eficiente para soportar reglas basadas en rangos arbitrarios de amplitud media (hasta 512 posiciones), permitiendo implementar rangos de hasta 14 bits sin la expansión propia de las TCAMs nativas. La cantidad de reglas N y el ancho de key M produce un incremento lineal en el consumo de recursos, permitiendo implementar rulesets de hasta 4096 reglas. El desempeño de estos esquemas satisface los requerimientos de redes 100G para rulesets de hasta 1024 reglas y keys de hasta 128 bits soportando rangos de 9 bits, degradándose a partir de estos límites pero manteniéndose siempre muy por encima de los 125 MHz requeridos por redes 40G para paquetes de 40 bytes.

El trabajo detallado en este Capítulo dio origen a la tercera contribución de la Tesis Doctoral, titulada *Performance Evaluation of Packet Classification on FPGA-based TCAM Emulation Architectures*, y presentada en la Conferencia IEEE GLOBECOM 2012. Este artículo fue posteriormente

citado en otros cinco trabajos, todos ellos de autores totalmente externos al lugar de trabajo del Tesista.

Cuadro 3.3: Stratix 2 EP2SGX130GF1508C3, rango de temperatura comercial, speed grade 3 (el más rápido)

WIDTH	DEPTH											
	256				1024				4096			
	M4Ks	LUTs	REGs	Fmax (85C)	M4Ks	LUTs	REGs	Fmax	M4Ks	LUTs	REGs	Fmax
32	**40**	2054	69	**300.57**	**145**	7725	69	**248.69**	**570**	31701	72	**195.01**
64	**80**	3885	109	**271.22**	**290**	14801	111	**218.48**	N/A			
128	**152**	6795	181	**244.2**	**551**	26128	185	**175.9**	N/A			
256	**296**	12713	326	**219.44**	N/A				N/A			
256 (preserve)	**296**	13741	1292	**235.96**	N/A				N/A			
512	**592**	24994	623	**181.26**	N/A				N/A			
512 (preserve)	**592**	27005	2584	**242.07**	N/A				N/A			

Cuadro 3.4: Stratix 3 EP3SL340F1517C2, rango de temperatura comercial, speed grade 2 (el más rápido)

WIDTH	DEPTH											
	256				1024				4096			
	M9Ks	LUTs	REGs	Fmax (1100mV, 85C)	M9Ks	LUTs	REGs	Fmax	M9Ks	LUTs	REGs	Fmax
32	**32**	1794	68	**351.62**	**116**	6705	68	**335.01**	**456**	27515	71	**263.99**
64	**64**	3168	105	**352.73**	**232**	12023	109	**275.03**	**912**	51278	105	**208.9**
128	**128**	5871	176	**332.23**	**464**	22136	176	**257.4**	N/A			
128 (preserve)	**128**	6274	609	**319.69**	**464**	22586	611	**252.91**	N/A			
256	**256**	11155	320	**276.09**	**928**	42644	328	**208.42**	N/A			
256 (preserve)	**256**	12051	1216	**336.93**	**928**	43547	1225	**196.04**	N/A			
512	**512**	21878	611	**240.1**	N/A				N/A			
512 (preserve)	**512**	23606	2432	**320.92**	N/A				N/A			

Cuadro 3.5: Stratix 4 EP4SE530H35C2, rango de temperatura comercial, speed grade 2 (rápido, donde 1=el más rápido)

WIDTH	DEPTH											
	256				1024				4096			
	M9Ks	LUTs	REGs	Fmax (900mV, 85C)	M9Ks	LUTs	REGs	Fmax	M9Ks	LUTs	REGs	Fmax
32	**32**	1794	68	**362.06**	**116**	6707	68	**334.67**	**456**	27787	68	**271.89**
32 (preserve)	**32**	1885	153	**347.46**	**116**	6796	152	**305.34**	**456**	27877	153	**257.33**
64	**64**	3168	104	**344.47**	**232**	12021	108	**298.51**	**912**	51275	106	**215.8**
64 (preserve)	**64**	3372	304	**343.29**	232	12230	306	**286.12**	**912**	51511	305	**199.6**
128	**128**	5871	176	**326.37**	**464**	22139	176	**261.57**	N/A			
128 (preserve)	**128**	6272	609	**322.89**	**464**	22587	612	**239.75**	N/A			
256	**256**	11169	321	**286.2**	**928**	42633	320	**213.81**	N/A			
256 (preserve)	**256**	12051	1216	**338.41**	**928**	43533	1218	**181.72**	N/A			
512	**512**	21862	610	**235.9**	N/A				N/A			
512 (preserve)	**512**	23612	2436	**318.37**	N/A				N/A			

Cuadro 3.6: Stratix 5 SGXMB6R2F43C2, rango de temperatura comercial, speed grade 2 (rápido, donde 1=el más rápido)

WIDTH	DEPTH											
	256				1024				4096			
	M20Ks	LUTs	REGs	Fmax (850mV, 85C)	M20Ks	LUTs	REGs	Fmax	M20Ks	LUTs	REGs	Fmax
32	**28**	1806	79	**344**	**104**	6706	68	**322.89**	**412**	27788	68	**301.11**
64	**56**	3187	115	**345.07**	**208**	12032	115	**309.6**	**824**	51293	115	**265.32**
128	**105**	5508	185	**298.24**	**390**	20802	185	**290.61**	**1545**	90282	185	**232.29**
128 (preserve)	**105**	5940	630	**267.24**	**390**	21252	630	**272.85**	**1545**	90723	630	**212.09**
256	**203**	10137	325	**308.26**	**754**	38711	325	**260.48**	N/A			
256 (preserve)	**203**	11035	1218	**314.76**	**754**	39615	1218	**265.6**	N/A			
512	**399**	19614	605	**265.67**	**1482**	74592	605	**217.2**	N/A			
512 (preserve)	**399**	21403	2394	**312.89**	**1482**	76379	2394	**234.85**	N/A			

Capítulo 4

Clasificación multi-dimensional

4.1. Motivación

Los arrays asociativos, analizados en el Capítulo 3, son aplicables tanto al caso de un campo para el que fueron originalmente concebidos, como al caso de múltiples campos o dimensiones. Intuitivamente, un clasificador de múltiples campos podría formarse simplemente extendiendo en ancho una arquitectura de búsqueda lineal como son las TCAMs. Con referencia al esquema general de la Fig. 2.14(b), vemos que el encabezado se compara linealmente con cada una de las reglas *rule_1,..., rule_n*; esta comparación lineal se puede realizar en forma secuencial (por ejemplo accediendo a una memoria RAM) o concurente (mediante TCAM). El almacenamiento requerido por este esquema es igual al requerido para almacenar la tabla de reglas original $O(M \cdot N)$; sin embargo el uso de RAM requiere $O(N)$ tiempo de búsqueda mientras que la TCAM consume elevada potencia y no escala para encabezados anchos o reglas basadas en rangos arbitrarios. De estas observaciones, surge la necesidad de abandonar el esquema original de búsqueda asociativa para clasificación multi-dimensional; identificando y explotando propiedades del proceso de clasificación y los rulesets reales para mejorar su escalabilidad.

En este capítulo se considerarán los esquemas existentes para clasificación en *múltiples campos* o dimensiones, haciendo un especial análisis de aquéllos esquemas basados en *descomposición* del espacio de búsqueda. En este contexto, llamaremos *lookup* a las operaciones de clasificación sobre cada campo individual, mientras que la combinación de los resultados obtenidos en cada campo se realiza mediante una posterior etapa de *agregación*. En particular, se observa que actualmente no existe una base teórica sobre la cual comparar objetivamente los esquemas existentes, por lo que se propone una nueva taxonomía que facilita esta tarea y permite encarar el análisis desde un enfoque común. De las observaciones realizadas, se propone un nuevo esquema aún no explorado previamente. Para el caso particular de plataformas FPGA, este esquema permite explotar más eficientemente los recursos tecnológicos ofrecidos con muy buen desempeño.

4.2. Técnicas de clasificación multi-dimensional

Según las características de complejidad, los problemas de clasificación pueden ser *uni-dimensionales (1D)*, *bi-dimensionales (2D)*, o *multi-dimensionales en k campos (kD)*. Además, las técnicas aplicadas pueden ser *algorítmicas* o *no-algorítmicas*. La clasificación 1D es esencialmente una operación de *lookup*, y es ampliamente aplicada para enrutamiento por máxima longitud de prefijos de redes IP. Este caso puede resolverse utilizando métodos algorítmicos como tries, búsqueda binaria por longitud de prefijo o búsqueda binaria en rangos; o métodos no-algorítmicos tales como búsqueda lineal o su contraparte recurrente ampliamente difundida, *memorias accesibles por contenido (CAMs)* [87] [88]. El caso 2D, en tanto, es particularmente relevante por su aplicación para lookup de duplas IP Source/ IP Dst en redes privadas virtuales (VPNs), en aplicaciones de supervisión (monitoring) y en esquemas de multicast. Este caso se puede resolver mediante la técnica Grid-of-Tries (GoT) [87]. Para el caso kD, en tanto, las técnicas mencionadas fallan ya que sus requerimientos de memoria o tiempo no escalan adecuadamente. Para controlar esta explosión de requerimientos, se deben explotar las propiedades de las tablas de reglas reales. Las principales técnicas para realizar esto se basan en el principio *divide y vencerás* aplicado a distintos aspectos del ruleset, ellas son:

1. *Particionado del espacio de búsqueda en sub-espacios no-ortogonales.* Este método, más formalmente del tipo *disminuye y vencerás* [89], consiste en la descomposición del key en base a los *hiper-espacios* formados por el ruleset en k dimensiones. Como principio fundamental, esta técnica considera el espacio k-dimensional y lo reduce gradualmente en múltiples etapas hasta alcanzar una complejidad manejable con técnicas de búsqueda lineal. Esta característica es naturalmente adecuada para implementación mediante *árboles de decisión* [90] [91] [92] [93] [94]. Para optimizar tales reducciones se aplican diferentes *heurísticas* que determinan en última instancia el rendimiento de estas arquitecturas. Para rulesets con patrones favorables o bajo solapamiento, esta técnica puede ser eficiente; sin embargo su rendimiento disminuye a medida que el ruleset presenta patrones de solapamiento de mayor complejidad. En particular, la profundidad de los árboles es fuertemente no-determinística [92]. Estas técnicas se refieren comunmente como de *clasificación basada en árboles de decisión.*

2. *Particionado del espacio de búsqueda en sub-espacios ortogonales.* Estas técnicas, basadas en las observaciones de [96] y [97], dividen el key en sus k campos ortogonales con complejidad de lookup reducida. A diferencia de la técnica anteriores, en éstas se aplica lookup 1D local a cada campo, y posteriormente se agregan los resultados obtenidos. Por ello, se las conoce también como técnicas de *clasificación por descomposición.*

Esta división de técnicas es claramente interpretada en [98]. Según se analiza en ese trabajo, las técnicas (1) trabajan *cortando* el ruleset kD original en otros más pequeños, *también k-dimensionales*, formando una estuctura de tipo árbol. Las técnicas (2), en cambio, *integran* resultados de lookup independientes en una o múltiples etapas, formando efectivamente una estructura de árbol invertido.

Las técnicas basadas en árboles obtienen su rendimiento del criterio particular de seccionamiento del espacio multi-dimensional a lo largo del árbol. HiCuts [90], la primera técnica de este tipo, realiza

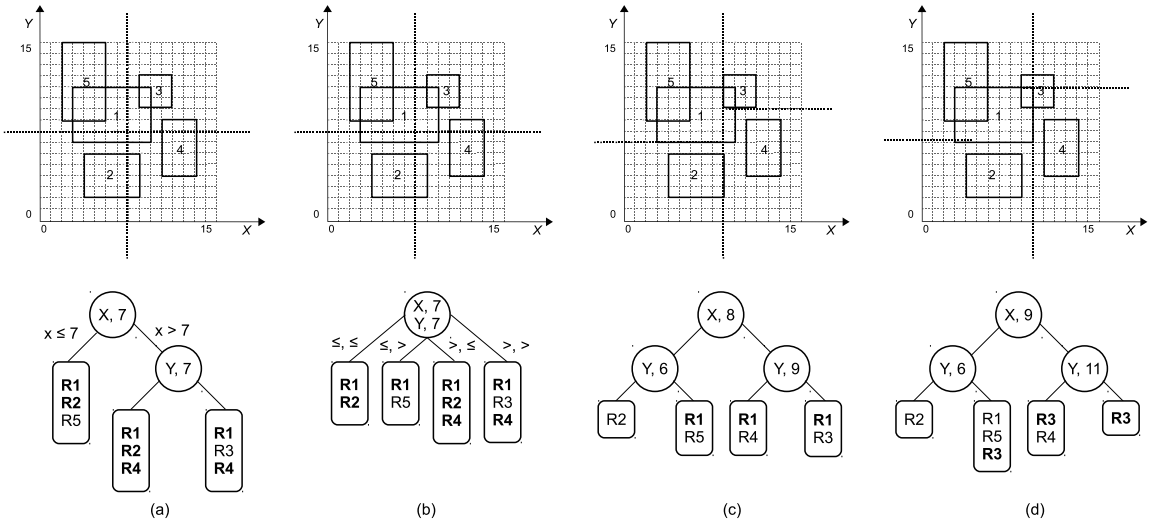


Figura 4.1: Replicación de reglas: (a) HiCuts, (b) HyperCuts, (c)(d) dos versiones de HyperSplit

múltiples cortes equi-espaciados en una sola dimensión a la vez, lo cual puede requerir abundante duplicación de reglas y árboles muy profundos. HyperCuts [91] mejora esto permitiendo múltiples cortes equi-espaciados en múltiples dimensiones a la vez; reduciendo de este modo la profundidad del árbol. Sin embargo el problema común de ambas técnicas es la alta replicación de reglas, ya que no son capaces de adaptar sus puntos de corte a los patrones de solapamiento del ruleset. HyperSplit [92] ataca este aspecto permitiendo realizar cortes no-equiespaciados en una dimensión a la vez; es decir, los puntos de corte se optimizan de acuerdo al mapa de reglas particular para reducir la replicación de reglas. A fin de ilustrar estos conceptos, se considera en la Fig. 4.1(a) un ruleset simple similar al analizado en [94], donde se aplica un árbol HiCuts de dos etapas, con un corte en los ejes X e Y respectivamente en base al valor del bit más significativo. El límite de complejidad para aplicar búsqueda lineal, es decir para determinar que se ha llegado a una hoja del árbol, se fija en este caso en 3 reglas. Como resultado se obtienen tres hojas conteniendo tres reglas cada una. En la Fig. 4.1(b), en tanto, se muestra el mismo ruleset procesado mediante HyperCuts, donde se realizan ambos cortes simultáneamente resultando en un único nivel de decisión. En ambos casos se nota una fuerte replicación de las reglas $R1, R2$ y $R4$, destacadas en negrita. En la Fig. 4.1(c) se aplica HyperSplit, el cual que permite reducir la replicación de reglas a costa de comparaciones más específicas en cada nodo; sin embargo se nota que $R1$ se sigue replicando en tres de las cuatro hojas existentes. En un intento por aislar $R1$ se ilustra en la Fig. 4.1(d) una segunda opción de HyperSplit; sin embargo en este caso $R3$ se replica en tres de las cuatro hojas. Como se observa, debido a los solapamientos existentes en el ruleset, estas técnicas sufren en última instancia de este tipo de problema, lo que puede conducir a excesivos consumos de memoria. De este modo, la eficiencia de los esquemas basados en árboles de decisión resulta muy variable según el ruleset a almacenar. Aún con rulesets favorables, las sucesivas actualizaciones incrementales pueden degradar significativamente el desempeño del esquema.

La técnicas de descomposición, por su parte, son naturalmente más adecuadas para implementación mediante arquitecturas concurrentes, por lo general en hardware especializado. Además, a

diferencia de las técnicas (1), conservan mejor sus propiedades al variar las características del ruleset. Esto es debido a que consisten de dos etapas bien diferenciadas, i) *lookup 1D*, completamente local a cada campo; y ii) *agregación*, que combina los resultados de lookup. La primera se implementa a través de técnicas convenientes a nivel uni-dimensional para cada campo, mientras que la segunda se puede implementar mediante direccionamiento directo de memoria, lógica combinacional o técnicas de hashing. El costo de la estabilidad con el ruleset es escalabilidad limitada, especialmente en la etapa de agregación; ya que tienen capacidad limitada para explotar el patrón de reglas.

Las técnicas basadas en árboles, naturalmente apropiadas para implementación por software, requieren extensivo trabajo de adaptación para implementarlas mediante arquitecturas concurrentes y su desempeño es dependiente del patrón de reglas; sin embargo tienen una visión global del ruleset en k dimensiones y pueden por tanto aplicar optimizaciones que las técnicas por descomposición no alcanzan a identificar. Como consecuencia, las técnicas de particionado no-ortogonal pueden escalar mejor con el tamaño del ruleset mediante ests optimizaciones, pero esta escalabilidad es a la vez altamente dependiente de las características del ruleset. En particular, las reglas que cubren grandes espacios de búsqueda o que incluyen *wildcards* en una o múltiples dimensiones generan el efecto de *duplicación de reglas*, que es el mayor limitante de las técnicas no-ortogonales.

La presente Tesis se basa en dos aspectos de los esquemas de clasificación: el *análisis de aplicación en redes* y el *diseño de arquitecturas de procesamiento*. El primer aspecto se introduce claramente en [98] y [99], mientras que sus propiedades generales son extraídas y analizadas en [97]. Este aspecto será abordado en la Sec. 4.3, mientras que el segundo se trata en la Sec. 4.4. Se definirá el trabajo en base a la *interpretación geométrica* del ruleset, adoptando la técnica de descomposición por su desempeño consistente al variar el ruleset y por ser especialmente apropiada para implementación en FPGAs. Los trabajos existentes en clasificación, y en particular en clasificación por descomposición, son bastante heterogéneos no sólo en cuanto a la metodología sino en cuanto a plataformas de implementación, por lo que su comparación directa es muy difícil. Por ello, se comenzó relevando estos trabajos y buscando conceptos fundamentales subyacentes. Sobre esta base, se pudo arribar a una comparación objetiva e independiente de las tecnologías adoptadas, sobre la cual fundamentar nuestra propuesta.

4.3. Análisis de aplicación en redes

4.3.1. Introducción

Lás técnicas de clasificación por descomposición básicamente permiten separar etapas de lookup y de agregación multi-dimensional, permitiendo optimizarlas individualmente. Sin embargo, debido a los posibles solapamientos entre reglas de clasificación, estas técnicas enfrentan una serie de desafíos que llevan a compromisos entre consumo de recursos y desempeño. En esta sección se realizará un análisis general a fin de comprender cualitativamente estos compromisos.

A fin de introducir nuestra discusión, consideremos un simple ruleset formado por las reglas

uni-dimensionales 1 y 2 definidas en base a rangos de valores de un campo F_1, como se muestra en la Fig. 4.2. En las Figs. 4.2(a), 4.2(b) y 4.2(c) se consideran tres casos de solapamiento posibles utilizando prefijos; ellos se caracterizan por presentar solapamientos *totales* o *nulos*. La Fig. 4.2(d), en tanto, muestra un caso de solapamiento *parcial*, sólo posible al utilizar rangos arbitrarios para la definición de reglas. La primera fila de figuras muestra cómo se resolverían estos casos aplicando lookup en base a la *mejor coincidencia (Best Match, BM)*, que es la técnica aplicada al lookup por prefijos. Si bien esta técnica es apropiada para buscar la mejor ruta en el caso de lookup por prefijos, observamos que no se puede diferenciar el caso en que se cumple la regla 1 de aquél donde se cumplen ambas reglas $1 + 2$. En el caso particular de solapamientos parciales, las prioridades no se pueden definir por mayor longitud de prefijo *(Longest-Prefix Match, LPM)* como se hace en enrutamiento IP, sino que se debe fijar un orden arbitrario de reglas. En la segunda fila, en tanto, se observan los mismos casos implementados mediante lookup basado en *múltiples coincidencias (multi match, MM)*, donde se reportan todas las reglas involucradas. Como se observa, mediante esta técnica se eliminan las ambigüedades de los resultados BM. Los casos MM en este simple ejemplo son $1, 2$ y $1 + 2$; a cada uno de ellos se les pueden asignar identificadores o etiquetas $1, 2$ y 3 respectivamente. Vemos que BM sólo sería capaz de devolver las etiquetas 1 y 2, mientras que MM abarca todos los casos posibles $1, 2$ y 3. El costo por esta mayor sensibilidad es una mayor complejidad de lookup como se verá más adelante.

Consideremos ahora el simple ruleset 2D de la Fig. 4.3(a), formado por dos reglas 1 y 2 definidas en base a la combinación de dos campos F_A y F_B con dos reglas cada uno. Se puede observar que, en este caso, la agregación de resultados BM 1D $1-1$ y $2-2$ no se puede relacionar fehacientemente con reglas 2D. En otras palabas, no existen reglas 2D formadas por los resultados BM 1D $1 - 1$ y $2 - 2$ respectivamente. Estos casos se resuelven mediante la generación de resultados llamados *pseudo-reglas*: la agregación $2 - 2$ no presenta incertidumbre, por lo que se genera la pseudo-regla 2D 0 (no-match). Para la agregación $1 - 1$ existe un conflicto entre las reglas 2D 1 y 2. Si bien las reglas 1D se pueden resolver por solaparse completamente, éste no es el caso cuando efectuamos su combinación en 2D. Se debe en este caso recurrir a prioridades arbitrarias; por ejemplo si asignamos prioridad a la regla 2D 1 sobre la regla 2D 2, se devolvería como resultado de la agregación $1 - 1$ la regla 1. En el caso de la Fig. 4.3(b), donde se registran solapamientos parciales, los casos de incertidumbre son aún más críticos. El caso $1 - 1$ se puede resolver por prioridad arbitraria, asignando como resultado la regla 2D 1. Para los casos de agregación $1 - 2$ y $2 - 1$, sin embargo, no existe un único resultado de agregación como muestran las líneas sombreadas en la tabla de reglas correspondiente. Para el caso $1 - 2$, por ejemplo, el resultado podría ser 0 o 2 según la zona del campo F_A donde se ubique el key.

Estos ejemplos demuestran que el lookup BM basado en longitudes de prefijo, muy difundido y eficiente para la determinación de rutas IP, no es efectivo para realizar clasificación multi-dimensional por agregación, en particular para el caso de reglas definidas sobre rangos de valores arbitrarios del key. Es así que la principal motivación de este capítulo es la de estudiar arquitecturas de lookup y agregación *multi-match*, aplicables a clasificación por descomposición sobre *rangos arbitrarios* de valores del encabezado.

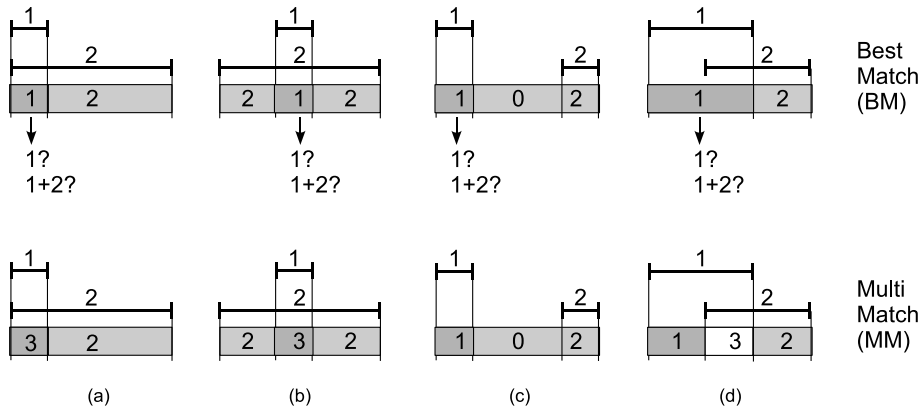
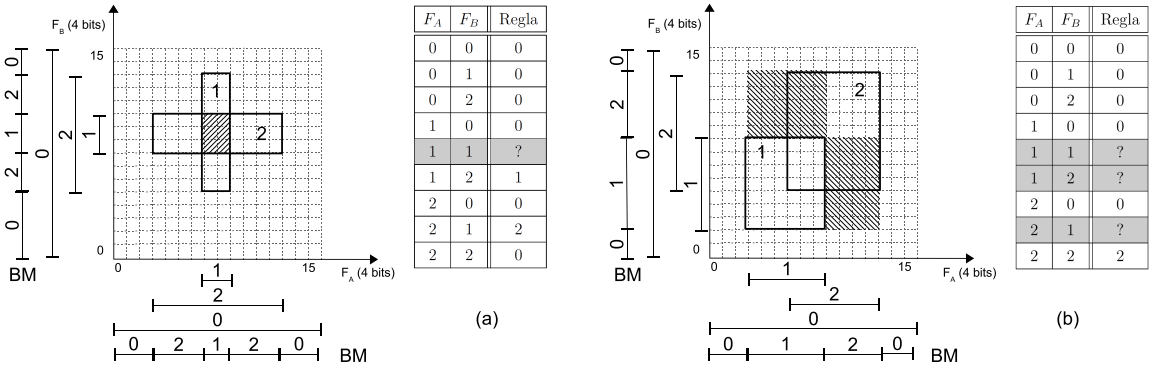


Figura 4.2: Best-Match (BM) vs. Multi-Match (MM) Lookup en un campo



F_A	F_B	Regla
0	0	0
0	1	0
0	2	0
1	0	0
1	1	?
1	2	1
2	0	0
2	1	2
2	2	0

F_A	F_B	Regla
0	0	0
0	1	0
0	2	0
1	0	0
1	1	?
1	2	?
2	0	0
2	1	?
2	2	2

Figura 4.3: Best-Match (BM) vs. Multi-Match (MM) Lookup en dos campos

4.3.2. Una nueva taxonomía de los esquemas de agregación

Los trabajos existentes sobre clasificación son muy numerosos y heterogéneos, tanto en métodos aplicados como en plataformas de implementación. Entre los estudios al respecto se pueden citar [62], [100] y [101]. En esta sección se presenta una nueva taxonomía propia y general de los esquemas de agregación para clasificación. A diferencia de los estudios mencionados que analizan detalladamente gran cantidad de trabajos, el fin principal de nuestro estudio es identificar opciones no exploradas hasta el momento y analizar su conveniencia.

Como contexto de esta discusión, se presenta en la Fig. 4.4 un ruleset general en k dimensiones donde $k = 2$. La Fig. 4.4(a) muestra la especificación de reglas de dos campos A y B, en base a rangos definidos sobre sus valores respectivos F_A y F_B. Por ejemplo, la regla 1 se define para intervalos $F_A = [2, 4]$ y $F_B = [4, 6]$ sobre los campos A y B respectivamente. En las Figs. 4.4(b) y 4.4(c), en tanto, se muestra la *interpretación geométrica* de tal conjunto de reglas. En general, se pueden identificar conjuntos de *valores (o intervalos) propios o únicos (Unique Values, UVs)* para cada dimensión. En la Fig. 4.4(b), por ejemplo, los valores únicos $UV_A = \{0...|UV_A|\}$ y $UV_B = \{0...|UV_B|\}$ en el espacio uni-dimensional se representan mediante segmentos de línea, y sus resultados de agregación $UV_{AB} = \{0...|UV_{AB}|\}$ en dos dimensiones se representan como rectángulos. En este caso las reglas

Rule (UV_C)	F_A	F_B	UV_A	UV_B	UR_A	UR_B
1	2:4	4:6	1	2	1,2	3,4
2	3:9	1:5	2	1	2,3,4,5	1,2,3
3	9:11	4:6	3	2	5,6	3,4
4	8:12	3:7	4	3	4,5,6,7	2,3,4,5
5	3:9	11:12	2	5	2,3,4,5	6
6	9:11	11:15	3	4	5,6	6,7

(a)

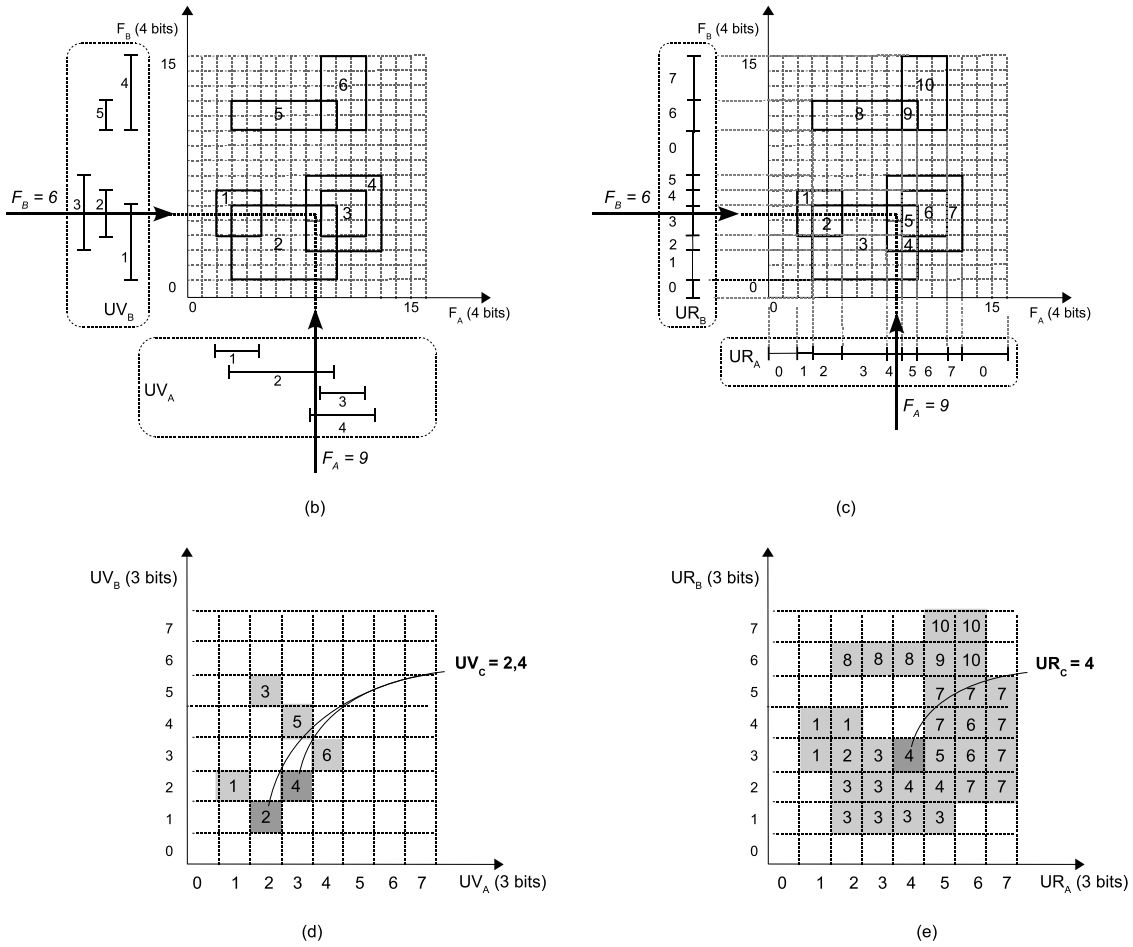


Figura 4.4: Ruleset 2D general: (a) especificación de reglas, (b)(c) interpretación geométrica de UVs y URs, (d) agregación de UVs, (e) agregación de URs

especificadas son bi-dimensionales, por lo que $UV^i_{AB} = R_i \forall i$. En el caso de $k > 2$, esto se extiende al espacio k-dimensional donde se definen $UV_{AB...k} = \{0...|UV_{AB...k}|\}$ valores únicos, representados por hiper-rectángulos, donde $UV^i_{AB...k} = R_i \forall i$. El concepto de *valor único*, utilizado para representar un intervalo de valores del encabezado, enfatiza el hecho de que estos intervalos *no se repiten* en su dimensión, si bien pueden hacerlo en una dimensión superior para el conjunto de reglas considerado. En referencia a la Fig. 4.4(b), por ejemplo, las reglas 2D 2 y 5 son valores únicos en el espacio

bi-dimensional, si bien ellas comparten el mismo $UV_A = 2$ definido por $F_A = [3, 9]$. A la inversa, un UV en cierta dimensión puede estar involucrado en más de un UV en dimensiones superiores; por ejemplo $UV_A = 2$ forma parte tanto de $UV_{AB} = 2$ como de $UV_{AB} = 5$, mientras que $UV_B = 2$ forma $UV_{AB} = 1$ y $UV_{AB} = 3$. Esta característica es la razón por la que $|UV|$ se incrementa a medida que se pasa a espacios de mayor orden, llegando eventualmente al espacio de reglas $|UV_{AB...k}| = N$. Esta es también la razón por la cual la complejidad de búsqueda 1D (lookup) en esquemas de clasificación por agregación es mucho menor que aquélla de los árboles multi-dimensionales, los cuales dividen incrementalmente el espacio kD. En la Fig. 4.4(b), $|UV_A| = 4$, $|UV_B| = 5$ y $|UV_{AB}| = 6 = N$. Así, $UV_{AB} = 1$ (R1) resulta de combinar $UV_A = 1$ y $UV_B = 2$, $UV_{AB} = 2$ (R2) involucra la dupla $UV_A = 2, UV_B = 1$, y así sucesivamente.

Como se observa, los UVs pueden solaparse entre sí, esto causa *conflictos entre UVs* como se comentó en la Sec. 4.3. Una forma de resolver dichos conflictos consiste en asociar un *peso* a cada UV generando *órdenes de precedencia* entre reglas; sin embargo como se ha discutido esto genera casos de incertidumbre para UVs definidos en base a rangos arbitrarios de valores. Para poder discriminar todos los casos de solapamiento, se deben en cambio discriminar las *combinaciones únicas de UVs*, definiendo las llamadas *regiones propias o únicas (Unique Regions, URs)*. En la Fig. 4.4(c), $|UR_A| = 8$, $|UR_B| = 8$ y $|UR_{AB}| = 11$. En general, tanto $|UV|$ como $|UR|$ crecen al agregar múltiples dimensiones, sin embargo lo hacen a una tasa mucho menor que la dictada por su producto cartesiano. Es decir, de los $|UV_A|.|UV_B| = 4 \cdot 5 = 20$ posibles productos sólo $|UV_{AB}| = 6$ son válidos para ruleset considerado, mientras que sólo $|UR_{AB}| = 33$ combinaciones de URs son válidas de entre las $|UR_A| \cdot |UR_B| = 8 \cdot 8 = 64$ posibles combinaciones de URs 1D. En la Fig.4.4(d) se observa el mapa de agregación de UVs, mientras que la Fig. 4.4(e) ilustra el mapa de agregación de URs. Es de destacar que la asignación de etiquetas es en este caso arbitraria ya que cada etiqueta representa una combinación única de UVs, sin depender de pesos asignados a tales UVs. Esto permite, por ejemplo, mantener una pila de etiquetas disponibles, agregando elementos a la pila durante la eliminación de reglas y asignándolos nuevamente durante la adición de nuevas reglas. En general $|UR| > |UV|$ (esto se analizará con mayor detalle); para el caso de BM se llega al extremo donde $|UR| = |UV|$, es decir, cada UR representa el UV de mayor prioridad en su intervalo.

En las Figs. 4.4(b) y 4.4(c) se ilustra asimismo un caso de clasificación para $F_A = 9$ y $F_B = 6$, resultando en el punto de coordenadas $\{9; 6\}$. Como se observa, este punto se encuentra dentro de múltiples rectángulos que representan las múltiples reglas coincidentes. Extendiendo este concepto para k campos ortogonales, decimos que el encabezado de un paquete representa un punto en el espacio k-dimensional, el que debe ser evaluado contra $|UV_{AB...k}| = N$ hiper-rectángulos definidos por las N reglas; para determinar en cuáles de ellos está contenido. Este escenario, es un problema clásico llamado *localización de un punto en el espacio multi-dimensional* en geometría computacional, y tratado extensivamente en [102]. Según ese trabajo, el caso general de localización de un punto para $k > 3$ dimensiones y n objetos *sin solapamiento* presenta complejidades bien $O(log_2\ (k-1))$ en tiempo y $O(log_2\ n)$ en espacio; bien $O(log_2\ n)$ en tiempo y $O(n^k)$ en espacio, resultando en un compromiso de complejidades espacio-tiempo. Sin embargo, en el caso general de clasificación, donde tales objetos *presentan solapamiento*, estos límites representan un caso particular de referencia en un contexto mucho más general [62].

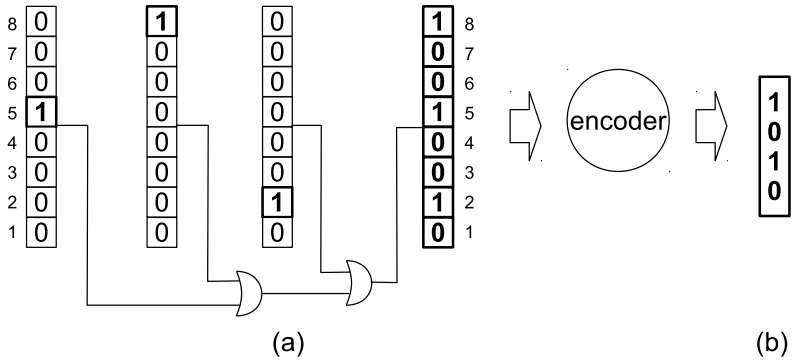


Figura 4.5: Formatos de metadatos en clasificación: (a) mapas de bits (bitmaps), (b) etiquetas (labels)

Del estudio de los trabajos relacionados, en particular sobre clasificación por descomposición, se determinó la línea de trabajo a seguir. Para una arquitectura de clasificación por descomposición, diferenciaremos claramente dos etapas relativamente independientes en su diseño: *búsqueda(lookup)* y *agregación*. En este capítulo nos concentraremos en la etapa de agregación, mientras que la etapa de lookup se trata en el Cap. 5. A continuación, nos concentraremos en el análisis de la *interfaz* entre ambas etapas, la cual determina en gran medida su comportamiento. Esta interfaz está determinada por un *metadato* que propaga resultados de las etapas de lookup hacia la etapa de agregación, o entre los diferentes nodos de procesamiento en una etapa de agregación multi-nivel.

En este punto, se definen dos tipos de metadato: los basados en *etiquetas (labels)*, y los basados en *mapas de bits (bitmaps)*. Ambos consisten en arrays de bits, sin embargo la información transportada es esencialmente diferente. Un bitmap es el resultado de una operación OR entre múltiples vectores del tipo *one-hot* como muestra la Fig. 4.5(a). En este tipo de vector, la información es transportada por el *estado* de cada bit, asociado a su *posición* dentro del array. En un bitmap, cada bit es *independiente* de los demás ya que no necesita de ellos para entregar su propio resultado. En una etiqueta, en cambio, cada bit forma parte de un conjunto y adquiere significado sólo en relación a los estados de los demás bits de este conjunto. En relación a un bitmap, se puede definir una etiqueta como un resultado *codificado* obtenido a partir de él, como se muestra en la Fig. 4.5(b). Esta codificación se puede realizar durante el proceso de clasificación, por ejemplo mediante una función hash; o mediante pre-procesamiento, asignando etiquetas a bitmaps. La ventaja de los bitmaps frente a las etiquetas es que no requieren pre-procesamiento, mientras que las etiquetas requieren menor ancho de palabra y son más escalables a medida que aumenta la cantidad de reglas.

El esquema de clasificación por descomposición generalmente aprovecha el hecho de que la complejidad de lookup suele ser bastante menor a la complejidad de clasificación; por lo que implementa lookup y agregación por separado. Desde el punto de vista de la arquitectura de agregación, que es lo nos ocupa en este capítulo, podemos dividir los trabajos sobre descomposición como basados en regiones únicas o en valores únicos según el metadato que se propaga. A su vez, se pueden utilizar *labels* o *bitmaps* como interfaces de agregación. Por el momento no se considera el método interno de búsqueda en las etapas de lookup, que será abordado en el Cap. 5 según una taxonomía similar a ésta.

Al analizar los esquemas basados en UVs y en URs, se destacan dos trabajos seminales basados en bitmaps y en labels respectivamente que, si bien siguen el esquema de descomposición, no explotan totalmente sus características. Posteriormente, y partiendo de estos casos extremos, compararemos los restantes trabajos analizados y definiremos nuestra propuesta.

El concepto clave en los esquemas basados en URs es la identificación de regiones definidas por *solapamientos únicos de reglas* en espacios multi-dimensionales o sus proyecciones en espacios uni-dimensionales; la cantidad de regiones es $|UR| \leq 2N-1$ en el peor caso para cada dimensión (campo), sumando $\sum_{i=1}^{N} i = [N \cdot (N+1)]/2$ en dos dimensiones [99] [103]. En los esquemas basados en UVs, en tanto, se identifican las reglas individuales o sus proyecciones en cada dimensión. Los UVs son $|UV| \leq N$ tanto para las dimensiones individuales $d(1 \leq d \leq K)$ como para cualquier espacio generado por sus combinaciones. En este sentido, se observa que los UVs son más escalables que las URs, sin embargo es importante destacar que las URs con *mutuamente excluyentes*, mientras que los UVs no lo son por presentar solapamientos. Los esquemas basados en UVs deben en consecuencia generar *pseudo-reglas* para resolver tales solapamientos, las cuales crecen exponencialmente al ascender en la jerarquía de espacios [104].

Comenzaremos nuestro análisis por el caso mencionado al principio del capítulo, donde la clasificación se reduce a una simple extensión del key de una TCAM; este es un caso extremo donde el esquema de agregación se confunde con el de lookup. Es decir, la arquitectura de clasificación es prácticamente una TCAM con mayor cantidad de palabras y mayor ancho que los requeridos para lookup. Definimos en este caso las interfaces de agregación como *aquéllos puntos que limitan el intervalo (scope) máximo de rangos arbitrarios*. Por ejemplo, como se ve en el Cap. 3, las memorias TCAM pueden requerir expansión de hasta w reglas por cada regla de rango arbitrario de ancho w. Al atravesar una interfaz de agregación, por ejemplo desde un campo de rangos arbitrarios a un campo definido en prefijos, esta expansión deja de ser necesaria. Para más detalles sobre esto referirse al Cap. 5.

El esquema Lucent Bit Vector (BV), propuesto en [105], se puede considerar el trabajo seminal de agregación basada en URs. Este esquema, mostrado conceptualmente en la Fig. 4.6(a), se caracteriza por utilizar una interfaz bitmap de ancho N, es decir, se consideran bitmaps en k-dimensiones aún en las etapas de lookup. De este modo, la etapa de agregación se reduce sólo a una operación AND entre resultados de lookup. Este esquema comparte la misma interfaz de agregación que el caso TCAM, pero se aparta de aquél en que separa las implementaciones de lookup y agregación utilizando una técnica algorítmica para lookup. Decimos que este esquema es basado en URs ya que para cada UR existe un BV único; si bien éste es un formato muy ineficiente en cuanto a almacenamiento ya que $N >> log_2|UR|$. La implementación de las etapas de lookup puede tener distintas formas y se aborda en el Cap. 5. En el caso particular de [105] se resuelve mediante búsqueda binaria en $|UR| = O(N)$ regiones, conduciendo a complejidad de memoria $O(k \cdot N \cdot N)$. En la Fig. 4.7(b) se muestra un ejemplo de cómo trabaja este esquema para el caso de dos campos *Address* y *Port*, especificados en la Fig. 4.7(a). Se observa que el bitmap para cada campo, llamado *Bit Vector* en este trabajo, reserva un bit para cada regla 2D *a...k* a fin de obtener su agregación mediante simples operaciones AND.

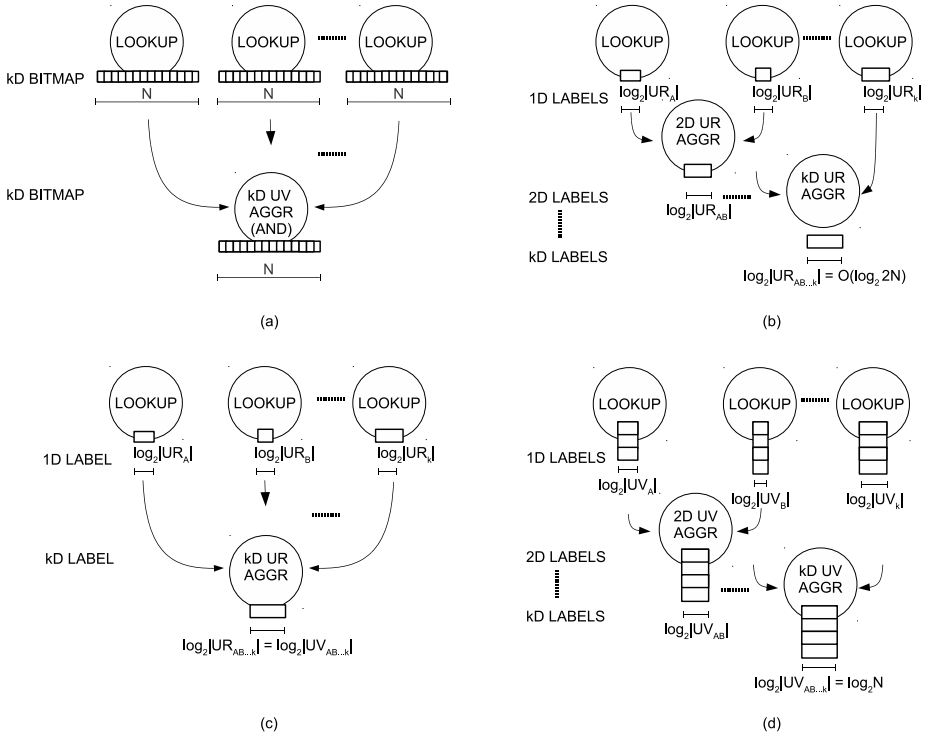


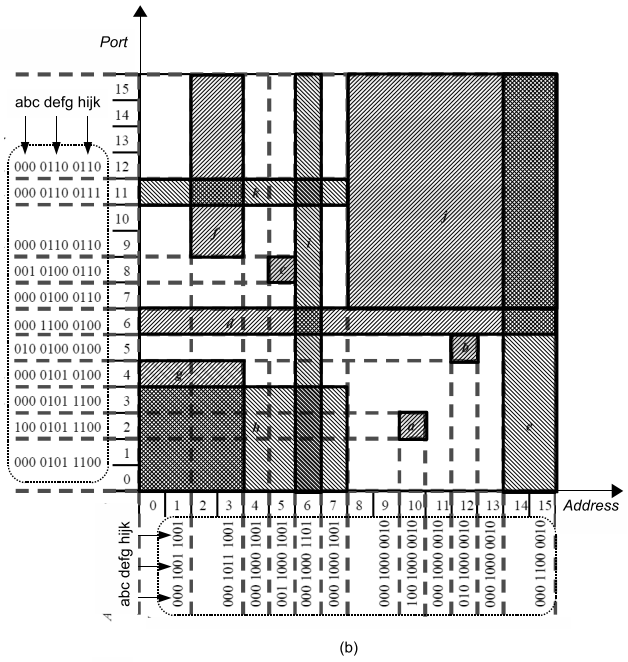
Figura 4.6: Metadatos de agregación: (a) BV, (b) RFC, (c) XPROD, (d) DCFL

En *Clasificación Recursiva de Flujos (Recursive Flow Classification, RFC)* [106], se explota el hecho de que sólo algunos de los 2^N posibles vectores de ancho N corresponden a URs en rulesets reales, es decir $|UR| << 2^N$. Sobre esta base, RFC propone utilizar etiquetas representando los llamados *identificadores de clase equivalente (Equivalence Class Identifiers, eqID)* en lugar de BVs; estos eqIDs esencialmente extraen sólo los BVs que corresponden a URs presentes en el ruleset y los almacenan en forma optimizada con ancho $log_2|UR| << N$. Los eqIDs se relacionan así biunívocamente con BVs en kD, estos BVs se utilizan sólo durante la actualización del ruleset y son llamados *Bitmaps de clase (Class Bitmap, cbm)* en ese trabajo. El costo de utilizar estas etiquetas de ancho óptimo es que cada una de ellas concentra información de múltiples reglas, esta información debe ser pre-procesada durante la construcción y actualización del ruleset conduciendo a mayor necesidad de pre-cómputo. RFC implementa las etapas de agregación mediante direccionamiento directo de memoria; por ello sufre de explosión de memoria para rulesets desfavorables. Este esquema se ilustra conceptualmente en la Fig.4.6(b). La propuesta *Mapeo Jerárquico de Espacios (Hierachical Space Mapping, HSM)* [107] sigue la filosofía de RFC, mejorando su consumo de memoria, mientras que el trabajo *Clasificación Mejorada Recursiva de Flujos (Enhanced Recursive Flow Classification, ERFC)* [108] aplica diversas técnicas de compresión para reducir dicho consumo; volveremos sobre ella en el Cap. 5.

El trabajo referido como Crossproducting (XPROD) [96] se puede considerar como un caso especial que, si bien es basado en agregación por URs, sirve de nexo con los esquemas basados en UVs. XPROD realiza agregación basada en labels *en una etapa*. Para ello identifica URs en cada dimensión, sin embargo en este caso se consideran sólo reglas basadas en prefijos, es decir que los solapamientos

Regla	Address	Port
a	1010	2:2
b	1100	5:5
c	0101	8:8
d	*	6:6
e	111*	0:15
f	001*	9:15
g	00*	0:4
h	0*	0:3
i	0110	0:15
j	1*	7:15
k	0*	11:11

(a)



(b)

Figura 4.7: Lucent Bit Vector (BV): (a) ruleset de ejemplo, (b) principio de funcionamiento

son totales o nulos, no parciales. Esta condición, como se discute en la Sec. 4.3, lleva al caso especial en que $UR_i = UV_i \forall i$ y $|UV| = |UR|$ $\forall d$ $(d = 1...k)$; de este modo podemos interpretar XPROD como un esquema basado en UVs o URs indistintamente. Sin embargo, este esquema es útil sólo para este caso ya que si consideramos rangos arbitrarios con solapamiento parcial i) la agregación por URs en una etapa causaría explosión de memoria, y ii) la agregación basada en UVs en una etapa no sería factible como se demostró en la Fig. 4.3. Aún así, este esquema requiere una tabla de agregación excesivamente grande ya que se debe pasar directamente de los k espacios 1D al espacio kD. Esto se ilustra en la Fig. 4.6(c). La Fig. 4.8(a) ilustra un ejemplo de agregación mediante XPROD; se observa en este caso la división del ruleset en grupos de intervalos o valores únicos para cada dimensión. El problema de agregación consiste en almacenar un resultado para cada combinación posible de valores únicos en k dimensiones como se muestra, llegando a requerir espacio $O(|UV_A|.|UV_B|...|UV_k|) >> |UV_{AB...k}|$. Si bien este requerimiento se mitiga mediante hashing, permanece como un problema intrínseco del esquema. En la Fig. 4.8(b) se muestra la interpretación geométrica del ruleset 3D de la Fig. 4.8(a); para mayor claridad y ya que *Prot* consiste en sólo dos valores exactos *TCP* y *UDP* se muestran dos mapas separados que luego co-existen en el ruleset. En la Fig.4.8(c) se observan todos los productos cruzados para el ruleset. En esta figura se puede apreciar el problema de este esquema, que consiste en la necesidad de múltiples *pseudo-reglas*. Las pseudo-reglas son reglas artificiales, no existentes en el ruleset original, cuyo único fin es generar resultados de agregación. Por ejemplo, para $F_c = 1$ observamos que la regla a se entrega como resultado de los productos cruzados $1-1$ y $1-2$, aun cuando la regla original a corresponde sólo al producto $1-1$. El producto $1-2$ *no existe* en el ruleset original, pero debe devolver la regla a según se observa en el mapa. Lo mismo sucede con el caso de la regla b. En general, este efecto ocurre para la combinación de UVs más específicos (de menor scope o mayor longitud de prefijo) con UVs menos específicos en otros campos, y se

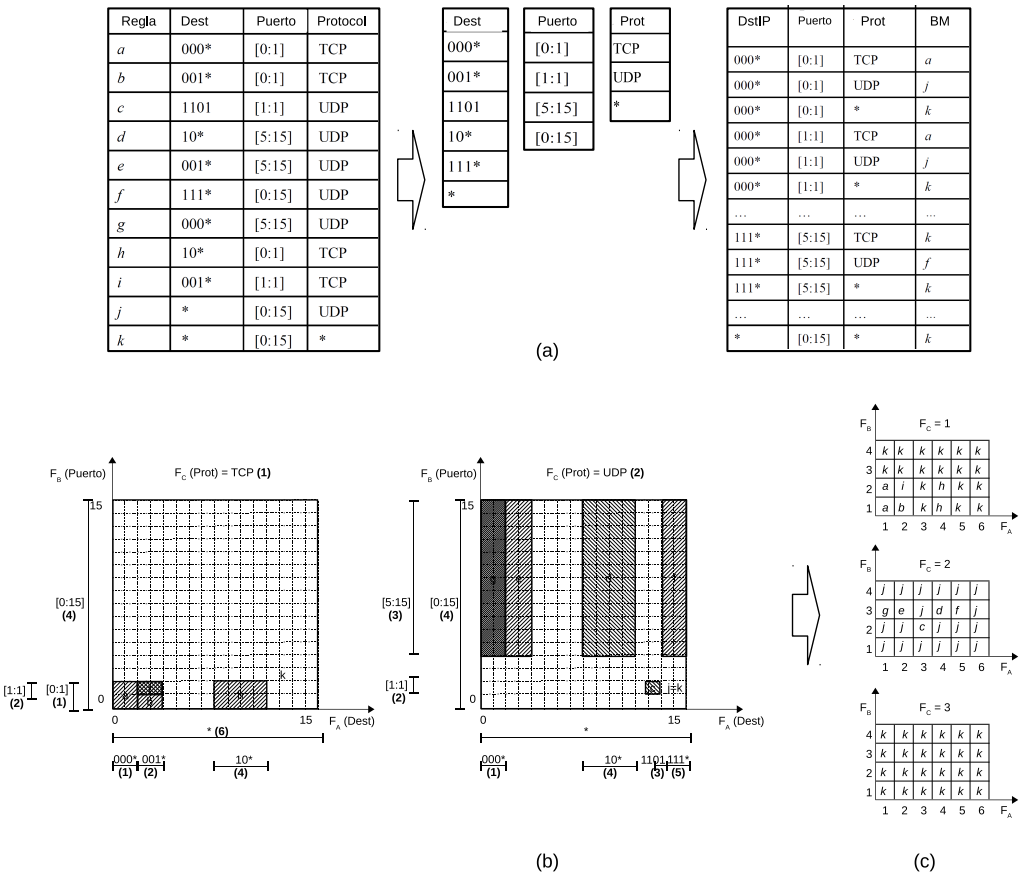
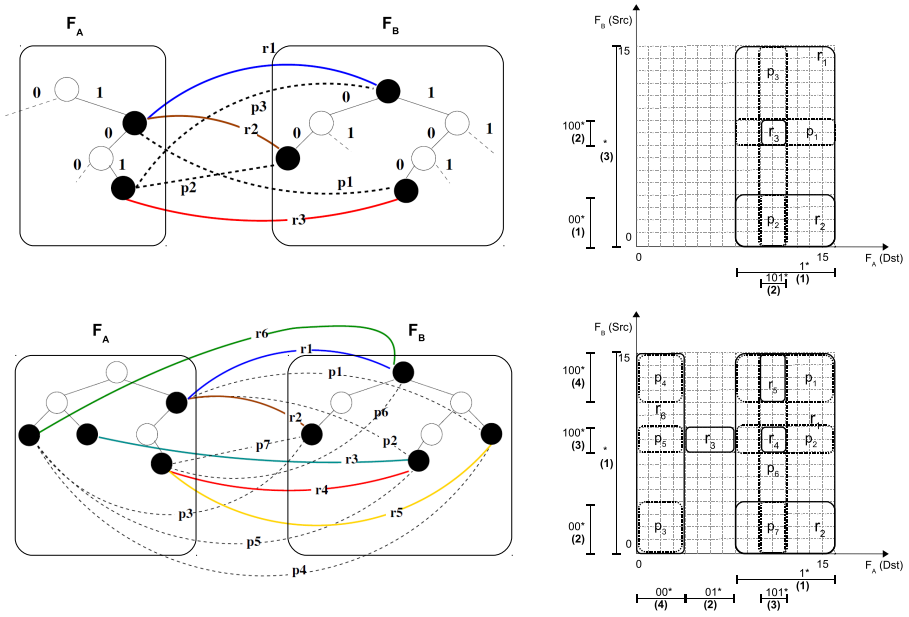


Figura 4.8: Crossproducting (XPROD): (a) principio de operación, (b) interpretación geométrica 3D, (c) tablas de agregación

vuelve más grave a medida que se consideran más dimensiones. Un caso extremo es el de reglas que involucran wildcards en al menos una dimensión, como es el caso de las reglas j y k, donde la tabla de agregación se forma mayormente con pseudo-reglas como muestra la Fig. 4.8(c). Este efecto produce crecimiento exponencial en el requerimiento de memoria, llegando a ser $O(N^k)$ en el peor caso. En la Fig. 4.9 se muestran dos ejemplos adicionales, mencionados en [104], donde se ve claramente la necesidad de utilizar pseudo-reglas en XPROD. Las especificaciones de reglas se muestran en la Fig. 4.9(a), mientras que la Fig. 4.9(b) muestra esquemáticamente las etapas de lookup para F_A y F_B y agregación entre ellas. En este caso las etapas de lookup se implementan mediante tries, donde los nodos rellenos representan resultados válidos para los respectivos campos del ruleset, es decir UVs. Las líneas continuas entre ambos grupos representan reglas pertenecientes al ruleset, mientras que las líneas de trazos representan las pseudo-reglas. Finalmente, la Fig. 4.9(c) muestra la interpretación geométrica de todas ellas.

La propuesta denominada *Producto cruzado distribuido de valores de campo (Distributed Crossproducting of Field Values, DCFL)* [97] alivia el problema de XPROD recurriendo en este caso exclusivamente al uso de UVs. Para ello, los autores analizaron exhaustivamente un conjunto de rulesets reales llegando a las siguientes conclusiones:

F_A	F_B	Regla
1*	*	1
1*	00*	2
101*	100*	3
1	100*	1
101*	00*	2
101*	*	3



F_A	F_B	Regla
1*	*	1
1*	00*	2
01*	100*	3
101*	100*	4
101*	11*	5
00*	*	6

Figura 4.9: Pseudo-reglas: (a) rulesets de ejemplo, (b) productos cruzados, (c) interpretación geométrica

1. el máximo número de valores únicos $|UV|_{max}$ para cada campo es significativamente menor que el número de reglas N. Esto se debe a que múltiples reglas comunmente comparten el mismo valor único de uno o múltiples campos

2. el máximo número de valores únicos *que se cumplen a la vez para cierto encabezado* para un campo es muy limitado y se mantiene relativamente constante para diferentes rulesets. Llamaremos a este parámetro *máxima cantidad de valores coincidentes (maximum matching values, V_{max})*. Según [97] y [83], $V_{max} \leq 5$ para rulesets reales.

3. el máximo número de *combinaciones únicas de valores únicos* (por ejemplo V_{AB}, V_{AC}, V_{AD}, V_{BC}, etc. para cuatro campos) que se cumplen a la vez para cierto encabezado está limitada por $2.V_{max}$. Es decir, si por ejemplo $V_{max} = 5$, los UVs resultantes de la agregación de a pares de campos no serán más de 10 mientras que los resultados de agregar cuatro campos no serían más de 20.

Dos características principales diferencian a DCFL de XPROD. Como se observa en la Fig. 4.6(d), en DCFL se agregan *múltiples etiquetas* para cada paquete, correspondientes a los múltiples UVs que rodean al punto definido por este paquete en el espacio multi-dimensional. Por otro lado, la agregación de tales UVs se realiza en *múltiples etapas*, de este modo se filtran incrementalmente las pseudo-reglas. Es decir, si la combinación de dos UVs 1D diferentes lleva al mismo UV 2D por efecto del solapamiento, sólo una etiqueta se propaga para agregación con otros UVs. En el caso particular de DCFL, este esquema se implementa mediante *nodos de agregación* los cuales reciben todos los UVs matcheados desde dos nodos previos, y seleccionan sólo las combinaciones válidas. Luego de analizar múltiples configuraciones de nodos, se adoptó agregación de a pares.

Ya que cada nodo recibe y genera múltiples UVs (a diferencia de lo que sucede en el caso de

URs), la agregación toma en general múltiples ciclos secuenciales de acceso a memoria *(Sequential Memory Accesses, SMAs)*. Esta es una métrica muy importante para las arquitecturas basadas en UVs, y determina la relación entre el número de matches *efectivos* obtenidos con respecto a los SMAs totales necesarios para obtenerlos. En un cierto nodo de agregación, los SMAs necesarios son el producto de los matches efectivos en los dos nodos previos, mientras que de acuerdo a c) el número de matches efectivos obtenido no es más que el doble del máximo de ellos. Por ejemplo, si se consideran 7 matches efectivos en cuatro campos, lo cual es bastante realista, se pueden requerir $7 \times 7 + 14 \times 14 = 245$ SMAs para obtener sólo $2 \times 14 = 28$ coincidencias efectivas. Los SMAs causan bloqueo del pipeline de agregación, por ello son de gran importancia en el diseño de arquitecturas optimizadas para máximo desempeño.

En DFCL, se proponen dos esquemas optimizados para reducir el número de SMAs requeridos: filtros Bloom y arrays de listas enlazadas. Sin embargo, no se brindan resultados de implementación. Los filtros Bloom, en particular, tienen el inconveniente de generar falsos positivos, por lo que requieren etapas auxiliares para su eliminación. Para solucionar este problema, el esquema de DCFL extendido (DCFLE) propone una implementación alternativa mediante indexado de memoria [83]. Como se profundizará más adelante, DCFL puede ser paralelizado disminuyendo la cantidad de SMAs; DCFLE implementa esta mejora mediante *replicación* de las tablas de agregación lo que aumenta notablemente el consumo de memoria.

4.3.3. Esquemas derivados

A partir de los trabajos fundamentales mencionados, surgieron una serie de propuestas que, sin modificar su base, intentan mitigar sus desventajas. Una línea de trabajo consiste en reducir el ancho de BV recurriendo a *agregación* de segmentos dentro del BV, aprovechando el hecho de que la cantidad de reglas coincidentes es mucho menor que N y por lo tanto los bits activos son muy dispersos. De este modo, se busca reducir la cantidad de accesos a memoria, lo cual es una seria limitación de BV cuando los bloques de RAM disponibles tienen ancho limitado. El problema de este esquema reside en la existencia de *falsos positivos*, que producen accesos innecesarios a memoria. Como primer trabajo en esta línea podemos citar a *Aggregated Bit Vector (ABV)* [109], el cual divide el BV de ancho N en $\lceil N/A \rceil$ secciones para luego tomar sólo la conjunción de las secciones que contienen coincidencias. Para mitigar el problema de falsos positivos, ABV propone un re-ordenamiento previo de reglas, lo cual complica el esquema; ABV además tiene mayor requerimiento de memoria que BV. *Condensate Bit Vector (CndBV)* [110], en tanto, busca reducir el costo $O(k \cdot N \cdot N)$ de BV mediante la reducción del ancho de BV (primer N) y la cantidad de BVs a almacenar (segundo N). Para ello, se observan tablas reales de prefijos para comprimir las estructuras de lookup asociadas; sin embargo en este caso no se ataca el caso de AR. Condensated & Ordered Bit Vector (CndoBV) [111] aporta nuevas optimizaciones a CndBV, mientras que aplica adicionalmente ABV. Otro trabajo, *Compressed Bit Vector (CBV)* [112], se centra en reducir el ancho de BV directamente removiendo los bits que no están en 1. Para ello, las regiones 1D se definen en base a un solapamiento máximo *(Maximum OverlaP, MOP)*, que fija el límite de ancho del bitmap utilizado. Ya que en este caso se pierde la codificación de reglas basada en posiciones de bits, se introduce una segunda

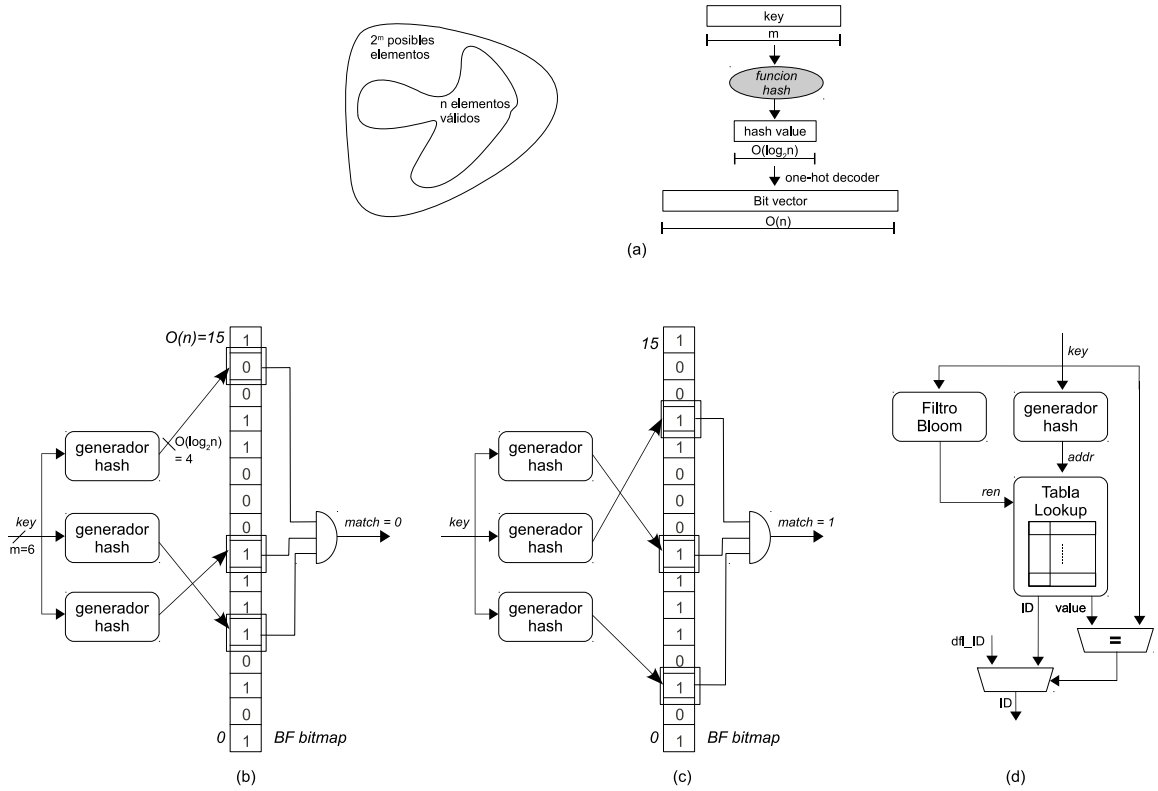


Figura 4.10: Filtros Bloom: (a) función de hashing, (b) caso de match negativo, (c) caso de match positivo, (d) aplicación en esquemas de clasificación

memoria que contiene los números de regla para cada bit en 1 dentro del bitmap reducido. Además, las reglas tipo WC, que afectan seriamente el solapamiento en todas las regiones, son almacenadas en una memoria aparte. De este modo, se logra un balance positivo con respecto al consumo de memoria de BV, disminuyendo la complejidad de almacenamiento desde $O(d \cdot N^2)$ a $O(d \cdot N \cdot log_2 N)$. Aun así, la agregación entre campos se debe efectuar mediante AND de bitmaps al igual que en BV, por lo que los bitmaps optimizados deben ser nuevamente expandidos antes de abandonar las etapas de lookup; esto limita el desempeño de la arquitectura.

Otros trabajos se basan en el uso de hashing y filtros Bloom [113] [114] para reducir el consumo de memoria en esquemas de clasificación por agregación. A modo de repaso, se discutirá brevemente el funcionamiento general de estos filtros y su rol en los esquemas de clasificación. Dado un determinado valor de entrada, un filtro Bloom sirve para determinar la *pertenencia de dicho valor a un determinado grupo*, es decir, entrega un valor binario que indica pertenencia. Supongamos que este valor es el key de ancho m; esta pertenencia se podría determinar simplemente accediendo a un array de tamaño $2^m \times 1$; sin embargo esto puede ser muy ineficiente ya que sólo dos estados (0 o 1) de salida son posibles, conduciendo a muchas repeticiones. Consideremos ahora reducir el espacio de búsqueda requerido a, digamos, n posibles resultados. Éstos se almacenan en un vector de tamaño $n \times 1$, al cual se accede a través de una *función hash* que procesa valores de key y genera posiciones para acceder al vector de resultados. De esta forma, como se muestra en la Fig. 4.10(a), este tipo de funciones

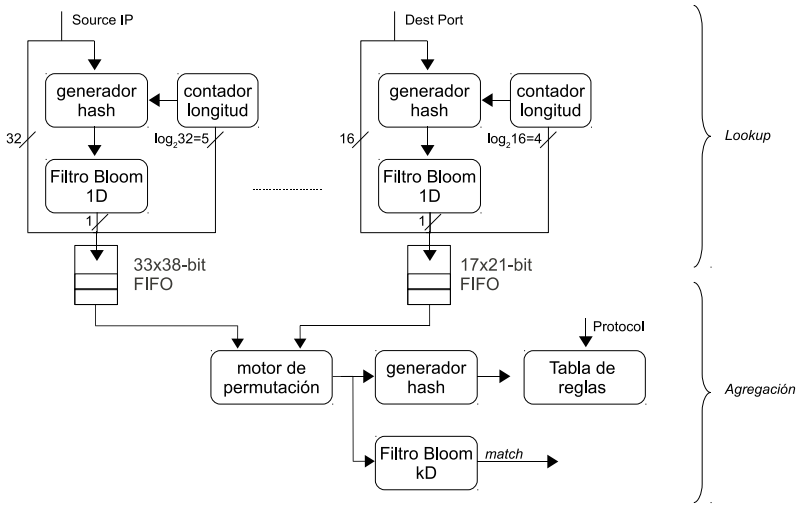


Figura 4.11: Esquema de clasificación basado en filtros Bloom *2sbfce*

pueden extraer, con cierta probabilidad de error, el conjunto de n elementos *válidos* del conjunto de 2^m elementos *posibles* para un key de ancho m. Ya que en general $2^m >> n$, existe una probabilidad *variable* de que un elemento (valor de key) no válido resulte en un hash válido produciendo una *colisión*; se toma para este caso el *máximo de esa probabilidad*. Por ejemplo para un ancho de hash $2 \cdot log_2 n$ la máxima probabilidad de colisión es $1/n$ [114]. El módulo crítico en este esquema es la función hash, que puede por ejemplo implementarse con lógica combinacional; el diseño de estas funciones es un tema muy amplio e interesante que no abordaremos aquí [113] [115]. Los filtros Bloom, en tanto, utilizan h funciones hash accediendo concurrentemente a múltiples posiciones del vector, resultando en una probabilidad *fija y sintonizable* de falsos positivos. Durante el almacenamiento de un nuevo elemento, cada función activa el bit correspondiente a su valor; mientras que durante la búsqueda se toma la conjunción (AND) de los bits resultantes. Mientras que la probabilidad de falsos *negativos* es nula, existe la posibilidad de que se generen falsos *positivos* dado a que una función hash puede llegar a generar la misma posición para dos elementos diferentes; esta probabilidad sin embargo se mantiene constante debido al uso de múltiples funciones. En las Figs. 4.10(a) y 4.10(b) se muestran resultados positivos y negativos respectivamente para un caso $m = 6$ ($2^6 = 64$ posibles elementos), $n = 16$ y $h = 3$. Llevando estas técnicas a nuestra aplicación concreta, la Fig. 4.10(d) muestra la aplicación de estos filtros a un esquema de clasificación; resultando esencialmente en una *búsqueda lineal filtrada*. Es decir, un generador hash auxiliar mapea 2^m valores de key a $n << 2^m$ índices de tabla de lookup; mientras que un filtro bloom indica si ese mapeo es válido para nuestro ruleset o si se trata de una colisión. Si bien este esquema disminuye drásticamente la posibilidad de colisiones, aún se deben eliminar los falsos positivos del filtro, por lo que finalmente se extraen los campos de la regla obtenida y se comparan con el key; en caso afirmativo se entrega el ID obtenido como resultado de la operación de lookup (o agregación según veremos).

El esquema original DCFL [97] utiliza arrays de filtros Bloom en cada nodo de agregación a fin de determinar si las diferentes combinaciones de las etiquetas entrantes son válidas en el ruleset. La propuesta [104], por otro lado, se basa en la técnica [96]. A fin de reducir la cantidad de pseudo-reglas, el ruleset se divide en múltiples sub-grupos, contruyendo una tabla de productos cruzados

para cada uno de ellos. Los filtros Bloom, implementados en memoria on-chip, se utilizan para limitar la cantidad de lookups sólo a aquellos sub-grupos donde existen reglas coincidentes, mientras que dichos lookups se realizan en memoria off-chip. Mediante identificación de tuplas de niveles anidados *(Nested Level Tuples, NLT)*, se separa el ruleset en grupos que no contienen pseudo-reglas; la cantidad de estos grupos puede ser excesiva por lo que luego se aplica un algoritmo de *NLT merging and crossproduct (NLTMC)* a fin de reducir esta cantidad permitiendo cierto grado de solapamiento interno a cada grupo. El esquema se diseña considerando lookups tipo LPM como los mostrados en la Fig. 4.9, por lo que resulta difícil estimar su comportamiento para un caso de lookup AR.

Otro ejemplo representativo del trabajo basado en filtros Bloom es el llamado *Dual Stage Bloom Filter Classification Engine (2sBFCE)* [115]. En este caso, funciones hash combinan los campos del key con longitudes de prefijo generadas secuencialmente, mientras que los filtros Bloom se encargan de determinar la validez o no de cada combinación para el ruleset considerado. Todos los resultados válidos obtenidos en cada campo se agregan mediante un *motor de permutación*, cuyos resultados se llevan asimismo a un filtro Bloom que extrae sólo las combinaciones válidas. Finalmente, una última función hash genera índices para una tabla de reglas; las reglas obtenidas se comparan contra el key para eliminar falsos positivos. Por un lado, podemos observar que esta técnica basa su funcionamiento en el uso de prefijos, debiendo implementar ARs mediante expansión. Por otro lado, se requieren múltiples ciclos dependientes de lectura en memoria, lo que conduce a desempeño reducido. Finalmente, es importate destacar que este esquema no soporta actualización incremental ya que los filtros Bloom concentran información de múltiples reglas; por ello no es posible predecir si la remoción de una regla podría afectar a otras. En la Fig. 4.11 se muestra el esquema de funcionamiento, basado esencialmente en replicación del esquema de la Fig. 4.10(d). En este caso los generadores de hash reciben tanto los valores de campo como una secuencia de longitudes de prefijo; el valor de los hashes mas sus longitudes relacionadas y las validaciones del filtro Bloom se almacenan en colas FIFO para su posterior agregación. Por ejemplo, para el caso *Dest Port* se almacenan 16 bits de puerto, 5 bits de longitud y 1 bit de validación. La agregación consta de un motor de permutación que considera todas las combinaciones validadas por sus correspondientes filtros, y otra función hash que genera índices a la tabla de reglas; mientras que un filtro Bloom final valida tales índices.

En resumen, los filtros Bloom son una opción interesante ya que permiten reducir drásticamente el consumo de memoria en esquemas de clasificación por agregación. En estos esquemas se van considerando todos los posibles resultados de lookup y agregación, mientras que los filtros indican para cada combinación si ésta es válida o no en el ruleset considerado. Debido a su naturaleza estocástica, sin embargo, los resultados reportados pueden contener falsos positivos, mientras que los esquemas son bastante dependientes del set de reglas considerado. Desde el punto de vista de implementación estos esquemas requieren múltiples accesos secuenciales a un mismo bloque de memoria, por lo que su desempeño es bastante limitado [94]. Finalmente cabe mencionar que, para el caso de implementación en FPGAs, se ha reportado que estos esquemas tienen alta complejidad de ruteo [116].

4.4. Arquitecturas de procesamiento en FPGAs

En lo que respecta a aplicación de FPGAs para funciones de clasificación, [94] resume muchas de las optimizaciones posibles para el caso de arquitecturas basadas en árboles o espacios no-ortogonales. Los recursos utilizados en este caso son principalmente BRAMs que almacenan las decisiones internas a cada nodo del árbol; mientras que las optimizaciones se concentran en i) algoritmos para particionado óptimo del ruleset en cada etapa, destinados a minimizar el problema de duplicación de reglas y soportar rulesets grandes, y ii) mapeos optimizados del árbol en etapas de pipelining, a fin de obtener óptimo desempeño y balancear el consumo de memoria en cada etapa del pipeline. Estas técnicas son dependientes del patrón de reglas y requieren intensivo procesamiento del ruleset durante cada actualización. Además, basan su efectividad en el pre-procesamiento de reglas, haciendo intensivo uso de memoria para implementación en FPGAs.

Las técnicas basadas en descomposición, en tanto, han recibido nuevo impulso en los últimos años ya que son naturalmente apropiadas para implementación concurrente en FPGAs, lo que no sucede con los árboles de decisión. Como claros ejemplos de esta tendencia podemos citar [117], [118], [119] y [120]. Estos trabajos explotan tanto los recursos de memoria como los de lógica en FPGAs, sin embargo todos ellos realizan agregación utilizando metadadatos basados en bitmaps kD, tal como lo hace BV. Este es un factor clave que limita su escalabilidad y no permite explotar correctamente las técnicas de descomposición.

4.5. Diseño propuesto

4.5.1. Planteo general

De lo discutido hasta el momento, se pueden extraer las siguientes observaciones:

1. En aplicaciones actuales, el objetivo de la clasificación se centra en hallar la *UR kD* donde se encuentra el encabezado. Esta UR representará el UV kD (regla) de mayor prioridad, en caso de tratarse de una aplicación BM; o una combinación de reglas en caso de tratarse de aplicaciones MM. Bajo este enfoque se atacan todas las aplicaciones posibles mediante un esquema único y flexible.

2. Los esquemas basados en árboles son excesivamente dependientes del patrón de reglas; mientras que los esquemas basados en descomposición lo son en menor medida y se prestan más naturalmente a implementaciones concurrentes en FPGAs.

3. Las etiquetas, como se mostró en la Fig. 4.5, concentran en cada uno de sus bits información proveniente de múltiples reglas, por ello requieren etapas de pre-procesamiento o codificación tanto más complejas como información se concentre. Este efecto es máximo en el caso de RFC (labels representando URs), y algo más relajado en el caso de DCFL (labels representando UVs). El uso de bitmaps, por otro lado, requiere mínimo pre-cómputo por conservar relación

directa entre bits y UVs. Sin embargo, los bitmaps utilizados hasta el momento han sido BVs, en los cuales cada bit se asocia a una regla kD. Este formato, como se discutió, no permite explotar las propiedades de descomposición eficientemente.

4. Los esquemas basados en UVs exigen en general menor pre-cómputo que los basados en URs; esto en el caso de DCFV es a costo de requerir múltiples ciclos de procesamiento para obtener todos los UVs coincidentes.

5. BV es completamente agnóstico del patrón de reglas pero escala según $O(k.N^2)$ [62], por lo que no es adecuado para rulesets grandes. Para mejorar esta escalabilidad, se deben necesariamente explotar las características del ruleset. Los esquemas basados en URs logran máxima optimización del ancho de palabra y óptima velocidad de clasificación mediante mapeo complejo entre el ruleset y los labels almacenados, dificultando la actualización incremental de las estructuras utilizadas. Los esquemas basados en UVs relajan el requerimiento de pre-procesamiento pero requieren múltiples ciclos para realizar la clasificación. Los trabajos que buscan reducir los ciclos requeridos mediante paralelización lo hacen a costa de replicación, aumentando el consumo de memoria en forma inversamente proporcional a la reducción de ciclos requeridos.

6. Los requerimientos de pre-cómputo de las etapas de agregación suelen ser mas críticos que los de las etapas de lookup, ya que deben considerar un número creciente de dimensiones a la vez. En particular, el número de etiquetas a considerar en esquemas basados en UVs es N para kD, mientras que las etiquetas basadas en URs escalan como $|UR|_{AB...k} \leq \sum_{i=1}^{N} i$. De aquí se observa que las etiquetas UV escalan mejor que las basadas en URs; ello es ya que los URs deben ser disjuntos (por lo tanto son mas cantidad) mientras que los UVs no lo son necesariamente.

En base a estas observaciones, nuestra propuesta se basa lo siguiente:

1. En base al punto 3), se consideran metadatos basados en *bitmaps optimizados*, donde cada bit se relacione con un UV *en su respectiva dimensión*. Cada UR se asocia bi-unívocamente con un bitmap diferente, como se muestra en la Fig. 4.12(a), por lo que se los llama *bitmaps disjuntos de valores únicos (disjoint bitmaps of unique values)*. De este modo, se deben determinar URs sólo en las etapas de lookup donde el pre-cómputo es menos crítico, utilizando en la etapa de agregación bitmaps que requieren menor pre-procesamiento. El esquema propuesto se denomina *Producto Cruzado Distribuido de Valores Únicos de Campo (Distributed Crossproducting of (Unique) Field Values, DCFV)*.

2. Por sus características, se espera que esta técnica permita explotación más equitativa de los recursos lógicos y de memoria del FPGA. A fin de analizar y validar este aspecto de la propuesta, se realizan implementaciones y se estudia su consumo de recursos con diferentes parámetros, como número de reglas y características de los campos involucrados.

El esquema propuesto permite reducir el tamaño de los bitmaps utilizados hasta el momento de N a $|UV|$ bits. Para ello, incrementa moderadamente la complejidad de la etapa de agregación con respecto a la utilizada en BV; sin embargo, al continuar utilizando bitmaps, esta etapa puede haceer

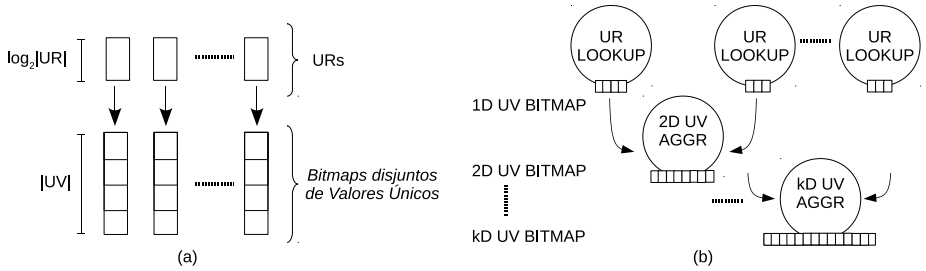


Figura 4.12: Esquema propuesto: (a) etapa de lookup, (b) etapas de agregación

uso intensivo de lógica combinacional a diferencia de RFC o DCFL, que demandan implementación mediante memoria RAM. En la Fig. 4.12(a) se muestra conceptualmente la etapa de lookup en DCFV, mientras que la Fig. 4.12(b) muestra su esquema de agregación en relación a los ya discutidos en la Fig. 4.6.

4.5.2. Un nuevo esquema de agregación

A fin de introducir nuestra propuesta, comenzamos por comparar detalladamente los esquemas CAM y BV. Cuando se realiza lookup en TCAM para un campo d de M_d bits, cada bit de ese campo $K_i(1 \leq i \leq M_i)$ se compara contra cada valor de bit $V_{ji} = 0, 1, X$ correspondiente a las reglas especificadas $R_j(1 \leq j \leq N)$. Los resultados de match para los respectivos $V_j(1 \leq j \leq N)$ son representados por los bits $a_j = 0, 1$, es decir $a_j = 1 \leftrightarrow K_i = V_{ji}\ \forall\ (1 \leq i \leq M)$. Este esquema se ilustra en la Fig. 4.13(a) para un caso $M = 4, N = 8$. Como se observa en forma ampliada, cada nodo en el esquema cumple la doble función de búsqueda y propagación de resultados. En la Fig. 4.13(b), en tanto, se aprecia el esquema de agregación de BV. En este caso, se observa que los bitmaps de entrada $a_1, ..., a_8$ y $b_1, ..., b_8$ provienen de etapas de lookup externas, concentrándonos ahora en la agregación de tales resultados; tal es la diferencia fundamental entre TCAMs y BV. N bits provenientes del lookup de dos campos A y B se agregan mediante N nodos que son esencialmente compuertas AND. Se observa que este esquema de agregación es extremadamente simple, ya que tanto la actualización de reglas como la agregación de resultados son directos. Sin embargo, el esquema es ineficiente ya que se deben obtener resultados de ancho N en cada etapa de lookup; es decir, se debe expandir el resultado de lookup a un formato que considere el impacto de cada UV sobre el ruleset en kD. Una posible alternativa consiste en ajustar el ancho de los vectores de lookup resultantes al número de valores únicos $|UV|$ para cada campo. De este modo, la arquitectura de agregación debe considerar bitmaps locales de UVs a y b, de dimensiones $|UV_A|$ y $|UV_B|$ respectivamente, combinándolos para obtener el vector agregado c de tamaño $|UV_c|$. Este esquema, basado en *bitmaps de UVs*, se ubica en un punto medio entre DCFL y BV; es decir, si bien los tamaños de bitmap se reducen a dimensiones $|UV_A|$ y $|UV_B|$, éstos son aún mayores que los de las etiquetas utilizadas por DCFL, de tamaños $log_2|UV_A|$ y $log_2|UV_B|$ respectivamente, y que las utilizados por RFC, de tamaños $log_2|UR_A|$ y $log_2|UR_B|$ respectivamente.

En nuestro nuevo esquema, se debe incorporar procesamiento a la etapa de agregación, a fin de combinar los resultados 1D (en este caso de tamaños $|UV_A|$ y $|UV_B|$ respectivamente) en resultados

multi-dimensionales (de tamaño $|UV_c|$ en este caso). Al utilizar bitmaps, estas operaciones se pueden implementar mediante una combinación de lógica combinacional y memoria, al contrario de esquemas como DCFL o DCFLE que se basan en memoria. Además, mediante adecuado pipelining, este esquema permite efectuar la agregación en un ciclo. En DCFLE la agregación toma múltiples ciclos ya que se basa en direccionamiento secuencial de memoria, mientras que la paralelización se logra mediante simple replicación de los bloques de memoria. Cabe mencionar que la propuesta en [120] presenta similitudes con la nuestra en cuanto al esquema general utilizado; sin embargo difiere fundamentalmente en que se basa en la comparación directa de etiquetes kD al igual que lo hace BV con sus vectores kD.

En la Fig. 4.13(c) se muestra una arquitectura presentada en [121] que influenció parcialmente nuestras ideas. En ese trabajo, que se abordará en más detalle el finalizar el Cap. 6, se analizan alternativas para disminuir el pre-cómputo necesario en RFC mediante la separación de URs en capas con solapamiento controlado. En el caso extremo donde cada capa contiene sólo un UV se producen bitmaps similares a los aquí considerados. En ese trabajo, sin embargo, la agregación de campos se realiza concatenando los bitmaps obtenidos y utilizándolos como key de entrada a una TCAM, la que los mapea directamente a un formato de BV (tamaño N). Esto es bastante ineficiente desde una perspectiva tecnológica y no es escalable al aumentar $|UV_A|$, $|UV_B|$ y N, además no aprovecha el hecho de que comunmente $|UV_c| << N$ en un esquema de agregación de múltiples etapas.

Desde la interpretación geométrica, observamos que la tarea de agregación en nuestro esquema consiste esencialmente en considerar las combinaciones válidas en el mapa de la Fig. 4.4(c) (cuadrados grises) y extraer aquéllas que se cumplen para el paquete a clasificar (cuadrados negros). Ya que los bits en cada bitmap son independientes unos de otros, podemos asignar "clusters" o segmentos de este mapa a elementos de procesamiento (Processing Elements, PEs) de granularidad configurable, los que colectivamente implementan dicho mapa; de este modo se puede explotar efectivamente la flexibilidad ofrecida por plataformas FPGA en una arquitectura concurrente. Este concepto se ilustra en la Fig. 4.13(d) donde los productos cruzados (c's subrayados) se extraen del producto cartesiano (todos los c's); la ubicación de productos cruzados dependerá del formato del ruleset y puede incluso ser explotada estadísticamente para definir el esquema de pipelining más eficiente.

4.5.3. Arquitectura de hardware

4.5.3.1. Alternativas de implementación

Consideremos, sin pérdida de generalidad, un caso simple 2D donde $|UV_A| = |UV_B| = 8$ y $|UV_{AB}| = 16$, mientras que se considera que un máximo de 5 UVs pueden coincidir a la vez en cada campo para un determinado encabezado [97]. Como punto de partida del análisis de arquitecturas, se muestra en la Fig. 4.14(a) una implementación de DCFL totalmente concurrente mediante direccionamiento de memoria, tal como lo propone DCFLE [83]. En este caso, la entrada a la arquitectura son 5 etiquetas de ($log_2 8 = 3$) bits para cada campo, mientras que la salida consta de $5 \times 5 = 25$ etiquetas de $log_2 16 = 4$ bits cada una, de las cuales $|UV_c| \leq 2.max\{|UV_A|, |UV_B|\} = 16$ serán pro-

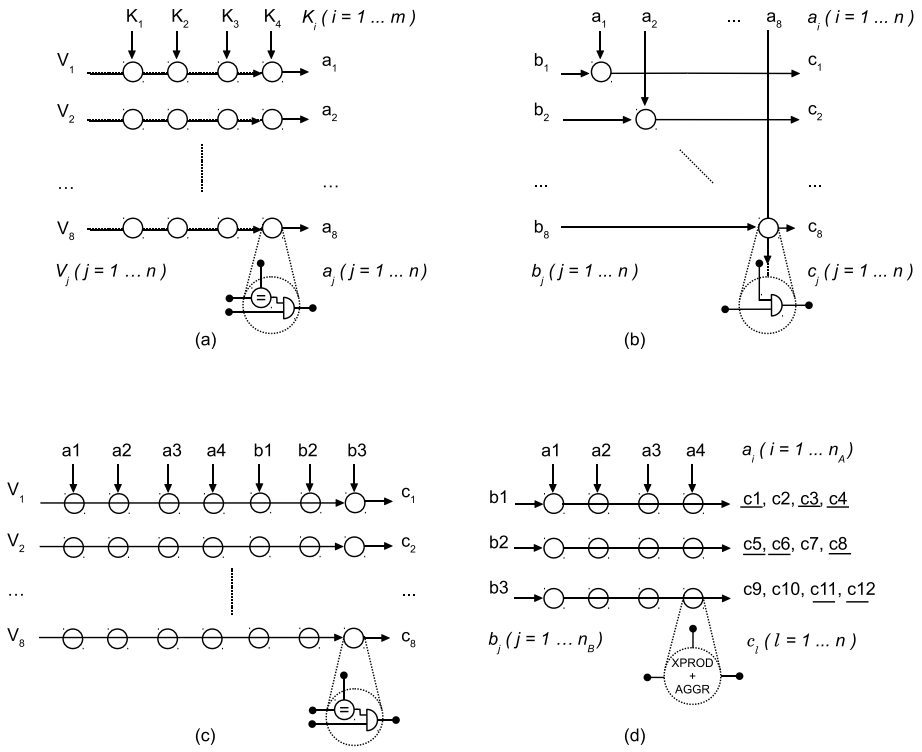


Figura 4.13: Razonamiento del esquema DCFV: (a) lookup en TCAM, (b) BV, (c) PPC style I, (d) DCFV

ductos 2D válidos. Asumiendo un caso de memoria de un puerto y agregación en un ciclo de reloj, se requieren $5 \times 5 = 25$ instancias de un bloque SRAM de tamaño $2^{3+3} \times 4 = 64 \times 4$. Con esto, sería posible determinar en un ciclo todas las coincidencias en un mapa como el de la Fig. 4.4(b). Consideremos ahora un esquema de direccionamiento de memoria con entradas basadas en bitmaps de anchos $|UV_A|$ y $|UV_B|$, tales como las aquí propuestas. Tal esquema requeriría excesiva profundidad de memoria si se implementara en un bloque único, por lo que se divide en segmentos de granularidad variable. Así, se pueden obtener distintos casos de granularidad llamados en conjunto *DCFV de granularidad gruesa (coarse-grained DCFV)*, representados mediante la Fig. 4.14(b), llegando finalmente al caso *DCFV de granularidad fina (fine-grained DCFV)* ilustrado en la Fig. 4.14(c). Aún así, las arquitecturas basadas exclusivamente en memoria son ineficientes para casos más grandes, por lo que se consideran alternativas basadas en lógica combinacional y en memoria/lógica. Estas alternativas, en conjunto, permiten evaluar compromisos de diseño para el caso de FPGAs.

En la Fig 4.14(d) se reproduce el mapa de la Fig. 4.4, mediante el que se pueden comparar claramente las opciones mencionadas. Como se puede observar, el caso de DCFL paralelo considera en cada bloque de memoria todo el mapa de UVs 2D (64 UVs posibles, de las cuales se considera sólo 16 pueden ser válidas). Por otro lado, cada bloque sólo puede entregar una coincidencia a la vez por su ancho de memoria altamente optimizado $log_2 16 = 4$, razón por la cual se debe replicar a fin de entregar las máximas posibles coincidencias en un ciclo (en este caso 5 an cada campo, y 25 en total). A partir de este extremo, se puede segmentar o *sectorizar* el mapa de UVs e implementarlo mediante bloques más chicos; sin embargo DCFL continuaría requiriendo replicación ya que no

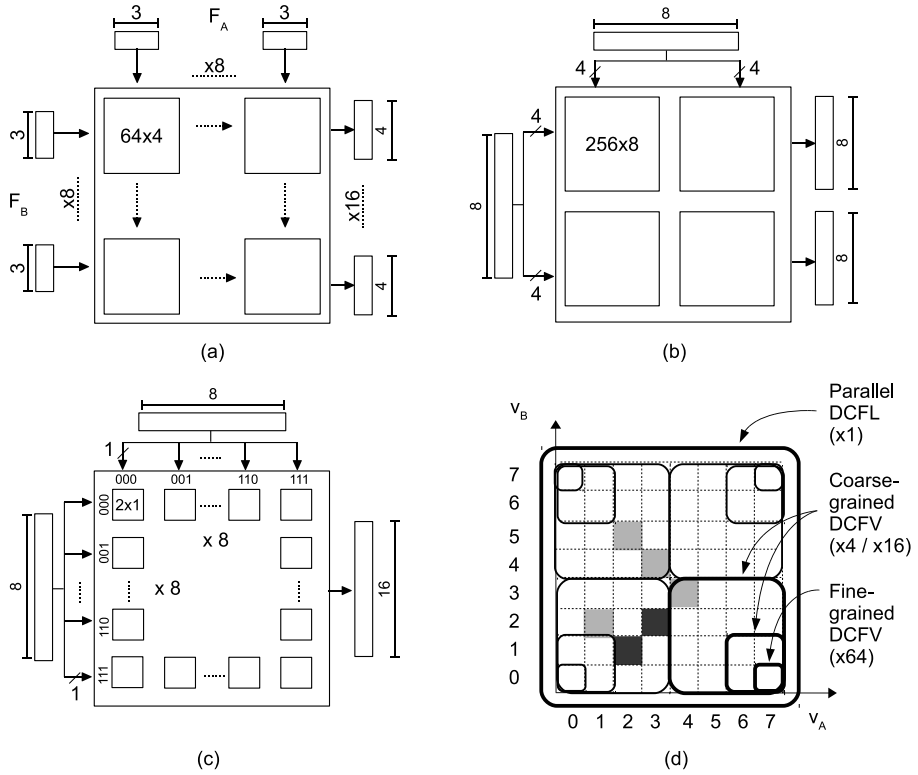


Figura 4.14: Opciones de agregación: (a) DCFL concurrente, (b) coarse-grained DCFV, (c) fine-grained DCFV, (d) mapeo de UVs para cada caso

es posible predecir en qué grupo se darán las múltiples coincidencias. DCFV puede resolver esto adoptando bitmaps, los cuales pueden ahora escalar correctamente gracias a la sectorización del mapa de UVs; en la Fig. 4.14(d) se muestran ejemplos para 4 sectores (16 UVs/sector) y 16 sectores (4 UVs/sector). En el extremo opuesto se encuentra fine-grained DCFV, con 64 sectores de 1 UV cada uno.

4.5.3.2. Pipeline simple (1-d)

El esquema planteado da lugar a una serie de arquitecturas hardware; en particular se pueden aplicar distintos esquemas de pipelining que dan lugar a compromisos entre espacio y tiempo requeridos [122]. Se comenzará considerando un pipeline uni-dimensional simple (1-d) donde cada PE considera un bit proveniente de cada campo (granularidad 1-b o 1-b grained); este esquema se ilustra en la Fig. 4.15(a) para el caso de agregación de dos bitmaps de A y B de tres bits cada uno en otro C de 9 bits. En esta representación, los registros (delay elements, DEs) de pipeline se representan mediante puntos, V representa la validez de la combinación de bits para el ruleset considerado, y M indica si dicha combinación se cumple para un determinado encabezado de paquete. Para el caso considerado $|UV_A| = 3, |UV_B| = 3$, el pipeline 1-d 1-b consta de $3 \cdot 3 = 9$ PEs con respectivos $3 \cdot 3 = 9$ DEs. Además, se deben disponer $2 \times \sum_{i=1}^{|UV_A||UV_B|} i$ DEs de entrada a fin de sincronizar el procesamiento de datos. Un aspecto fundamental de esta arquitectura reside en el metadato propa-

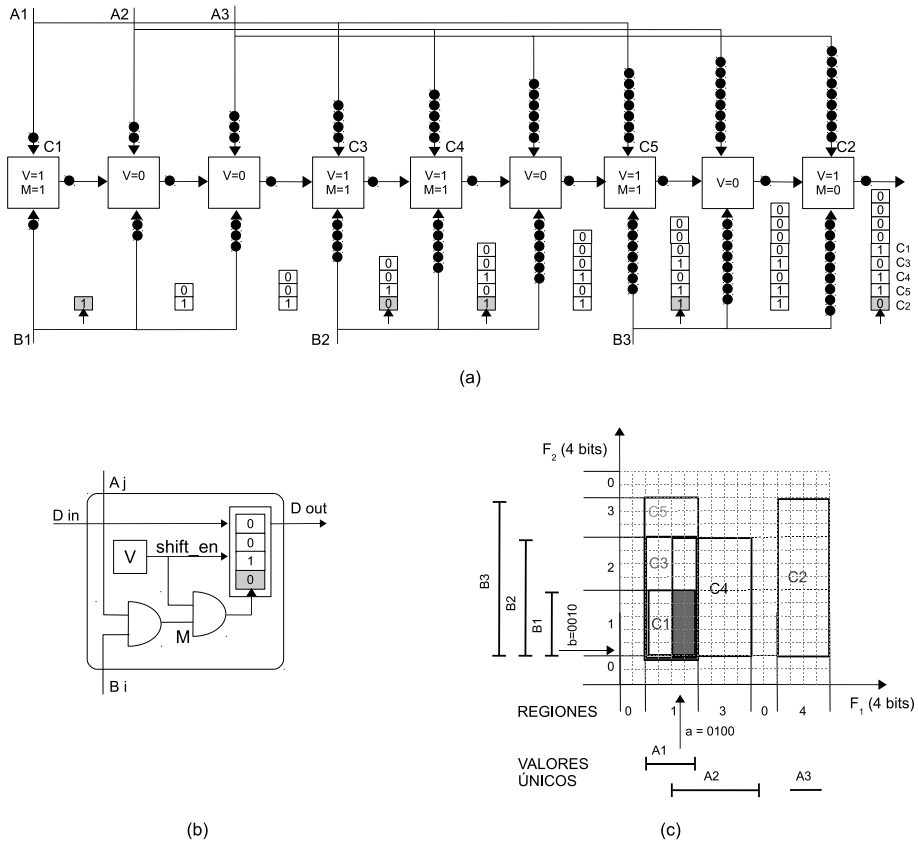


Figura 4.15: Pipeline 1-d 1-b: (a) esquema general, (b) PE, (c) caso de agregación

gado a lo largo del pipeline, el cual es en este caso un bitmap. A diferencia de lo que ocurre en otras arquitecturas de procesamiento de patrones donde sólo se debe propagar un bit de coincidencia [123], debemos en este caso propagar múltiples bits conteniendo información de múltiples coincidencias. En este primer caso, se adoptó un esquema basado en registros de desplazamiento; si $V = 1$ para un determinado PE, el resultado de match M correspondiente se ingresa al bitmap propagado corriendo un lugar los demás resultados, esto se ilustra en el PE de la Fig. 4.15(b). El funcionamiento de esta arquitectura se ilustra finalmente en la Fig. 4.15(a) mediante un ejemplo de encabezado $a = 0100,\ b = 0010$, utilizando el ruleset de la Fig. 4.15(c). En este ejemplo, tenemos $|UV_A| = 3$,$|UV_B| = 3$ y $|UV_C| = 5$, mientras que los UVs 2D coincidentes son C1,C3, C4, y C5.

4.5.3.3. Pipelines 2-d

El desempeño de la arquitectura básica de la Fig. 4.15(a) se degrada fuertemente cuando la cantidad de UVs considerados aumenta, por lo que se deben considerar otras opciones de pipelining de mejor escalabilidad. En particular, se estudian esquemas en dos dimensiones (2-d), inspirados en los conceptos de [122] y [123], llamados *pipelines sistólicos*. Estos esquemas han sido exitosamente aplicados a otros casos de computación de altas prestaciones y han sido recientemente explorados para clasificación [119] [120]. Ellos permiten la ejecución simultánea de múltiples pipelines y son

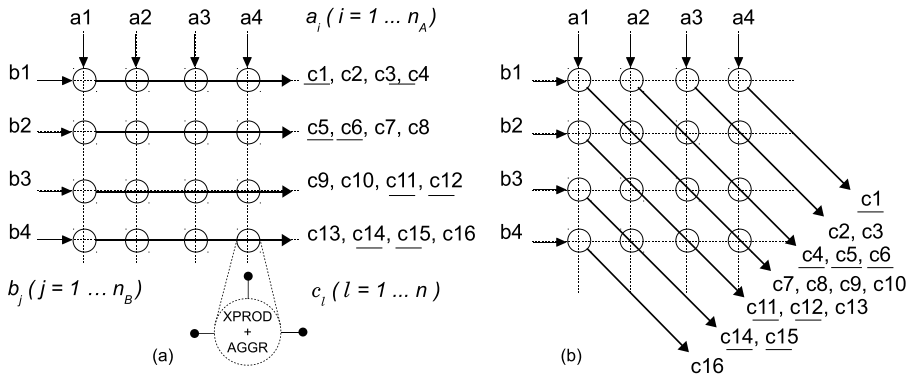


Figura 4.16: Dos posibles sentidos de propagación: (a) por filas, (b) por diagonales

particularmente apropiados para aplicaciones donde el procesamiento sigue un esquema regular y requiere alto desempeño.

Los pipelines 2-d pueden adoptan diferentes sentidos de propagación tales como los mostrados en la Figs. 4.16(a) y 4.16(b) [124], nosotros adoptamos el primero de ellos por presentar distribución más homogénea entre filas de agregación que entre diagonales. En forma similar al pipeline 1-d, se propagan dos registros en cada fila: el vector de resultados MM parciales, y los bitmaps de entrada.

Ya que no hay necesidad de agregar entre sí los UVs de cada campo, se pueden agregar en paralelo los bits de un campo (por ejemplo el bitmap a) contra cada uno de los bits del otro campo (por ejemplo los bits b_1, b_2, etc.); de este modo se obtendrán en paralelo $|UV_B|$ bitmaps parciales de ancho $|UV_A|$ con latencia de $|UV_A|$ ciclos como se muestra en la Fig. 4.17. Los bitmaps obtenidos de cada fila así definida deben desplazarse una cantidad igual al número de $V = 1$ de la fila, por lo que se incluyen *barrel shifters* al fin de cada una de ellas. Finalmente, tales bitmaps son agregados mediante unión (OR) en un bitmap final C de ancho $|UV_{AB}|$. En esta arquitectura, uno de los bitmaps de entrada se propaga directamente, mientras que el otro lo hace mediante DEs; por ello se lo llama arquitectura *semi-sistólica (semi-systolic)*. Una segunda opción involucra la propagación de *ambos bitmaps* de entrada mediante DEs; resultando en una arquitectura *completamente sistólica (full systolic)*. Esta última arquitectura presenta mayor consumo de recursos y requiere pipelining adicional de la etapa de agregación de salida como se muestra en puntos grises en la Fig. 4.17, por lo que se debe analizar la conveniencia de una u otra según el caso.

4.5.3.4. Esquemas de granularidad gruesa (coarse grained)

En la Fig. 4.17 se observa que el cómputo realizado en cada PE es muy simple, mientras que la cantidad de etapas de pipelining crece linealmente con $|UV_A|$. Es conocido que las técnicas de pipeling proveen beneficios incrementales decrecientes (diminishing returns) superada cierta cantidad de etapas, ya que los retardos de interconexión y registros asociados comienzan a perjudicar la ganancia de dividir el procesamiento en múltiples etapas. Esto dificulta también la adopción de arrays full-systolic, los cuales tienen el potencial de mejorar notablemente la escalabilidad. Para

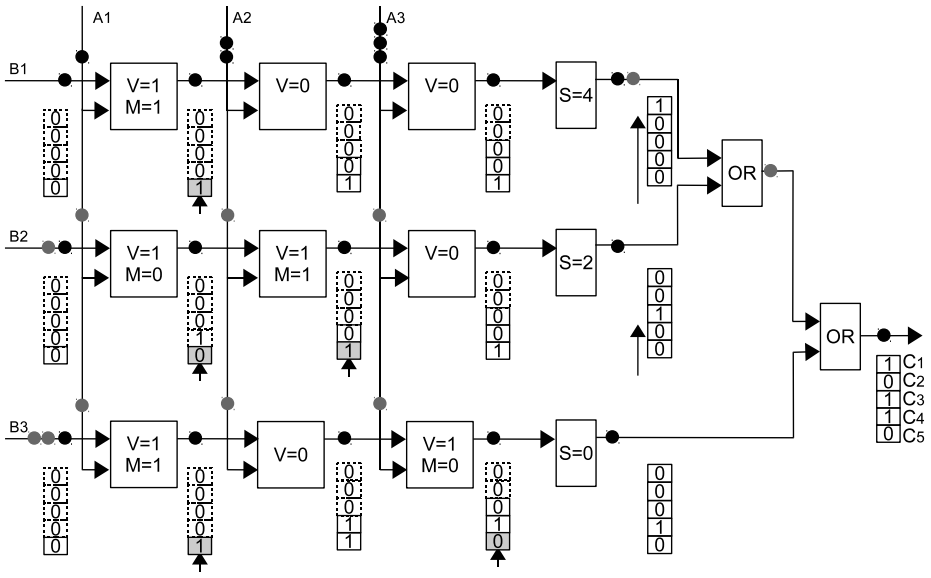


Figura 4.17: Pipeline 2-d 1-b: camino de datos de agregación

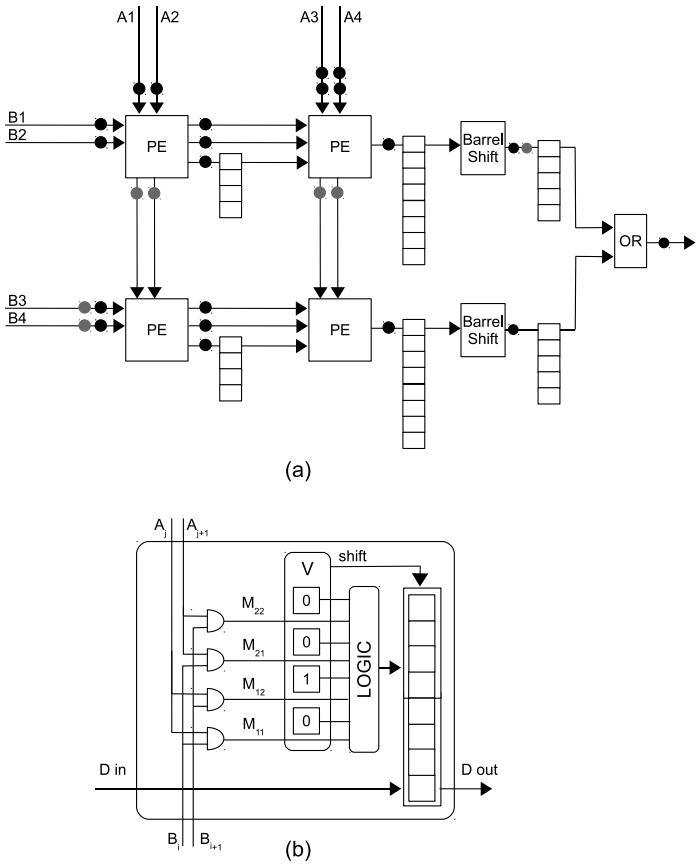


Figura 4.18: Pipeline 2-d 2-b: (a) camino de datos de agregación, (b) PE

solucionar este problema, se introducen diferentes *granularidades* de procesamiento, llevando a la práctica el concepto de *DCFV de granularidad gruesa* analizado en las Figs. 4.14(b) y 4.14(d).

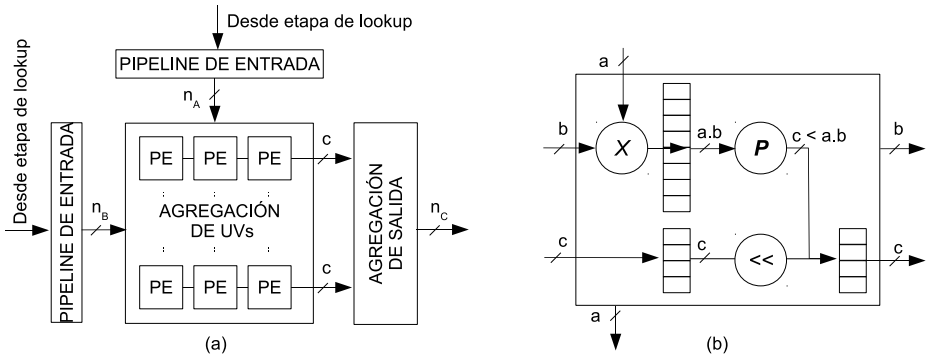


Figura 4.19: Coarse-grained DCFV: (a) arquitectura genérica, (b) PE

Decimos así que, para PEs que procesan $a \times b$ bits, la granularidad del esquema es $g = a \times b$. De este modo, se pueden controlar la latencia y consumo de recursos, obteniendo desempeño más escalable con el tamaño del ruleset. El costo de estos beneficios es que cada PE debe incorporar capacidad para procesar múltiples bits de cada bitmap, por lo que su implementación es más compleja y debe ser analizada cuidadosamente.

En la Fig. 4.18(a) se muestra una arquitectura 2-d 2-b ($g = 2 \times 2$) que procesa dos bitmaps de tamaños $|UV_A| = 4, |UV_B| = 4$ y entrega resultados $|UV_{AB}| = 8$. La Fig. 4.18(b), en tanto, muestra un esquema general para los PEs asociados. En este esquema, no se conoce en principio una implementación eficiente para el bloque combinacional referenciado como *Logic*, el cual se abordará en los siguientes párrafos. Es de destacar que todos estos esquemas conservan un mapeo donde cada bit representa un UV sin depender de los demás, por ello decimos que las agregaciones consideradas se basan en *bitmaps*. Un importante beneficio de esto es que la inserción o eliminación de una regla implica una simple decodificación de su correspondiente identificador *(Rule ID, RID)* y respectiva modificación del bit de validez V en el PE correspondiente. Esta característica es compartida por los esquemas basados en BV; sin embargo BV no posee la flexibilidad de DCFV para ajustarse al valor de $|UV|$ en cada campo ahorrando recursos, ni de explotar los recursos lógicos del FPGA en forma flexible como veremos que lo hace DCFV.

La Fig. 4.19 muestra un esquema genérico para la arquitectura DCFV coarse-grained, donde se considerará el caso full-systolic. Como se indica en puntos grises en las Figs. 4.17 y 4.18(a), este tipo de pipelining requiere pipelines auxiliares de sincronización en las entradas a la matriz de procesamiento, indicadas como *pipeline de entrada* en la Fig. 4.19(a). Además, se muestra el pipeline de agregación de filas ubicado en la salida, indicado como *agregación de salida*. Como se comentó, se espera que los esquemas coarse-grained ($g > 1$) sean más escalables que los esquemas fine-grained; sin embargo estos esquemas a su vez requieren más procesamiento en cada PE lo que afecta su desempeño. De este modo, el desempeño es limitado por los retardos de registros y ruteo, en el extremo de granularidad fina; y por el camino crítico de cada PE, en el extremo de granularidad gruesa. Para buscar el compromiso más conveniente, se analizan a continuación diversas arquitecturas de implementación del PE.

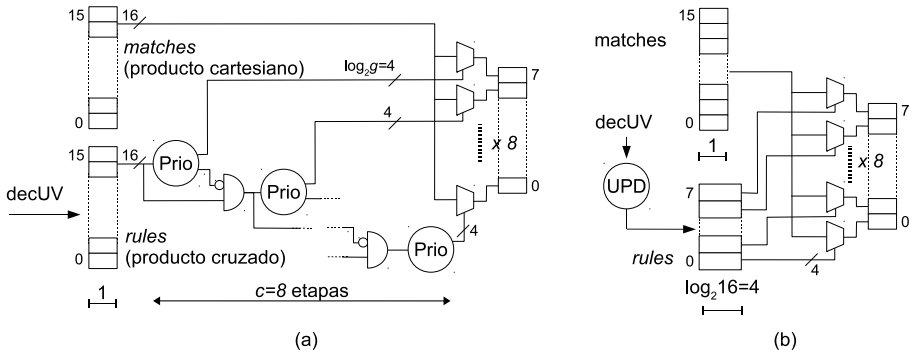


Figura 4.20: Arquitectura del PE: (a) basada en lógica, (b) híbrida

4.5.3.5. Elemento de Procesamiento (PE)

En la Fig. 4.19(b) se muestra un modelo genérico del PE a implementar. Este elemento evalúa todas las posibles combinaciones de a UVs del campo A con b UVs del campo B, entregando un resultado de tamaño reducido $c < a \cdot b$. El bloque indicado como X evalúa el producto cartesiano $a \times b$, mientras que el bloque P se encarga de extraer los productos que son válidos para el ruleset considerado constituyendo la clave para optimizar el esquema. Según se vio en la Fig. 4.14(b), realizar este procesamiento mediante direccionamiento de memoria no es eficiente; por lo que el bloque P distribuye este procesamiento entre lógica combinacional, registros y memoria, permitiendo evaluar compromisos entre ellos. Mediante apropiado diseño, el bloque P determina el camino crítico del pipeline de agregación. Se debe notar que el PE no debe ser necesariamente simétrico; por ejemplo, la granularidad $g = 8$ puede provenir de configuraciones 2×4 o 4×2. Sin embargo, algunas simetrías pueden ser más convenientes que otras para la etapa de agregación de filas, como veremos más adelante.

En principio, el PE podría entregar el producto cartesiano de a y b, de ancho $a \cdot b$. Sin embargo, como se comprueba en [97], para $|UV_A|$ y $|UV_B|$ valores únicos agregados sólo se puede producir un máximo de $|UV_{AB}| < |UV_A| \cdot |UV_B|$. En efecto, para los rulesets analizados, se concluyó que $|UV_{AB}| < 2 \cdot max\{|UV_A|, |UV_B|\}$, esto es válido para la matriz completa de agregación. Para propagación de resultados por filas como en la Fig. 4.16(a), en tanto, podemos asumir que en cada fila de PEs se procesan como máximo $c = \frac{|UV_{AB}| \cdot b}{|UV_B|}$ productos válidos. Como consecuencia, los PEs propagarán bitmaps de tamaño reducido $c < a \cdot b$.

Como resultado de nuestra investigación, surgieron tres opciones de implementación principales para PE: basada en *lógica*, basada en *memoria* (ya discutido), e *híbrida*. Las opciones basadas en lógica e híbrida se discutirán a continuación.

PE basado en lógica combinacional. En esta arquitectura, no se incluye pre-procesamiento alguno del ruleset por lo que la complejidad de actualización y el consumo de memoria son mínimos. En cambio, todo el procesamiento para extraer los productos válidos del espacio total de productos y propagar sólo estos productos se realiza en lógica combinacional durante la clasificación. Nuestra primera implementación fue totalmente ad-hoc, describiendo un caso particular mediante tablas de

verdad en lenguaje HDL. Este método se vuelve impracticable para granularidades $g > 2 \times 2$, ya que se deberían especificar bitmaps de salida para $(2^a \cdot 2^b)$ productos multi-match posibles; por ello se buscó definir un modelo general para implementacion de alto nivel en HDLs. El modelo obtenido se ilustra en la Fig. 4.20(a) para un caso general $g = 16$ y $c = 8$. Este modelo consiste de c codificadores de prioridad en cascada, los cuales extraen y agrupan c productos válidos de los g productos cartesianos originales, obteniendo esta agrupación mediante c multiplexores de g entradas. Como se observa en la Fig. 4.20(a), este modelo utiliza un array de registros *rules* a fin de extraer los productos válidos para el ruleset almacenado; sin embargo esta arquitectura es aún basada en lógica combinacional ya que el procesamiento de clasificación se realiza en forma totalmente combinacional. En consecuencia, la actualización de reglas es totalmente directa, almacenando el nuevo UV decodificado en one-hot $decUV$ en la posición correspondiente del array *rules*.

PE híbrido. Esta arquitectura surgió de la necesidad de obtener una solución intermedia entre los esquemas basados en lógica y en memoria, que por sí mismos no logran buena escalabilidad. En este caso, el concepto fundamental consiste en migrar parte del procesamiento de clasificación a la fase de actualización, almacenando la codificación binaria de UVs en memoria y eliminando así los codificadores en cascada de la Fig. 4.20(a). La agrupación de productos válidos se realiza aún mediante multiplexores implementados en lógica combinacional, sin embargo el retardo del camino crítico es mucho menor debido a la eliminación de elementos en cascada. Este esquema se muestra en la Fig. 4.20(b). En este caso, el array *rules* se implementa mediante bancos de registros o *regfiles*, ya que por su tamaño reducido no es conveniente utilizar BRAM, mientras que se debe acceder a sus múltiples posiciones en forma concurrente.

4.5.3.6. Etapa de agregación de salida

De los posibles sentidos de propagación, se adoptó en nuestros ensayos el ilustrado en la Fig. 4.16(a) por su mayor homogeneidad en la distribución de resultados. Ya que cada fila de PEs obtiene sólo una parte de los resultados de clasificación, es necesaria una etapa de salida que agregue los resultados; esta etapa debería ser adecuadamente diseñada para no afectar el desempeño del esquema completo. La etapa de salida toma $\lceil |UV_{AB}|/c \rceil$ bitmaps de ancho c y los agrega mediante concatenación en un bitmap de salida de ancho $|UV_{AB}|$. Se debe notar que, en general, no todos los bits c de cada fila serán resultados válidos, por lo que la etapa de salida debe seleccionar los que lo son en cada fila previo a su concatenación. Además, ya que las latencias de propagación en cada fila son diferentes, se deben introducir registros de pipelining adicionales en la etapa de salida. Luego de ensayar varios diseños, se adoptó la implementación de la Fig. 4.21, la que agrupa y concatena resultados de pares de filas. La selección de UVs en cada una de ellas se realiza mediante barrel shifters controlados por los registros *#rules* como se observa en la misma figura.

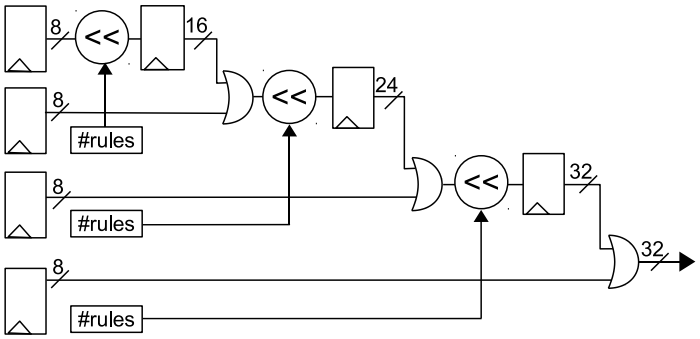


Figura 4.21: Pipeline de agregación de filas

4.5.4. Actualización dinámica

El esquema DCFV hereda de las TCAMs su actualización directa, ya que como aquéllas utiliza mapas de bits para codificar directamente el ruleset. Los UVs involucrados en la regla a actualizar son decodificados a fin de actualizar los bitmaps correspondientes a cada dimensión. En caso de utilizar PEs basados en lógica no se requiere otra operación; sin embargo para los PEs híbridos se debe actualizar su memoria interna mediante un sencillo mecanismo que se detalla a continuación. Finalmente, se debe incrementar o decrementar el registro *#rules* correspondiente al PE involucrado, a fin de considerar la modificación realizada en la fila correspondiente como se ilustra en la Fig. 4.21.

El esquema de agregación DCFV basa su codificación en *posiciones* de bits, por lo que es fundamental mantener esta información durante la actualización. En el caso híbrido en particular, se debe buscar la posición de memoria correspondiente al almacenar una nueva regla en el regfile *rules* de la Fig. 4.20(b). Para hacer esto en un ciclo de reloj, el nuevo UV codificado se compara concurrentemente contra los UVs ya presentes; esta operación se asemeja a la utilizada para lookup en memorias ETCAM [72]. En la Fig. 4.22 se muestran distintos ejemplos de actualización del regfile para el caso donde se extraen $c = 8$ productos cruzados del espacio total de productos cartesianos $g = a \cdot b = 16$. El bloque *ENC* se encarga de codificar el UV entrante, previamente decodificado para almacenarse en el PE adecuado del array; mientras que un comparador de magnitud implementa el ordenamiento de UVs. Se debe notar que todas las posiciones de *rules* se incializan con el identificador más alto, en este caso 15, a fin de preservar el ordenamiento. Los lugares que almacenan tales productos de "relleno" serán automáticamente eliminados por los registros de desplazamiento ubicados al fin de cada fila, los cuales son controlados por los registros *#rules* que almacenan el número de UVs que contienen las filas. En la Fig. 4.22(a), por ejemplo, se encuentran dos UV iniciales 5 y 9. A partir de este estado, se muestran tres casos de actualización donde el nuevo UV se almacena (b) al tope de la lista, (c) en una posición intermedia, y (d) al inicio de la misma.

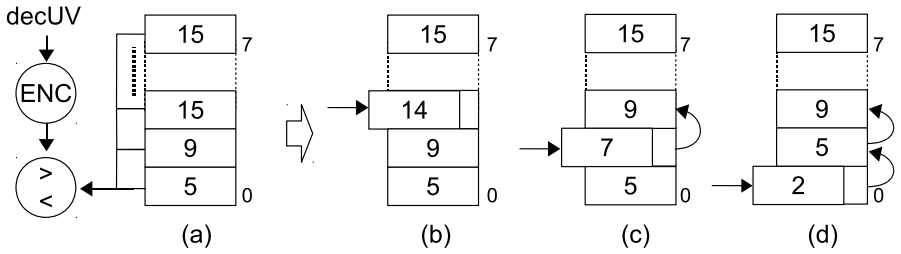


Figura 4.22: Actualización del PE híbrido: (a) estado inicial, (b)(c)(d) casos de actualización

4.6. Resultados

Los resultados de estimación e implementación obtenidos se dividirán en dos partes principales, de acuerdo a como se desarrolló el trabajo. En primera instancia se analizan esquemas basados puramente en lógica, mientras que en un segundo análisis se incorporan las versiones basadas en memoria e híbrida.

Las métricas principales a tener en cuenta en arquitecturas de clasificación son i) consumo de recursos (registros, LUTs y SRAM), ii) velocidad de procesamiento (desempeño), y iii) complejidad de actualización. La métrica i) se analizará en detalle, evaluándola luego junto con ii) en casos de implementación sobre FPGAs. En cuanto a la complejidad de actualización, como se demostró será mínima y similar a la de BV, por lo que estará en última instancia determinada por la complejidad de las arquitecturas de lookup asociadas. Este punto se aborda en el Cap. 5.

4.6.1. Estudio analítico

Para el primer caso analizado, basado puramente en lógica, se debió adoptar una solución de compromiso. Por un lado, se comprobó que las arquitecturas 1-d 1-b presentan escalabilidad muy limitada, por lo que no son profundizadas. Por otro lado, para nuestra primera implementación basada en tablas de verdad se utilizó el esquema de pipelining semi-sistólico con granularidad no mayor a $g = 2 \times 2$. Los recursos a considerar son registros, lógica combinacional y memoria. Por claridad de presentación, en las figuras siguientes se adopta la nomenclatura v_A, v_B y v_C en lugar de $|UV_A|$, $|UV_B|$ y $|UV_C|$ respectivamente.

En primer término analizaremos las ventajas de adoptar granularidad gruesa para un PE basado totalmente en lógica combinacional. El análisis se basará en el consumo de registros; el consumo de lógica se obtendrá de los resultados de síntesis mientras que el consumo de memoria no se considera en este primer caso. Por simplicidad, llamaremos al tamaño del bitmap agregado $|UV_{AB}| = |UV_C|$. Como se puede comprobar por inspección de la Fig. 4.17, el consumo de registros para la arquitectura semi-sistólica 2-d 1-b es $\sum_{i=1}^{|UV_A|} i + |UV_B| + |UV_A||UV_B| + |UV_B| \sum_{i=1}^{|UV_A|} i + |UV_B||UV_C| + |UV_C| + |UV_A||UV_B|$, mientras que la arquitectura 2-d 2-b semi-sistólica de la Fig. 4.18 lo reduce a $\sum_{i=1}^{|UV_A|} i + |UV_B| + \lceil \frac{|UV_A|}{2} \rceil |UV_B| + \lceil \frac{|UV_B|}{2} \rceil \sum_{i=1}^{\lceil \frac{|UV_A|}{2} \rceil} 4i + \lceil \frac{|UV_B|}{2} \rceil |UV_C| + |UV_C| + 4 \lceil \frac{|UV_A|}{2} \rceil \lceil \frac{|UV_B|}{2} \rceil$ registros. Los

casos de mayor granularidad (3-b, 4-b, etc.) pueden comprobarse de manera similar. En la Fig. 4.23(a) se observan los consumos de arquitecturas 1-b, 2-b, 3-b y 4-b para los casos $|UV_A| = |UV_B| = 64$ (4 curvas superiores), $|UV_A| = |UV_B| = 32$ (4 curvas intermedias), y $|UV_A| = |UV_B| = 16$ (4 curvas inferiores), todo ello en el rango $0{,}1 \leq \frac{|UV_C|}{|UV_A||UV_B|} \leq 0{,}5$. De esta forma, se cubre por ejemplo el rango $|UV_C| \leq 0{,}5 \times 32 \times 32 = 512$ para $|UV_A| = |UV_B| = 32$, lo que representa casos realistas. En la Fig. 4.23(b), en tanto, se estiman los registros requeridos para un caso intermedio $|UV_C| = 100$ y $|UV_A|, |UV_B|$ variables. A fin de que el producto cruzado $|UV_C|$ se mantenga lo más cercano posible al producto cartesiano $|UV_A||UV_B|$, es decir el peor caso, se hace en todo momento $\lfloor \frac{|UV_A||UV_B|}{|UV_C|} \rfloor = 1$. De estas curvas se puede observar que el caso 2-d 1-b presenta el mayor consumo de recursos en ambos casos, mientras que los casos 3-b y 4-b reducen modestamente el consumo. El caso 2-d 2-b, en tanto, resulta un compromiso conveniente de complejidad vs. consumo.

Con base en los resultados obtenidos, en nuestro siguiente análisis se considera un esquema 2-d 2-b con PE basado en lógica, siempre dentro de la versión descripta mediante tablas de verdad. Interesa ahora comparar el consumo de DCFV con el de otros esquemas actuales basados en descomposición. De entre la gran cantidad de trabajo existente, seleccionamos BV [105] y DCFL [97] como los más relevantes. BV presenta mínimas complejidades de agregación y actualización, si bien tiene baja escalabilidad con N. DCFL, en tanto, representa actualmente uno de los esquemas más escalables con el número de reglas N, a costa de desempeño reducido (esquema original) o alto consumo de memoria (versión DCFLE). Ambos han sido ampliamente adoptados en propuestas recientes; podemos citar [118] en la línea de BV y [83] en el caso de DCFL. En el caso de BV, se consideran dos etapas implementadas mediante TCAM emulada de ancho $|UV_C|$, mientras que la etapa de agregación consta de simples operaciones AND por lo que no suma al consumo. Para DCFV, se consideran dos etapas de lookup mediante TCAM emulada con anchos V_A y V_B respectivamente, junto con el consumo de registros del array sistólico de agregación. Para el caso de DCFL, finalmente, se consideran dos bloques de memoria similares a los utilizados para emular TCAM pero de anchos $5.log_2|UV_A|$ y $5.log_2|UV_B|$ respectivamente; esto es de hecho un DCFL "paralelizado" (similar al caso DCFLE) para compararlo en igualdad de condiciones con BV y DCFV. DCFL suma también el consumo de las 5 respectivas memorias de agregación, de profundidad $2^{(log_2|UV_A|+log_2|UV_A|)}$ y ancho $log_2|UV_C|$. Como antes, consideramos casos de $|UV_A| = cte$, $|UV_B| = cte$ en la Fig. 4.24(a), mientras que $|UV_C| = cte$ en la Fig. 4.24(b). Cabe mencionar que las etapas de lookup se han optimizado respecto a lo visto en el Cap. 3 mediante la consideración de $|UR_A| = 2|UV_A| - 1$ y $|UR_B| = 2|UV_B| - 1$ para los dos campos respectivamente, esto se amplía en el Cap. 5.

De las Figs. 4.24(a) y 4.24(b) se pueden extraer las siguientes observaciones. BV comparte con DCFV el uso de bitmaps para simplificar la agregación y actualización. Sin embargo, BV lleva esta simplicidad a un extremo a costa de ser completamente incapaz de explotar las diferencias de tamaño entre $|UV_A|$, $|UV_B|$ y $|UV_C|$. DCFL, en el extremo opuesto, explota etiquetas de anchos óptimos $log_2|UV_A|$, $log_2|UV_B|$ y $log_2|UV_C|$ respectivamente a costa de ser completamente incapaz de representar más de un UV por etiqueta, debiendo utilizar múltiples accesos secuenciales o replicación exhaustiva de bloques de memoria. Nuestra arquitectura, en cambio, resigna consumo adicional moderado de memoria respecto a DCFL a cambio de ser capaz de implementar clasificación multi-match en un ciclo como lo hace BV. Finalmente, cabe destacar que DCFV no es dependiente del máximo número de UVs *coincidentes* como lo es DCFL, por lo que el límite máximo de UVs que pueden

coincidir a la vez está dado en DCFV por los anchos $|UV_A|$, $|UV_B|$ y $|UV_C|$ respectivamente. Este límite, mucho más holgado que aquél impuesto en DCFL, permite soportar patrones de solapamiento mucho más generales que DCFL, el cual se basa fuertemente en observaciones sobre rulesets reales.

Como segundo análisis se considera el consumo de memoria para las versiones de PE basadas en memoria (DCFV-mem), lógica (DCFV-logic) e híbrida (DCFV-hybrid); nuevamente contra BV y DCFL. Para DCFL se consideran dos versiones, una puramente secuencial (DCFL) y otra concurrente para un máximo de 5 coincidencias por campo (DCFLE-5) tal como lo propone [83]; esta última requiere 5 instancias idénticas de DCFL. En la Fig. 4.25(a) se observa el consumo de memoria para los cuatro esquemas en un caso de agregación 2D, considerando $|UV_A| = |UV_B| = 0{,}5|UV_C|$ para $|UV_C|$ variable. DCFL requiere $5 \times 5 = 25$ ciclos de reloj para obtener las múltiples coincidencias, mientras que BV, DCFLE-5 y DCFV lo hacen en un ciclo. DCFL presenta mínimo consumo de memoria, en tanto que su contraparte concurrente DCFLE-5 muestra máximo consumo de este recurso. BV, en tanto, presenta consumo de memoria algo mayor que DCFL para estos casos de $|UV_C|$ moderado, sin embargo se debe notar que crece linealmente con $|UV_C|$. DCFV-mem se ubica aún en un punto medio entre DCFLE-5 y BV, lo que representa consumo bastante elevado. Buscando mejorar este aspecto, se evalúan en la Fig. 4.25(b) las implementaciones basadas en lógica e híbrida. DCFV-logic presenta mínimo consumo de memoria ya que realiza todo el procesamiento de agregación en lógica combinacional, pero como veremos no es escalable en cuanto a velocidad en implementaciones reales. DCFV-hybrid, en cambio, resigna un moderado consumo adicional a cambio de mejor escalabilidad de implementación, ubicándose aun así muy por debajo de los requerimientos de DCFV-mem. Cabe destacar que el almacenamiento en DCFV-logic y DCFV-mem consiste en bancos de registros, mientras que el almacenamiento en DCFV-mem consiste en memoria RAM como se indica en la Fig. 4.25(b).

4.6.2. Resultados de síntesis en FPGAs

A fin de comprobar el desempeño de las arquitecturas planteadas, se realizaron implementaciones para FPGAs Altera Stratix V de las que se reportan resultados post-fitting. El dispositivo utilizado es el modelo 5SGXMB5RF43C2, de características intermedias dentro de la familia Stratix V, cuyos recursos de interés son 370000 LUTs, 740000 registros, y 2100 BRAMs de 20 Kbits cada una, configurables éstas en diferentes modos $profundidad$x$ancho$ desde 512x40 hasta 20Kx1. La evaluación se centra en la etapa de agregación, mientras que las etapas de lookup se evalúan en el Cap. 5.

Los primeros resultados de síntesis, obtenidos mediante arrays semi-sistólicos 2-d 1-b y 2d-2b utilizando PEs implementados mediante tablas de verdad, se muestran en los Cuadros 4.1 y 4.2 respectivamente. En estos cuadros, se reportan resultados para diferentes combinaciones de $|UV_A|$, $|UV_B|$ y $|UV_C|$, indicadas como $v_A\_v_B\_v_C$. Estos resultados confirman que una granularidad doble efectivamente reduce el consumo de registros a la mitad. El consumo de LUTs es incluso reducido, ya que la versión de granularidad fina utiliza las LUTs más ineficientemente mientras que las versiones de granularidad gruesa permiten a la herramienta de síntesis optimizar este consumo. El desempeño de ambas se mantuvo por encima del requerido por redes de 100Gbps para todos los casos; sin embargo comienza a decrecer rápidamente para casos más grandes que el máximo 32_32_64 aquí considerado.

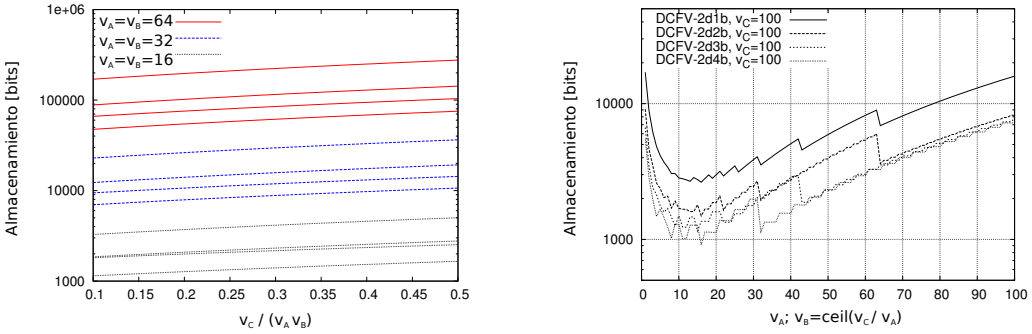
Cabe destacar que los factores $|UV_A|$ y $|UV_B|$ afectan mucho más el consumo que $|UV_C|$.

En segundo término, buscando mejores características de escalabilidad, se implementan y evalúan casos de PE-logic (descripción de alto nivel) y PE-hybrid (combinación de regfile y lógica), mientras que PE-mem se descarta por su gran consumo de memoria. Por simplicidad, las granularidades se expresan abreviadamente como *a by b* mientras que las relaciones del producto cartesiano con el producto cruzado se indican como *(a · b) to c*, resultando la nomenclatura abreviada a*by*b_*cto*c. En general, *a* afectará la latencia del pipeline mientras que *b* afectará la etapa de agregación de filas. En el Cuadro 4.3 se muestra que PE-logic sufre abrupta caída de desempeño más allá del caso $2by2\_4to4$ utilizado en los ensayos anteriores. Se evalúa entonces una implementación híbrida a fin de buscar el mejor compromiso de lógica y memoria. Como se muestra en el Cuadro 4.4, esta versión tiene mayor flexibilidad para implementar granularidades y simetrías sin perder desempeño. El costo adicional en registros es debido al bloque *rules*, pasando de $a \cdot b$ a $c \cdot log_2(a \cdot b)$. En particular, se ensayaron granularidades de hasta $g = 64$ (8by8) manteniendo velocidades en el orden de las requeridas por redes 100Gpbs.

Del análisis de los casos $2by8\_16to8$ y $4by4\_16to8$ podría parecer que la versión donde $a < b$ es más conveniente, ya que presenta mayor velocidad y requiere menos etapas de agregación de salida $|UV_C|/c$. Sin embargo, debe tenerse en cuenta que, dado un determinado $|UV_C|$, el valor de c debe ajustarse tal que $c = |UV_C| \cdot b/|UV_B|$, por lo que al duplicar b, c también debe hacerlo si $|UV_C|$ y $|UV_B|$ son constantes; esto significa que la complejidad del PE debe en realidad aumentar de $4by4\_16to8$ a $2by8\_16to16$ con desempeño peor que su caso simétrico. Por otro lado, se observa que la relación costo-beneficio de aumentar g comienza a decrecer más allá del caso $4by4$; por ello se adopta el esquema $4by4\_18to8$ en nuestro posterior ensayo.

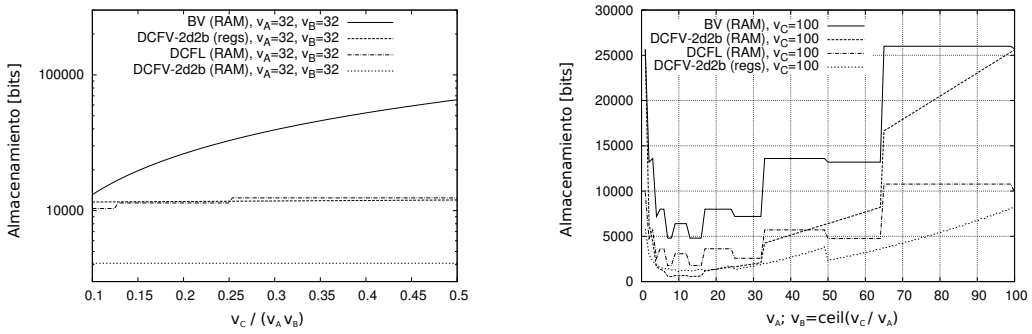
En el Cuadro 4.5 se muestran los resultados de aplicar el PE adoptado en un array 2-d totalmente sistólico. Como se observa, es posible soportar tamaños mayores que en el caso de PE-logic manteniendo desempeño satisfactorio.

Finalmente, se busca realizar una evaluación extendida considerando casos más grandes y generales. Al separar las distintas etapas que forman nuestra arquitectura de agregación, se detecta que la mayor limitación de velocidad proviene de los pipelines de entrada y de salida. Ya que la principal innovación de nuestra arquitectura reside en el array de PEs, resulta de interés individualizar este array y observar sus características. En el Cuadro 4.6 se muestran los resultados post-fitting de distintas configuraciones de este array, en ellas se asume, como en los análisis anteriores, que $|UV_C| = 2 \cdot max\{|UV_A|, |UV_B|\}$. Así, por ejemplo, para agregación de dos campos con $|UV_A| = 16$ y $|UV_B| = 256$ se asume que $v_C = 2 \cdot max\{16, 256\} = 512$. Las Figs. 4.26(a), 4.26(b) y 4.26(c), en tanto, muestran los consumos y velocidades resultantes de las combinaciones de anchos considerados. En general, se observa que se puede obtener una gran variedad de combinaciones a velocidades satisfactorias con consumos moderados. Cabe mencionar que, en este cuadro y figuras, se denotan los anchos $|UV_A|$, $|UV_B|$ y $|UV_C|$ como $WIDTH\_A$, $WIDTH\_B$ y $WIDTH\_C$ respectivamente. Estos resultados sirven de base para seleccionar las configuraciones más adecuadas a ciertos casos de aplicación con requerimientos concretos de anchos y velocidad de operación.



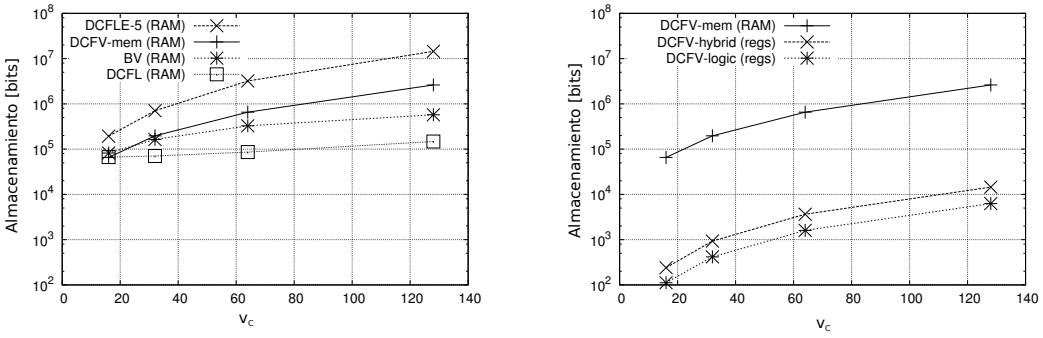
(a) (b)

Figura 4.23: Consumo de recursos: (a) $|UV_C|$ variable, (b) $|UV_A|$ y $|UV_B|$ variables



(a) (b)

Figura 4.24: Consumo de BV vs. DCFV vs. DCFL: (a) $|UV_C|$ variable, (b) $|UV_A|$ y $|UV_B|$ variables



(a) (b)

Figura 4.25: Esquema propuesto: (a) DCFV-mem vs. esquemas previos, (b) opciones de implementación de DCFV

4.7. Conclusiones

En este Capítulo, se propone una nueva arquitectura de clasificación multi-dimensional con soporte natural multi-match, baja complejidad de actualización y capaz de procesar paquetes a velocidad de línea. En particular, esta arquitectura implementa agregación de resultados de lookup en un

Cuadro 4.1: Arquitectura 2-d 1-b semi-sistólica

v_A_v_B_v_C	Mpps	Registros	LUTs
8_8_16	717	588	450
8_8_32	735	732	450
16_16_32	545	3352	1314
16_16_64	528	3896	1314
32_32_64	386	21552	4262

Cuadro 4.2: Arquitectura 2-d 2-b semi-sistólica

v_A_v_B_v_C	Mpps	Registros	LUTs
8_8_16	704	254	62
8_8_32	714	356	82
16_16_32	717	1570	282
16_16_64	656	1896	322
32_32_64	493	10442	1202

Cuadro 4.3: PE basado en lógica (impl. alto nivel)

Geometría	LUTs	Registros	Mpps
2by2_4to4	27	19	717
2by8_16to8	368	47	80
4by4_16to8	368	45	76

Cuadro 4.4: PE híbrido

Geometría	LUTs	Registros	Mpps
2by2_4to4	28	23	717
2by8_16to8	133	62	512
4by4_16to8	149	60	480
8by2_16to8	149	62	446
2by8_16to16	280	111	400
4by8_32to16	390	129	320
6by6_36to16	497	145	288
8by8_64to16	630	149	280

esquema de clasificación por descomposición, el que es especialmente adecuado para implementación en plataformas hardware. Esta propuesta es el resultado de un exhaustivo análisis y comparación de las alternativas actuales para clasificación por descomposición, a partir del cual se elaboró una nueva taxonomía basada en los metadatos propagados dentro de la arquitectura.

El esquema propuesto es concebido buscando combinar las ventajas de los trabajos previos, atacando asimismo sus inconvenientes. En particular, se busca explotar *tanto* los recursos de lógi-

Cuadro 4.5: Array de agregación de PEs híbridos 4by4_16to8

v_A_v_B_v_C	LUTs	Registros	Mpps
16_16_32	2568	1016	395
32_16_64	4952	1952	330
16_32_64	5122	2178	321
32_32_64	10058	3870	320
32_64_128	20663	8132	300
64_32_128	19957	7344	270
64_64_128	40238	14830	260

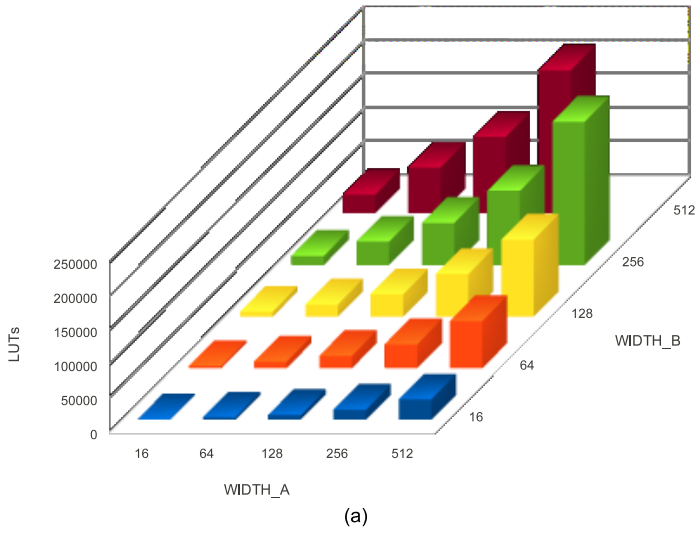
ca combinacional *como* los recursos de memoria que brindan actualmente los dispositivos FPGA, evaluando los compromisos de diseño resultantes.

Los resultados de implementación en FPGAs demuestran que nuestra propuesta es capaz de soportar tasas de transferencia de datos en el orden de 100Gbps para paquetes mínimos de 40 bytes, manteniendo un compromiso conveniente en consumo de recursos. Esto permite, por ejemplo, su integración a sistemas de red donde debe co-existir con otros módulos de procesamiento de datos. En particular, se demuestra cómo nuestro esquema logra reducir el consumo de memoria de propuestas previas migrando ciertas funciones a lógica combinacional.

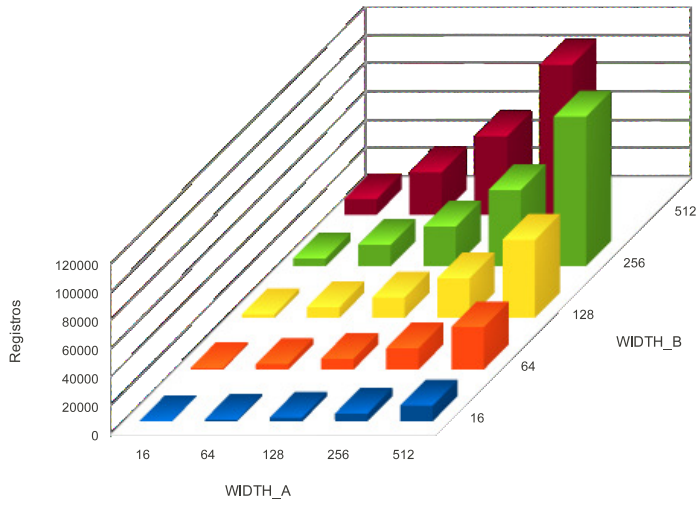
El trabajo reportado en el presente Capítulo fue plasmado en la cuarta contribución de la Tesis de Doctorado, titulado *Multi-match packet classification on memory-logic trade-off FPGA-based architecture*. Esta contribución fue presentada y publicada en la IEEE 14th. International Conference on High Performance Switching and Routing HPSR 2013.

Cuadro 4.6: Array de agregación sin pipelines de E/S: casos extendidos

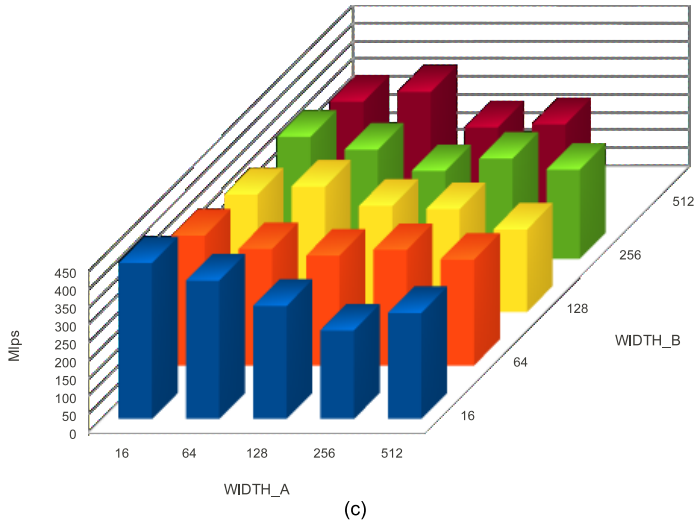
$WIDTH\_A$	$WIDTH\_B$														
	16			64			128			256			512		
	LUTs	Regs	Mlps	LUTs	Regs	Mlps	LUTs	Regs	Mlps	LUTs	Regs	Mlps	LUTs	Regs	Mlps
16	1299	471	438	3865	1527	365	7836	2935	331	14281	5751	343	28193	11383	292
64	4087	1527	389	10284	4311	328	18605	8023	353	35100	15447	307	67938	30295	319
128	7911	2935	318	18908	8023	309	33629	14807	299	62830	28375	247	113257	55511	220
256	15431	5751	249	36028	15447	325	63454	28375	290	109750	54231	282	211274	105943	228
512	30408	11383	298	70062	30295	298	113928	55511	233	211719	105943	251	N/A	N/A	N/A



(a)



(b)



(c)

Figura 4.26: Array de agregación sin pipelines de E/S: (a) consumo de LUTs, (b) consumo de registros, (c) velocidad de procesamiento

Capítulo 5

Esquemas de lookup

5.1. Motivación

En el Cap. 3 se evaluaron arquitecturas de emulación de TCAM, que permiten implementar clasificación general (uni- o multi-dimensional) a altas velocidades, con escalabilidad limitada por su crecimiento lineal con la cantidad de reglas N. En el Cap. 4, en tanto, se analizaron esquemas existentes para optimización de clasificación multi-dimensional en base a propiedades de los rulesets reales. Sobre esta base, se propuso una nueva arquitectura para clasificación multi-campo especialmente diseñada para explotar equitativamente los recursos disponibles en un FPGA. En particular, se adopta un esquema de clasificación por descomposición, el cual consta de dos etapas bien diferenciadas de *búsqueda (lookup)* en cada campo y de *agregación* de los resultados de búsqueda obtenidos. En al Cap. 4 se abordó la etapa de agregación, mientras que este capítulo ataca el problema de lookup[1].

Los FPGAs son plataformas muy adecuadas para la implementación de funciones de clasificación basada en flujos en los esquemas de red actuales. Como se demuestra en el Cap. 4, los esquemas de *clasificación por descomposición* son especialmente adecuados para implementación concurrente en estos FPGAs; estos esquemas constan en general de una primera etapa de *lookup* en cada campo, seguida se una segunda etapa de *agregación* de los resultados individuales. Sin embargo, los esquemas de lookup se han desarrollado hasta el momento en forma independiente de los esquemas de agregación, en muchos casos orientados a diferentes plataformas de implementación, lo que frecuentemente dificulta su óptima integración con la etapa de agregación en FPGAs. En este contexto de *lookup aplicado a clasificación por agregación*, y extendiendo el análisis del Cap. 4, se propone en este capítulo una nueva interpretación del problema de lookup que permite abarcar las propuestas existentes para compararlas directamente. Sobre esta base, se establecen casos generales

[1] En este capítulo adoptaremos el término *lookup* por considerarlo específico al problema tratado, a diferencia del traducción al español *búsqueda* el cual es más general.

que son comparados e implementados en FPGAs, reconociendo así oportunidades de optimización. Como característica particular, los módulos evaluados son de simple implementación y pueden ser fácilmente adaptados para su óptima combinación con distintos esquemas de agregación. A fin de determinar estas combinaciones, se proveen curvas comparativas y resultados de síntesis para distintas configuraciones. Este enfoque totalmente general no ha sido anteriormente abordado.

Como segundo aporte de este capítulo, se proponen mejoras para el esquema que, según el análisis efectuado, resulta más apropiado para clasificación por flujos. En particular, se excluyeron de nuestra evaluación los esquemas basados en hashing concentrándose en técnicas determinísticas, ya que se considera en estas funciones es fundamental ofrecer garantía de desempeño para casos generales.

5.2. Técnicas de lookup uni-dimensional

Como ya se mencionó, la técnica básica para realizar lookup MM independiente del ruleset es la búsqueda lineal; los array asociativos, en particular las TCAMs, permiten realizar esto a muy altas velocidades. Sin embargo, las TCAMs presentan inconvenientes en cuanto a consumo de energía, costo y el efecto de *expansión de rango*, los cuales son mitigados mediante técnicas de habilitación selectiva [72] [125] o pre-codificación de rangos [75]. Como se demuestra en el Cap. 3, su emulación en plataformas FPGA permite mitigar estos problemas teniendo la precaución de obtener el correcto balance respecto a la expansión de direccionamiento.

El problema particular de lookup por prefijos para ruteo de paquetes en redes IP según el esquema de *ruteo inter-dominio sin división por clases (Classless Inter-Domain Routing, CIDR)*, de importancia fundamental para el funcionamiento de la internet, ha sido extensivamente desarrollado y existen numerosas soluciones eficientes sobre diversas tecnologías [88] [126] [127] [128] [129]. El problema más general de rangos arbitrarios, en tanto, ha adquirido relevancia en los últimos tiempos debido al surgimiento de protocolos con funcionalidades más allá del ruteo IP. Una importante limitación a la comprensión y solución de este problema surge de propuestas muy relacionadas a la plataforma específica de implementación y la falta de comparación objetiva entre las propuestas existentes.

5.2.1. Esquemas basados en árboles

Si bien las TCAMs pueden ser aplicadas tanto a clasificación uni-dimensional (lookup) como multi-dimensional (clasificación), en este capítulo nos centraremos en su aplicación uni-dimensional, es decir en un campo del encabezado de un paquete. Para una TCAM general de profundidad N, los puntos de interfaz entre lookup y agregación pueden identificarse como los límites de implementación de rangos arbitrarios. Este concepto permite independizarse del formato de los metadatos propagados, y se aclarará más adelante. Suponiendo un campo de ancho M, la TCAM compara estos M bits contra las N reglas concurrentemente; este completo paralelismo que la hace independiente del formato de reglas es también la principal causa de sus inconvenientes. Para atacar dichos incon-

venientes, surgen técnicas *algorítmicas* de lookup, las cuales *reducen incrementalmente el espacio de lookup* sin disminuir excesivamente el desempeño. Estos esquemas se pueden llevar a hardware eficientemente utilizando memorias SRAM o DRAM, mucho más económicas y escalables que la TCAM, y aplicando técnicas de pipelining. Una de estas técnicas se basa en el uso de *tries*, árboles especiales donde el camino entre la raíz y las hojas se determina por el estado de los *bits sucesivos* del campo o key. Estos bits pueden procesarse de a uno o en grupos llamados *strides*, dando lugar a los *multi-bit tries* con distintos compromisos de consumo vs. velocidad. El uso de strides *comprime* el trie con criterios tales como agrupación de múltiples nodos con sólo una rama o hijo (path compression), agrupación de múltiples niveles con ramas completas (level compression), o múltiples hojas que llevan a la misma decisión (Lulea) [55] [87]; sin embargo estas técnicas no satisfacen los requerimientos actuales de actualización dinámica. Una combinación de ellos, con actualización más simple, se propone en el esquema Tree Bitmap [55]. Otras propuestas más recientes mantienen estas propiedades, orientándose al soporte efectivo de IPv6; para ello recurren a técnicas de hashing [130] [127]. Si bien estos esquemas trabajan bien para el caso de direcciones IP, están diseñadas en base a un esquema de lookup por prefijos, por lo que su comportamiento para casos de rangos arbitrarios es incierto. Finalmente, se debe mencionar que muchos de ellos están sujetos a patentes, lo que dificulta su adopción en investigación [87].

Los *árboles binarios de búsqueda* son otra opción desarrollada para lookup IP, donde el camino de búsqueda se define en base a criterios más generales que seguir los valores de bits consecutivos. El costo de esta mayor abstracción es la necesidad de contar con comparadores de magnitud (no sólo de igualdad) de ancho M así como memoria de ancho M que almacena los límites involucrados en cada nivel del árbol. Una variante de este esquema general se orienta a prefijos, agrupando reglas y tomando decisiones de camino según las longitudes de máscara IP. Esto simplifica la búsqueda ya que, por ejemplo, para IPv4 sólo existen 33 posibles longitudes de prefijo (considerando wildcards). De esta forma, la búsqueda en cada etapa se efectúa sobre prefijos de longitud conocida mediante técnicas de hashing [131]. Este esquema es atractivo para rulesets que contienen muchas reglas sobre un key ancho, pero que en realidad comparten pocas longitudes de prefijo; por otro lado, el uso de hashing hace difícil predecir su comportamiento para casos desfavorables. Otra variante de los árboles binarios de búsqueda es *búsqueda binaria sobre rangos* o *árboles de rangos*. Este esquema toma reglas basadas en prefijos, que en general presentan solapamientos totales (ver Cap. 4), generando nuevos intervalos sin solapamiento que se almacenan en la memoria del árbol de búsqueda. El lookup se efectúa comparando el key contra los límites de estos intervalos, entregando el LPM asociado al intervalo resultante. Si bien los árboles de rangos son originalmente más lentos que los tries multibit, han generado abundante investigación [132] [133]. De las técnicas basadas en árboles, esta variante parece la más apropiada para implementación de rangos arbitrarios, sin embargo no existe un estudio completo sobre su comportamiento en este caso de lookup. En el presente capítulo se abordará este análisis, entre otros.

Las esquemas basados en árboles han sido implementados tanto en software como en hardware; en este último caso se explotan técnicas de pipelining y la memoria de reglas se distribuye en bloques menores locales a cada nivel del árbol, logrando arquitecturas de alto desempeño. Sin embargo, las últimas etapas del árbol (hojas) requieren bloques de memoria más grandes que las primeras (raíz), generando gran desbalance de memoria entre ambos extremos del pipeline. Esto resulta en

aprovechamiento ineficiente de la memoria, lo que lleva a limitada escalabilidad para rulesets grandes. Además, ya que el desempeño depende del camino crítico, definido por la etapa más lenta del pipeline, el desbalance de memoria se traduce en reducción del desempeño de todo el esquema. Las propuestas recientes atacan el problema de desbalance de memoria explotando los recursos de los FPGAs [133]. En estos esquemas se utilizan combinaciones de SRAM embebida para bloques pequeños y DRAM externa para niveles que exigen mayor volumen de almacenamiento. De este modo, son capaces en el mejor caso de soportar rulesets de hasta 256K prefijos IPv6 a velocidades de hasta 400 mega lookups por segundo (Mlps); sin embargo el uso específico de memoria DRAM externa exige interfaces mucho más complejas y lentas que las utilizadas en la SRAM presente en FPGAs.

Dos optimizaciones comunmente aplicadas a los árboles de búsqueda son las técnicas de *leaf pushing* y *node grouping*. En general, cada nodo en un árbol contiene información tanto sobre las ramas a seguir (punteros a memoria del siguiente nivel), así como sobre el resultado de lookup en caso de que ese nodo sea una hoja. Esto es debido a que las hojas del árbol se pueden encontrar en cualquier nivel intermedio dependiendo de las longitudes de prefijo o amplitudes de rangos. La técnica de leaf pushing mapea todas las hojas exclusivamente en el último nivel del árbol, separando estos nodos de los demás nodos intermedios que contienen sólo información sobre decisiones de ramificación; y optimizando de este modo el consumo de memoria. Node grouping, en tanto, agrega múltiples nodos en uno solo que concentra sus decisiones de ramificación. El consumo de memoria de estos nodos es lógicamente mayor, sin embargo el árbol requiere menos niveles resultando en beneficios en cuanto a desempeño. La efectividad de ambas optimizaciones es, sin embargo, muy dependiente de las características del ruleset particular.

5.2.2. Esquemas basados en TCAM emulada

En años recientes, se ha registrado creciente interés en arquitecturas de *emulación de TCAM* para implementaciones FPGA. El análisis de arquitecturas de emulación basadas en RAM es objeto del Cap. 3 de esta Tesis; las FPGAs ofrecen asimismo otros recursos lógicos que pueden ser explotados con este fin. Además, las FPGAs permiten explorar versiones modificadas de una TCAM, permitiendo por ejemplo explorar distintos esquemas de pipelining, extraer directamente el vector de coincidencias (matches) o una versión codificada ad-hoc de éste. De este modo, la contribución [135], asociada al Cap. 3 de esta Tesis, queda enmarcada dentro de una línea de investigación más amplia que será revisada a continuación. Cabe aclarar que, como se comentó en al Cap. 3, estas arquitecturas utilizan invariablemente un vector de matches de ancho N por lo que se pueden utilizar para lookup en un campo o, mediante simple extensión del key, a múltiples campos. En este capítulo nos concentraremos en su aplicación a un campo, planteando optimizaciones para este caso particular.

Los primeros trabajos de investigación sobre emulación de TCAM buscaban el solo objetivo de implementar TCAM en FPGAs sin considerar las posibilidades particulares que introduce la implementación en FPGAs. Los recursos utilizados para emulación eran LUTs utilizadas como registros de desplazamiento (SRL16E) y LUTs utilizadas como memoria RAM, ambas características propias de FPGAs Xilinx, y por tanto no generales a todos los dispositivos FPGA. Al incorporar estos disposi-

tivos mayores recursos de memoria BRAM, y ser esta característica común a los distintos fabricantes, se consideró esta alternativa como más conveniente. El primer trabajo de investigación en explorar una combinación de TCAM con otras técnicas de lookup para aliviar el problema de expansión de rangos fue *BV-TCAM* [136]. En ese trabajo, se plantea una arquitectura multi-dimensional donde los campos basados en rangos se resuelven mediante tree bitmaps [137], mientras que los campos en valores exactos o prefijos se resuelven mediante TCAM emulada con primitivas SRL16E basadas en LUTs [138]. Sin embargo, no se explora resolución de rangos en la propia TCAM emulada. La propuesta llamada *Field-Split Bit Vector* [139], en tanto, explora la combinación de TCAM nativa (externa al FPGA) y emulada dentro del FPGA; sin embargo también aquí se utiliza la emulación como un reemplazo ineficiente de la TCAM nativa sin analizar sus características propias. *Packet Classification with Incremental Update capability (PCIU)* [140] presenta una implementación y evaluación más completa de TCAM emulada, considerando plataformas software y hardware. *StrideBV* [141] es una propuesta muy similar que comienza a considerar mejoras en la eficiencia de implementación mediante el uso de diferentes factores de forma en BRAM. Con base en estos antecedentes y considerando la madurez de la tecnología de FPGAs, nuestro trabajo ([135] y Cap. 3) explora exhaustivamente el potencial de la emulación basada en BRAM y, aun más, evalúa la implementación de rangos arbitrarios en esta plataforma a través del control de expansiones rango/direccionamiento. En trabajo posterior [142], se comparan versiones de StrideBV implementadas en RAM distribuida (distRAM) y BRAM [141] contra TCAM emulada en SRL16E [138]. El desempeño y costo de estas implementaciones dependen fuertemente de las características de las primitivas SRL16E y distRAM, por lo que no son portables. Una propuesta reciente [117] combina BRAM para el caso de rangos amplios, y distRAM para mejorar la eficiencia de memoria en rangos pequeños. Esta propuesta es la primera en adoptar las ideas de ETCAM [72] para el caso de FPGAs bajo la denominación de *comparación explícita de rangos (Explicit Range Match, ERM)* y explotarlas para rangos amplios en lugar de utilizar BRAM. ERM almacena los extremos de rango en registros de propósito general, utilizando comparadores de magnitud cada rango almacenado. Durante el lookup, se accede a todos estos registros concurrentemente y se comparan contra el key. Esta implementación hace intensivo uso de registros y comparadores, lo que limita su escalabilidad y produce alto consumo de energía. Finalmente, la propuesta denominada *Strided and Clustered Bit Vector (SCBV)* [119], integra las propuestas previas en una arquitectura de pipeline sistólico, concentrándose luego en esquemas eficientes para su actualización dinámica. Utilizando distRAM y ERM, este esquema alcanza velocidad de 100Gbps para un ruleset OpenFlow de 15 campos y 1000 reglas. Sin embargo, si bien se reporta implementación de ERM mediante distRAM, lo que mitigaría el problema de consumo de registros de [117], no se aclara cómo se logra el acceso completamente concurrente a distRAM. Efectivamente, tal implementación mediante distRAM no parece una opción válida. Una implementación alternativa de ERM, que efectivamente utiliza distRAM, se halla en [83], sin embargo en este caso el principio de funcionamiento es diferente al planteado y sus beneficios son muy modestos, como se demostrará en este capítulo.

Al momento, el esquema ERM se presenta como un medio efectivo y simple de solucionar el problema de expansión de rangos en TCAM emulada, tal como lo hace ETCAM con respecto a la TCAM nativa; por ello es conveniente analizar sus compromisos de implementación. Considerando dos límites de rango para N reglas definidas sobre un campo de M bits, ERM requiere acceso concur-

rente a $2MN$ registros independientes; estos registros deben ser interconectados a sus comparadores asociados y debe accederse a ellos para cada operación de lookup. De este modo, si bien este esquema resuelve el problema de expansión de rangos de las TCAMs, repite sus demás compromisos de diseño: ambos obtienen lookup a velocidad de línea con mínimos requerimientos de almacenamiento, mientras que sus necesidades de interconexión y granularidad producen alto consumo energético y limitada escalabilidad tanto con M como con N. Los retardos de interconexión son efectivamente mitigados mediante pipelining en FPGAs, sin embargo los beneficios del pipelining disminuyen para rulesets grandes y aumentan considerablemente el consumo de registros. Finalmente, cabe mencionar que las características de ERM pueden ser implementadas mediante ETCAM con mejor desempeño, por lo que no es un beneficio propio de la implementación en FPGAs. En este capítulo se explorarán en profundidad estos conceptos, así como soluciones alternativas al problema de expansión de rangos.

Un aspecto importante, común a todas las arquitecturas basadas en TCAMs, es que ellas asumen *lookups independientes* sobre cada una de las N reglas kD, aun cuando se trate de lookups en un campo (1D). Como ya se mencionó, esto permite utilizar TCAMs en 1D o kD por simple concatenación de resultados, sin embargo también fuerza su escalabilidad a $O(N)$. Las TCAMs se pueden utilizar directamente como esquemas de lookup en un esquema de clasificación por descomposición utilizando bitmaps de tamaño optimizado como metadato; en este caso el ancho del vector de salida para el campo d ($d = \{1, ..., k\}$) se puede reducir a $O(|UV_d|)$. Su adaptación a un esquema de clasificación por descomposición utilizando metadatos basados en labels, en cambio, requeriría una etapa de codificación adicional, que en el caso de lookup MM puede resultar muy ineficiente. Existen sin embargo otros esquemas de lookup que producen directamente metadatos basados en etiquetas y que en este caso pueden ser más eficientes que aquéllos basados en TCAM.

5.2.3. Definiciones generales

En líneas generales, diferenciaremos en este capítulo:

- técnicas de lookup en 1D o en kD: TCAM (kD, ancho N) vs. todos los demás (1D)
- desde el punto de vista del mecanismo de búsqueda: lookup basado en búsqueda independiente de reglas (o por UVs) vs. lookup basado en búsqueda conjunta de regiones (o por URs)
- desde el punto de vista del metadato de salida: metadato basado en bitmaps (bits independientes sin pre-procesamiento) vs. metadato basado en etiquetas (bits inter-dependientes generados por pre-procesamiento)
- desde el punto de vista de la semántica del metadato de salida: basado en URs o en UVs
- desde el punto de vista de la implementación: basada en lógica, en memoria, o combinación de ellas

Considerando las combinaciones de estas opciones, se pueden cubrir los casos de lookup existentes y se logra identificar nuevas optimizaciones.

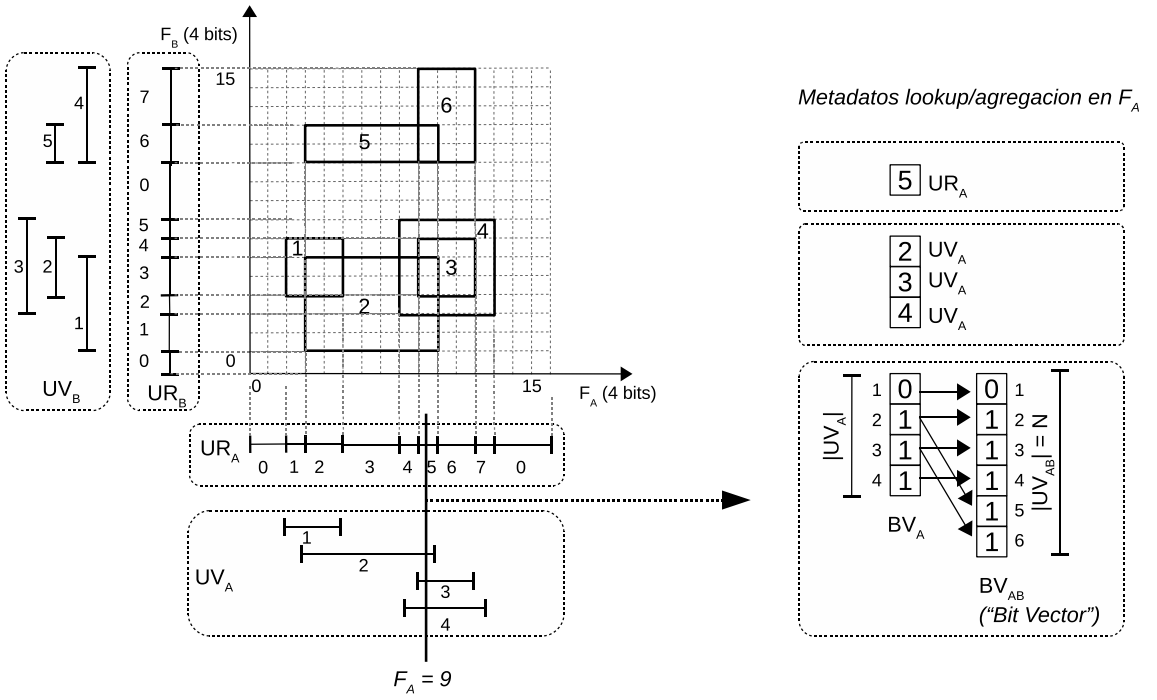


Figura 5.1: Tipos de metadatos en la interfaz lookup-agregación

En la Fig. 5.1 se hace referencia al ruleset de ejemplo ya discutido en la Fig. 4.4, haciendo énfasis esta vez en la búsqueda 1D o lookup. Para cada campo, el lookup puede representar los resultados de búsqueda mediante distintos *metadatos*, que pueden ser del tipo bitmap o etiqueta como se ilustró en la Fig. 4.5. Las etiquetas pueden representar una UR, mediante un identificador *URID* asociado; o determinados UVs, mediante sus respectivos identificadores *UVID*. Tanto los URIDs como los UVIDs son en general asignados según el criterio del algoritmo de pre-procesamiento. Para un determinado valor de campo, el URID resultante es sólo uno ya que las regiones son mutuamente exclusivas, sin embargo los UV no lo son por lo que los UVIDs devueltos pueden ser múltiples. Los posibles bitmaps tienen en general ancho dependiente de la dimensión considerada $|UV_i|(i = 1, ..., k)$, en cuyo caso son locales a su dimensión de pertenencia. Dadas dos dimensiones A y B de valores asociados $|UV_A|$ y $|UV_B|$, sus bitmamps respectivos pueden también *expandirse* al ancho de su bitmap agregado $|UV_{AB}|$ a fin de concatenar resultados de lookup mediante simple intersección. También se han propuesto bitmaps donde cada bit representa una región con el fin de representar resultados MM mediante TCAMs nativas [65]; sin embargo éstos no tienen sentido para implementaciones en FPGA ya que las regiones son mutuamente exclusivas, es decir sólo un bit se activa a la vez. En la Fig. 5.1 se ilustran estos formatos para el caso de lookup en un campo A con un key $F_A = 9$. Se puede observar que todos estos formatos representan en esencia una determinada UR; la etiqueta URID lo hace con mínimo ancho pero su generación requiere intensivo pre-cómputo, mientras que el bitmap de UVs en dD ($d = \{1, ..., k\}$) es el más ineficiente en ancho pero no requiere pre-cómputo alguno. El bitmap BV es un caso aún más extremo ya que concentra información de las k dimensiones sin requerir pre-cómputo ni agregación en el sentido formal.

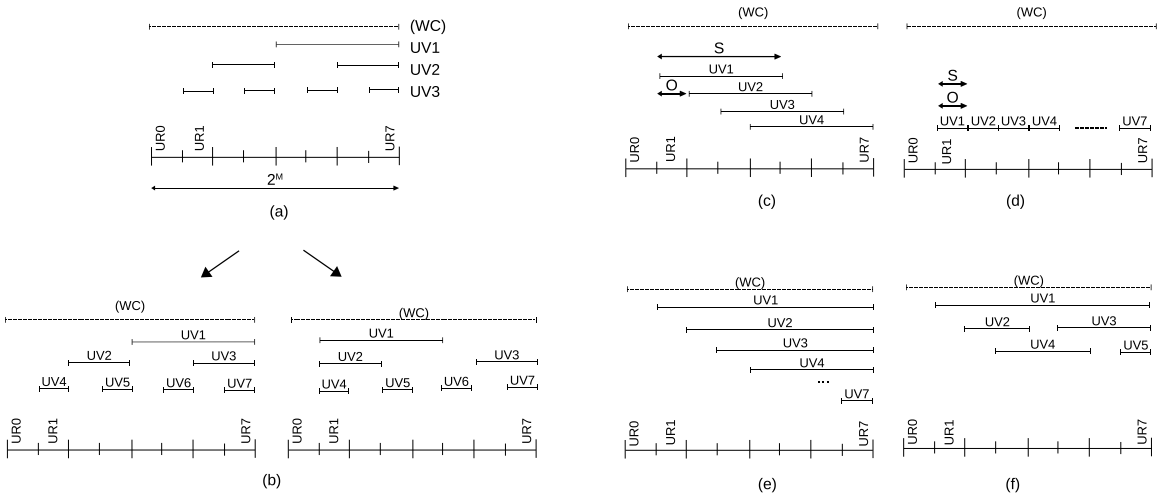


Figura 5.2: Peores casos de solapamiento: (a) caso teórico $|UR| = 2^M$, (b) peor caso para PX, (c) peor caso para AR, (d) peor caso para EX, (e) y (f) casos intermedios de GR

5.3. Análisis de requerimientos

5.3.1. Complejidades de lookup

A fin de comprender más profundamente el problema de lookup en un campo, y definir así las posibles contribuciones, nos preguntamos qué tan costosa puede llegar a ser esta operación. Para ello, consideraremos los límites en la cantidad de UVs y URs para los casos PX, EX y AR. En la Fig. 5.2 se muestra el espacio de búsqueda para un campo de ancho M y 2^M valores de campo. Consideramos asimismo $|UV|$ UVs en este campo, que eventualmente escalarán a N al combinarse con otros campos mediante agregación. Cada uno de estos UVs está asociado a un intervalo de valores de key, definido ya sea por *extremos de inicio (start) y fin (end)* $[s, e]|_{s \leq e}$, o sus respectivos *desplazamiento (Offset) e intervalo (Scope)* $(O, S)|_{O \geq 0, S \geq 0}$. Asimismo, existe un UV0, del tipo WC, que abarca todo el espacio de búsqueda; el objeto de este UV es definir una regla por defecto que se devuelve como resultado cuando ninguna de las demás reglas definidas coincide; las regiones donde sólo esta regla está presente definen asimismo una región por defecto UR0. Intuitivamente, podríamos decir que, para un determinado $|UV|$, pueden existir $2^{|UV|}$ URs como ilustra la Fig. 5.2(a), sin embargo este caso teórico implicaría UVs definidos por múltiples puntos $[s, e]$ lo cual no es posible en la práctica. Como se observa en los casos de la Fig. 5.2(b), este es en realidad un caso de $|UV|$ UVs basados en prefijos, por lo que $|UR| = |UV|$. Para AR, en tanto, el peor caso es $|UR| = 2|UV| - 1$ como se ilustra en la Fig. 5.2(c). El caso EX, en tanto, consiste esencialmente en prefijos de longitud máxima (scopes unitarios), por lo que vale asimismo el límite $|UR| = |UV|$ como muestra la Fig. 5.2(d). En las Figs. 5.2(e) y 5.2(f), finalmente, se muestran dos ejemplos de rulesets GR (PX, EX y AR combinados), en el primer caso se registra $|UR| = |UV|$ mientras que en el segundo se da $|UV| = 5$ y $|UR| = 7$ ($|UV| \leq |UR| \leq 2|UV| - 1$).

Desde el punto de vista opuesto, es de interés analizar cuántos UVs como mínimo ($|UV|_{min}$) son necesarios para obtener estos casos límite; para acotar el análisis consideramos el caso donde

todos los UVs comparten un mismo scope. El offset entre UVs para obtener $|UV|_{min}$ debe ser $O_{min} = 1$, es decir, se debe obtener una nueva UR para cada valor del key. Asimismo, el scope máximo para obtener ($|UV|_{min}$) es $S_{max} = 2^M/2$; para scopes mayores no es posible obtener un nuevo UR para cada key sin recurrir a scopes donde $e < s$ (scopes "circulares", no posibles en la práctica). En general, puede demostrarse que $|UV|_{min} = 2^M - S$. En la Fig. 5.2(c) se considera el caso (O_{min}, S_{max}) ($|UV|_{min} = 8 - 4 = 4$), mientras que la Fig. 5.2(d) muestra el caso opuesto (O_{min}, S_{min}) ($|UV|_{min} = 8 - 1 = 7$).

Como se observa en este análisis, el caso $|UR| = 2^{|UV|}$ no puede obtenerse con $|UV|$ UVs como podría pensarse, ya que estos UVs deberían ser discontinuos. Aun más, el peor caso para reglas basadas en prefijos es mucho más simple que para reglas basadas en rangos arbitrarios. Finalmente, se observa que este peor caso se da para condiciones muy especiales de ruleset, mientras que los rulesets generales se alejan bastante de él. Es de destacar que estas observaciones son puramente estadísticas, por lo que no dependen de un patrón determinado de reglas.

5.3.2. Requerimientos

Con el fin de comparar objetivamente nuestros esquemas de lookup, se analizarán y relacionarán sus requerimientos actuales en el contexto de un esquema de clasificación multi-dimensional por descomposición, ellos son:

1. actualización (update) dinámica simple y rápida
2. soporte igualmente eficiente de reglas generales, basadas tanto en EX, PX o AR; así como de resultados BM y MM
3. los metadatos de interfaz entre lookup y agregación no deben introducir ineficiencia
4. buena escalabilidad respecto al tamaño del ruleset, tanto en profundidad (N) como en ancho (M)
5. moderada complejidad de pre-cómputo
6. moderado consumo de recursos
7. máximo desempeño (procesamiento a velocidad de línea)
8. moderados consumos de potencia y energía

Los requerimientos (1), (2) y (3) se relacionan íntimamente en el contexto de esquemas de clasificación por descomposición. Con ayuda de las Figs. 5.3(a) y 5.3(b) consideraremos las operaciones de lookup y update para MM y BM respectivamente, así como las complejidades de update para ambos.

Al realizar la operación de lookup, se ingresa un valor de key que resulta en una UR; esta UR representa el UV de más peso en el caso de BM o una combinacion particular de UVs en el caso

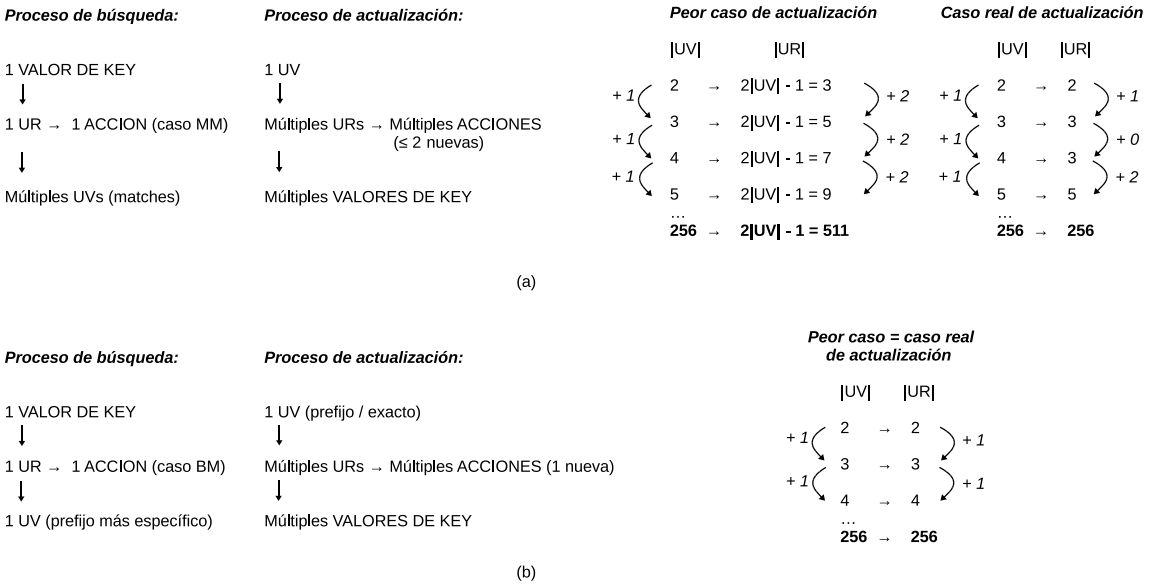


Figura 5.3: Procesos de búsqueda y actualización incremental: (a) caso MM, (b) caso BM

de MM. Es decir, para BM cada UR se relaciona con sólo un UV, mientras que en MM una UR se relaciona con un número no predecible de UVs. En cualquier caso, cada UR está bi-unívocamente asociada con una determinada acción a tomar sobre el paquete.

Cuando se realiza una operación de actualización, en tanto, se desea introducir una regla que se asocia con rangos de valores $[s, e]$ en cada campo del key $F_d(d = \{1, ..., k\})$. Para cada campo, estos rangos abarcan porciones de su respectivo espacio de búsqueda $F_d = \{0, ..., 2^{M_d}\}$. La adición de una regla multi-campo puede provocar o no la adición de un nuevo UV en un determinado campo, según este rango esté ya considerado o no por otra regla existente. Nos concentraremos en el caso de que la nueva regla suma un nuevo UV al campo considerado. En general, este nuevo UV impactará sobre múltiples URs; las URs totalmente abarcadas por el nuevo UV no se ven afectados, mientras que las nuevas URs se generan sólo en los extremos del nuevo UV. Así, la cantidad de URs afectadas serán no más de 2 en el caso MM, y no más de 1 en BM. Como se observa, las complejidades de lookup y actualización son en general diferentes para los casos BM y MM, así como para los casos EX, PX y AR; por ello en un esquema de clasificación general debe considerarse el requerimiento (2).

En la Fig 5.4 se consideran los posibles casos de solapamiento en AR y PX, y cómo la adición de un nuevo UV puede afectarlos. El caso EX no se considera por ser directo. En AR, como dijimos, los UV pueden solaparse completamente, parcialmente, o no solaparse en absoluto, mientras que en PX los UVs presentan solapamientos totales o nulos. Sin pérdida de generalidad, en las Figs. 5.4(a) y 5.4(b) se consideran dos UVs UV1 y UV2, y la actualización incremental mediante un UV adicional UV3. Se puede comprobar que los dos casos de overlap considerados son representativos de todos los casos posibles. Los demás patrones de solapamiento involucrando dos UVs son versiones espejadas de éstos, mientras que si consideramos mayor números de UVs iniciales se producen las mismas URs, sólo que involucrando más cantidad de UVs cada una. Según se puede comprobar en las actualizaciones incrementales de la Fig. 5.3, la adición de un nuevo UV genera como máximo 2

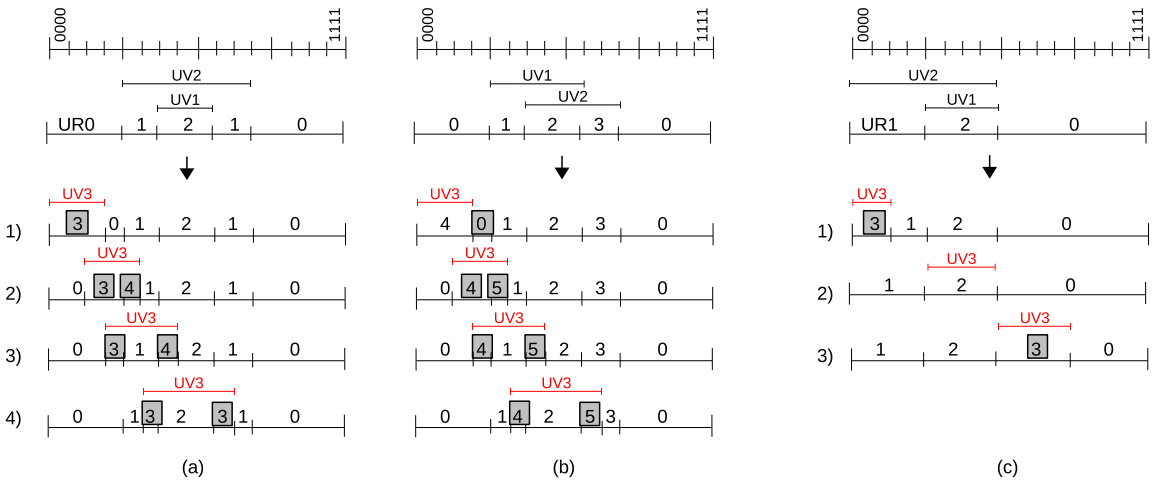


Figura 5.4: Actualización incremental: (a) (b) caso AR, (c) caso PX

nuevas URs en AR, y 1 nueva UR en PX. Como se expondrá más adelante, se puede aprovechar este hecho para reducir la complejidad de actualización de los esquemas basados en URs, que es actualmente su mayor limitación.

El requerimiento (3) surge de la clara separación entre etapas de lookup y agregación en un esquema de descomposición, y no ha sido considerado en ninguno de los trabajos previos. Los esquemas de lookup se han desarrollado como esquemas auto-contenidos y mayormente orientados a prefijos; mientras que los esquemas de clasificación por descomposición consideran un cierto formato de metadatos pero no estudian la efectividad de los esquemas de lookup para generarlos. Por ello, este requerimiento será uno de nuestros principales objetos de análisis.

El requerimiento (4) es dependiente del metadato utilizado como interfaz entre etapas. El requerimiento (5), en tanto, se relaciona con los (1), (4), (6), y (7). Cuanto mayor es la complejidad de pre-cómputo, mayor es la complejidad de actualización, se puede lograr mejor escalabilidad a través de la explotación intensiva del ruleset y mantener alto desempeño. En particular, los esquemas basados en búsqueda por URID tienen estas características; en ellos el cómputo se traslada esencialmente desde el momento de lookup hacia el momento de actualización, por lo que el lookup se realiza más eficientemente a costa de actualización más compleja. Mediante el almacenamento de resultados pre-procesados en memoria, se ahorran recursos lógicos, se disminuye el ancho de banda de memoria requerido y se logra mejor escalabilidad. Sin embargo, el mayor pre-cómputo requerido debe también satisfacer las tasas de actualización necesarias según la aplicación.

El consumo de potencia y energía (8) se han vuelto factores de importancia a considerar en nuevos trabajos. Para un esquema de pipelining, el consumo de energía implica el consumo de potencia durante el tiempo de latencia introducido por el pipeline [143]; ambos también se relacionan con (6) y (7).

5.3.3. Una taxonomía de esquemas de lookup

A fin de analizar la conveniencia de combinar determinados esquemas de lookup y agregación, comenzaremos por repasar los esquemas de lookup mencionados en algunos de los trabajos sobre clasificación por descomposición, si bien sus implementaciones no son detalladas en la mayoría de ellos. Los trabajos considerados son Lucent BV [105], DCFL [97], DCFLE [83], StrideBV [118], SCBV [119] y RFC [106]. [144] también provee una comparación general sobre esquemas de lookup, si bien no se realiza en el contexto de clasificación por descomposición como se pretende aquí.

Para match exacto, la solución más conveniente es el hashing según [97] y [144]. Para match de prefijos, [97] propone utilizar las técnicas Binary Search on Prefix Lengths y Tree Bitmap, mientras que [83] propone utilizar TCAMs, Balanced Interval Tree y Fat Inverted Segment Tree; [105] considera Binary search, [83] utiliza ETCAM, mientras que [118] y [119] utilizan TCAM emulada en FPGAs. RFC [106], en tanto, adopta indexado de memoria para todos los casos de match. Vemos que, si bien se mencionan muchas opciones, no existe una comparación clara de sus compromisos de diseño en relación con la etapa de agregación. Aún más, no todos los casos realizan implementación en una tecnología en común, por lo que no son directamente comparables; por ello se buscará en los siguientes párrafos realizar esta comparación utilizando en nuestro caso dispositivos FPGA. Comenzamos por analizar CAMs en FPGAs ampliando las opciones de emulación más allá de las consideradas en el Cap. 3; considerando luego los árboles de decisión binaria como esquemas que relajan la concurrencia de las CAMs en favor de menor ancho de banda de memoria. De este análisis, se puede visualizar una nueva taxonomía de los esquemas de lookup que permite analizarlos desde una base común.

Como se comentó en el Cap. 3, uno de los principales problemas de la TCAM para implementar ARs es la *expansión de rango*, lo cual ha generado numerosas propuestas. Para el caso de las FPGAs, el problema de AR puede abordarse de dos formas:

- almacenando en memoria los *resultados de búsqueda pre-procesados*, o
- almacenando los *extremos de rango* en forma separada y utilizando un comparador para efectuar la búsqueda

Llamaremos al primer método *memory-based* ya que utiliza exclusivamente direccionamieto de memoria, mientras que el segundo método se denominará *logic-based* ya que se basa en comparadores de magnitud combinacionales para realizar la búsqueda.

Si consideramos la implementación de una regla, vemos que los esquemas basados en memoria son más rápidos y requieren sólo $O(1)$ ancho de memoria; sin embargo por su expansión de direccionamiento requieren más memoria que la regla original, es decir, su *factor de utilización de memoria* es $\mu << 1$. Los esquemas basados en lógica, en tanto, presentan utilización similar al de las ETCAMs ($\mu = 0{,}5$) ya que requieren dos filas de M registros por regla; esta eficiencia es a costa de ancho de memoria $O(2M)$ y complejidad de ruteo $O(2M)$ para cada regla. Además, requiere

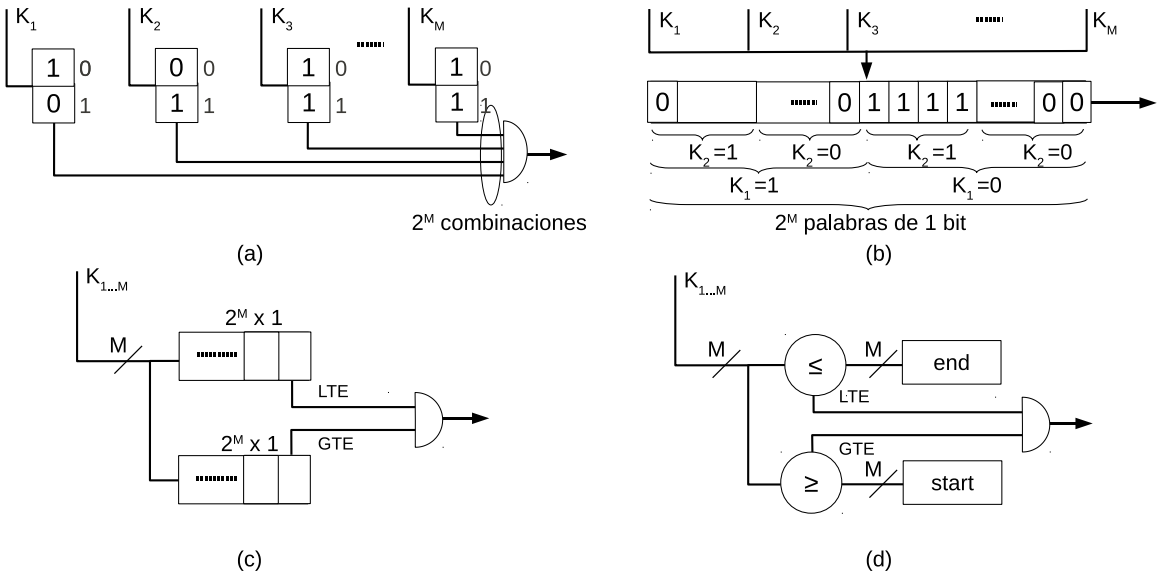


Figura 5.5: Lookup de 1 regla: (a) emulación de TCAM por registros, (b) memory-based interval case, (c) memory-based bounds case, (d) logic-based bounds case

comparadores de magnitud de ancho considerable que, como veremos más adelante, introducen ineficiencia. Volviendo al caso basado en memoria y una regla definida según un AR $[s, e]$, surgen dos alternativas. Por un lado, se puede almacenar el intervalo completo $[s, e]$ de la regla en un solo bloque de memoria; mientras que también se puede dividir este intervalo en dos rangos $[s, 2^M - 1]$ y $[0, e]$ y almacenar cada uno de ellos en un bloque independiente de memoria, luego la búsqueda se realiza tomando la conjunción de ambos resultados.

En la Fig. 5.5 se muestran todas las opciones de emulación de TCAM en FPGAs, teniendo en cuenta una regla. En la Fig. 5.5(a) se muestra la emulación fiel de una TCAM mediante registros; esta opción consume mínima memoria pero sufre la misma expansión de rango que las TCAMs nativas. La Fig. 5.5(b), en tanto, muestra la emulación de TCAM tal como se abordó en el Cap. 3, donde los rangos se resuelven a costa de mayor consumo de memoria; llamaremos a este método *memory-based interval case* ya que almacena todo el intervalo en un solo bloque de memoria. Las Figs. 5.5(c) y 5.5(d) muestran dos opciones para implementar el esquema de ETCAMs, llamados ERM en este trabajo. En la Fig. 5.5(c) se muestra el método utilizado en [83], que elimina el comparador de magnitud mediante pre-cómputo de los resultados de match para los intervalos $[s, 2^M - 1]$ y $[0, e]$, llamaremos a este método *memory-based bounds case*. El caso de la Fig. 5.5(d), utilizado en [119], implementa ERM mediante almacenamiento de límites y lógica de comparación, lo llamaremos *logic-based bounds case*.

Si ahora consideramos $|UV|$ reglas para una etapa de lookup, la búsqueda se puede realizar por UVs inviduales o por URs. Para un regla ambos métodos son indistintos ya que existe sólo un UV que define una UR; sin embargo para múltiples reglas ambos difieren según el solapamiento entre ellas. En general, un paquete puede satisfacer múltiples UVs, por lo que intuitivamente los esquemas basados en UVs deben ser concurrentes, replicando $|UV|$ veces alguna de las opciones de la Fig. 5.5. En la

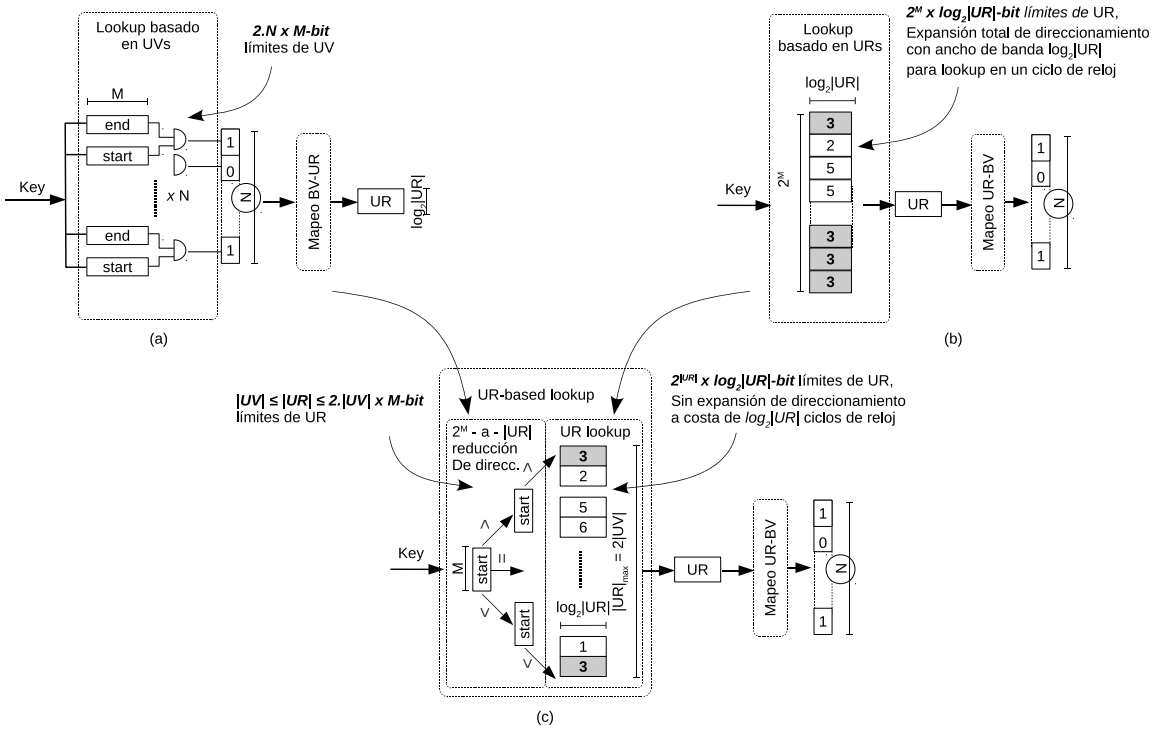


Figura 5.6: Lookup en N reglas: (a) búsqueda por UVs mediante método *logic-based bounds*, (b) búsqueda por URs por método *memory-based interval*, (c) reducción del espacio de direccionamiento mediante árboles

Fig. 5.6(a) vemos un ejemplo de búsqueda basada en UVs donde se utiliza el método de *logic-based bounds*. En las búsquedas por URs, por el contrario, sólo una UR coincide para cada paquete, por lo que las arquitecturas utilizadas pueden presentar menor paralelismo. En la Fig. 5.6(b) se muestra el caso extremo donde se pre-computan los RIDs almacenándolos en un bloque de memoria similar al de la Fig. 5.5(b) pero de ancho $log_2|UR|$. La Fig. 5.6(c), en tanto, muestra un caso intermedio donde se utiliza un árbol binario auxiliar para reducir la expansión de direccionamiento. Este árbol toma los límites de la Fig. 5.6(a) y los analiza sucesivamente, generando como resultado una dirección de memoria de ancho (y expansión de direcionamiento) reducido. En efecto, la etapa de *reducción de direccionamiento* (árbol binario) de la Fig. 5.6(c) dispone $2|UV| - 1$ límites $[s, e]$ en $log_2|UR|$ niveles; esto genera como máximo $|UR|$ direcciones eliminando la redundancia de direcciones de la Fig. 5.6(d). De este análisis, podemos considerar a los tres esquemas como casos especiales de una solución al problema de lookup, con distintos compromisos en cuanto a almacenamiento, desempeño, latencia y pre-cómputo.

Por otra parte, podemos analizar los esquemas de lookup desde el punto de vista del *metadato de salida*; éste puede ser basado en URs o UVs y en bitmaps o etiquetas, como se mostró en la Fig. 5.1. Cada método de búsqueda tiene un metadato de interfaz asociado naturalmente; para el caso de búsqueda por UVs es el bitmap mientras que para la búsqueda por URs son las etiquetas de ancho $log_2|UR|$. En el caso de que la etapa de agregación requiera otro formato, se puede adoptar otro tipo de lookup o implementar etapas de adaptación como las mostradas en la Fig. 5.6; estas etapas pueden sin embargo disminuir el rendimiento natural del esquema de lookup.

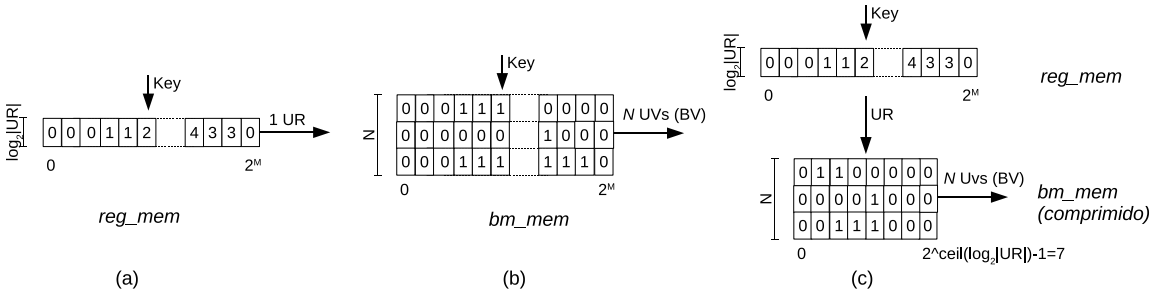


Figura 5.7: Metadato en IND: (a) etiquetas UR, (b) bitmap de UVs, (c) combinación de ambos

Ejemplos de lookup basado en UVs son [83] y [119], mientras que la búsqueda por URs se utiliza en [106] y [105] (que en efecto requiere una etapa de adaptacion de UR(label) a UV(bitmap)). Las arquitecturas concurrentes, tales como TCAM y ETCAM, realizan naturalmente búsqueda por UVs, mientras que los esquemas de emulación de TCAM con memoria y los árboles binarios realizan naturalmente búsqueda por URs. La búsqueda por UVs tiene muy baja exigencia de pre-cómputo por su mapeo directo entre reglas y metadatos, pero carece de escalabilidad ya que no es capaz de introducir optimizaciones basadas en el ruleset. Los esquemas basados en URs, en cambio, pueden explotar casos de moderado solapamiento para soportar grandes rulesets, pero requieren para ello de mayor pre-cómputo durante la actualización de reglas.

5.4. Diseños propuestos

En base a nuestro análisis anterior y la nueva taxonomía definida, se proponen tres diseños standard que pueden ser combinados con las arquitecturas de agregación mencionadas en el Cap. 4. Estos diseños consideran el caso de rangos generales (GRs) que permite englobar EX, PX y AR. Por un lado, permiten implementar búsqueda por UVs o URs; mientras que en otro orden permiten adoptar metadatos basados en UVs o URs bajo la forma de bitmaps o etiquetas, según la necesidad de la etapa de agregación. Estos diseños son implementados en FPGAs y evaluados en cuanto a su desempeño y consumo de recursos. De esta manera, se genera extensiva información sobre sus características que puede ser utilizada como criterio de diseño. Los casos considerados son:

- direccionamiento de memoria (Memory Indexing, IND)
- árbol de búsqueda binaria (Binary Search, BS)
- comprobación explícita de rangos (Explicit Range Match, ERM)

Para cada caso, se plantean y analizan optimizaciones tanto a nivel teórico como de arquitectura.

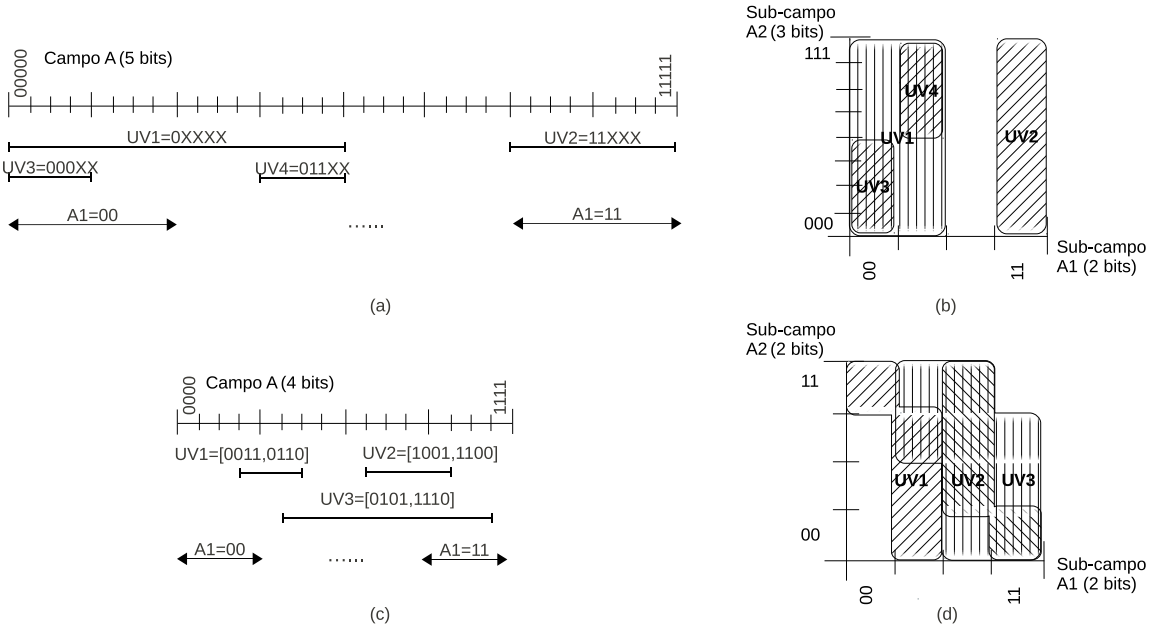


Figura 5.8: Efecto del particionado (stitching): (a) lookup basado en PX, (b) particionado de PX, (c) lookup basado en AR, (d) particionado de AR

5.4.1. Direccionamiento de memoria (IND)

La técnica de direccionamiento de memoria tiene la flexibilidad de adaptarse fácilmente a etapas de agregación basadas en UVs o URs, si bien es esencialmente una técnica de búsqueda basada en URs. De esta forma, es capaz de implementar efectivamente la etapa de adaptación UR-UVs de la Fig. 5.6(b) como parte de su propia arquitectura de lookup. Esto se realiza almacenando bien un URID de ancho $log_2|UR|$ o un bitmap de UVs de ancho N en cada una de las 2^M posiciones de memoria asociadas con URs. Las Figs. 5.7(a) y 5.7(b) muestran ambas opciones para el ruleset de ejemplo de la Fig. 5.8(c). La fig. 5.7(c), en tanto, muestra la combinación de ambos mediante dos módulos de memoria, *reg_mem* que mapea valores de key a URs seguido por *bm_mem* que mapea URs a bitmaps. Se muestra el caso de BV, donde se requiere un bitmap de ancho N para cualquier dimensión.

5.4.1.1. Agregación intra-campo

El direccionamiento de memoria puede implementar cualquier rango de ancho W mediante memoria de al menos 2^W posiciones; sin embargo esto causa expansión de direccionamiento que puede ser impracticable para valores grandes de W. Se puede ver al direccionamiento de memoria como un árbol de búsqueda de sólo una etapa. La expansión de direccionamiento establece en última instancia un límite práctico; para extender este límite la expansión se puede aliviar mediante *segmentación horizontal o vertical* según lo propuesto en [145]. La segmentación horizontal, ya discutida en el Cap. 3 se logra dividiendo los M bits del key en $\lceil M/m \rceil$ segmentos o *chunks* de ancho m, di-

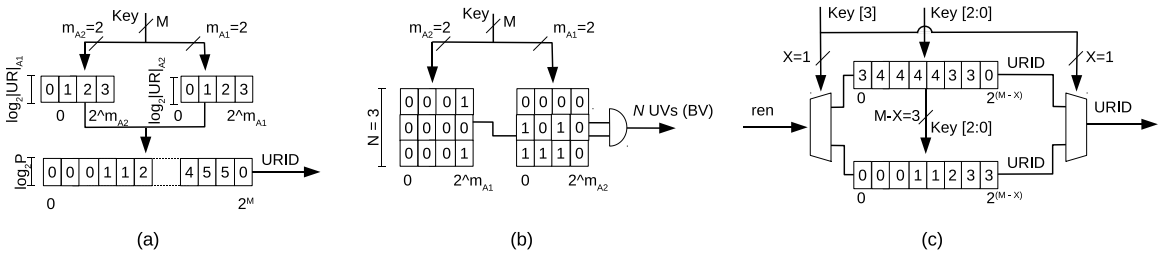


Figura 5.9: Segmentación de campo: (a) horizontal por URs, (b) horizontal por bitmaps BV, (c) vertical

reccionando cada uno su propio bloque de memoria; sin embargo esto introduce dos sub-campos ortogonales entre sí dentro del lookup 1D. En las Figs. 5.8(a) y 5.8(b) se ilustra claramente este hecho para el caso de un ruleset PX en un campo A con $M = 5$ y $N = 4$, dividido en dos segmentos $A1$ y $A2$ de 2 y 3 bits respectivamente, mientras que las Figs. 5.8(c) y 5.8(d) ilustran un ruleset AR de 4 bits dividido en dos segmentos de 2 bits. Como se observa, la búsqueda en cada segmento no es ya sensible a los restantes segmentos; por ejemplo dado un key=00110, el segmento $A1$ devuelve la regla UV1 pero luego depende del resultado en el segmento $A2$ para determinar el resultado definitivo en el campo A. En consecuencia, los resultados para cada sub-campo requieren de una etapa adicional de *agregación intra-campo*; esta etapa presenta una diferencia fundamental respecto a la agregación *inter-campo* analizada en el Cap. 4. Como se observa en la Fig. 5.8(b) para el caso de prefijos, los rangos de valores de A_2 para cada rango de valores de A_1 son uniformes; por ejemplo $F_{A2} = [000, 111]$ para $F_{A1} = [000, 011]$. En la Fig. 5.8(d), en cambio, se observa que para el caso de ARs los rangos definidos para A_2 dependen del valor particular en $A1$; por ejemplo, en UV3 $F_{A2} = [00, 11]$ para $F_{A1} = 10$ mientras que $F_{A2} = [00, 10]$ para $F_{A1} = 11$. Los patrones de la Fig. 5.8(b), basados en hiper-rectángulos, son similares a los patrones que se presentan en clasificación por descomposición; sin embargo los patrones que surgen de la segmentación de ARs son *nuevos patrones geométricos*, exclusivos al caso de segmentación intra-campo.

La agregación intra-campo se puede implementar de dos formas principales:

1. obteniendo URs locales a cada segmento y agregándolas en una etapa similar a RFC [106]; la etapa de lookup entrega luego la UR resultante. Este método se ilustra en la Fig. 5.9(a).

2. utilizando bitmaps internos a cada segmento. Si se desea agregar mediante ANDs, estos bitmaps deben ser como mínimo de ancho $|UV|$; debiendo ser de ancho N si la agregación es basada en BV [105]. Este método se ilustra en la Fig. 5.9(b).

Ambos esquemas pueden implementar rangos generales (EX, PX y AR), *aun considerando rangos que abarquen más de un segmento* ($W > m$). Sin embargo, debido al problema de expansion de rango y como se discutió en el Cap. 3, para el caso (2) se debe propagar en general más de 1 bit por UV. Esto, como veremos, permite más flexibilidad que la emulación de TCAM del Cap. 3 a costo de expansión de rango *reducida* a un valor fijo.

La segmentación vertical, ilustrada en la Fig. 5.9(c), reemplaza parte de los circuitos internos

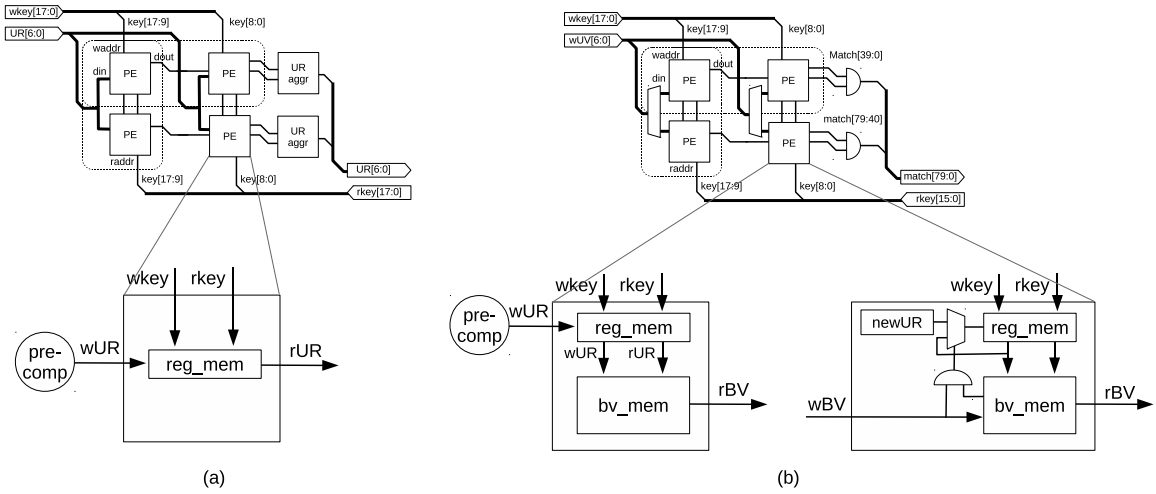


Figura 5.10: Arquitectura de segmentación horizontal: (a) agregación por URs, (b) agregación por UVs (BV)

de la SRAM con lógica externa de multiplexado (MUX)/demultiplexado (deMUX); de este modo se pueden utilizar bloques menos profundos y más anchos lo que reduce la expansión de direccionamiento y en definitiva mejora la utilización de memoria. En este esquema, X bits del key son utilizados para control de los MUX/deMUX, mientras que los restantes $M - X$ bits sirven de bus de direccionamiento común a todos los bloques de memoria. Las señales de control *ren/wen* se demultiplexan hacia sus respectivos bloques, mientras que sus puertos de datos se multiplexan hacia el puerto de datos de salida. Un aspecto sobresaliente de este método es que el resultado de lookup proviene ahora *exclusivamente* de uno de los bloques de memoria, por lo que no es necesaria su agregación. Esto permite, por ejemplo, realizar habilitación selectiva de bloques para obtener ahorro de energía.

Para nuestros objetivos, los casos particulares a implementar dependerán en general de los recursos disponibles en un FPGA, ya introducidos en el Cap. 3. En las Figs. 5.10(a) y 5.10(b) se reproducen arquitecturas de segmentación horizontal similares a la de la Fig. 3.7(b) para los casos de segmentación por URs de la Fig. 5.9(a) y por bitmap de UVs (BV) de la Fig. 5.9(b) respectivamente. En estos ejemplos $M = 18$, $N = 80$, $|UR| = N = 80$ y BRAM en modo 512×40. Estos esquemas utilizan pipeling tanto horizontal como vertical formando un array sistólico, a fin de lograr adecuada escalabilidad con M y N. En los casos en que se utilizan URs, es necesario pre-cómputo para considerar el efecto de un nuevo UV sobre múltiples URs, este pre-cómputo se implementa comunmente en una plataforma externa basada en procesadores de propósito general (GPPs). En la Fig. 5.10(b) se ilustra el procesamiento involucrado en forma simplificada; se observa que el nuevo bitmap wBV debe esencialmente compararse con el bitmap almacenado para cada valor del nuevo rango, y en base al resultado de comparación se mantiene el URID actual o se reemplaza por un nuevo URID $newUR$. Abordaremos este procesamiento en más profundidad en la Sec. 5.5.

Las memorias BRAM pueden utilizarse en modos de puertos *simple, dual,* o *true dual*. El modo de puertos true-dual es utilizado intensivamente en trabajos previos para realizar dos lecturas simultáneas desde dos paquetes sucesivos, duplicando efectivamente la tasa de lookup. Sin embargo, en este modo se deben considerar dos problemas. Por un lado, el uso de puertos true-dual limi-

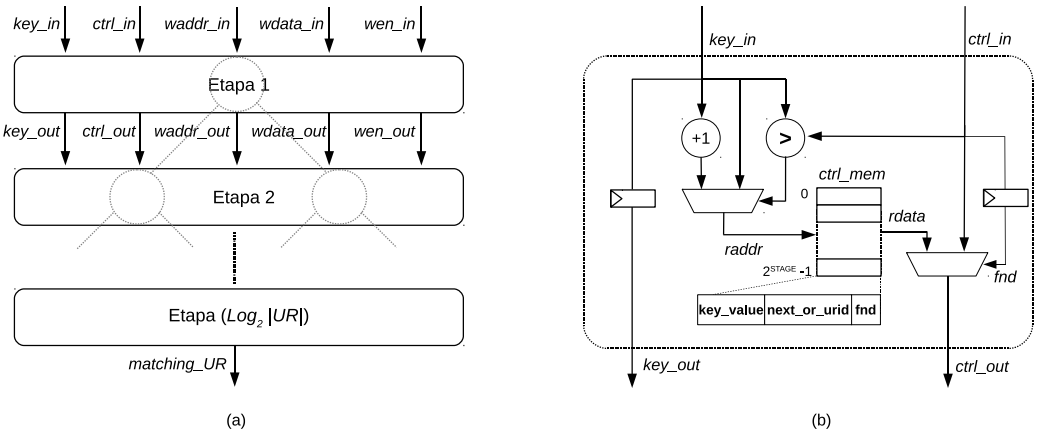


Figura 5.11: Árbol de decisión binario: (a) arquitectura general, (b) arquitectura de nodo (1 nodo/nivel)

ta los factores de forma disponibles, por lo que la utilización de memoria es peor que en modos simple o dual [146], esto en definitiva significa mayor consumo de memoria para el mismo ruleset. Por ejemplo, los modos a utilizar para las arquitecturas 5.10(a) y 5.10(b) estarán limitados a factores de forma $1K \times 20$ - $20K \times 1$ en lugar de 512×40 - $20K \times 1$, de modo que se necesitará un mínimo de $\lceil M/m \rceil \times \lceil N/n \rceil = \lceil 18/10 \rceil \times \lceil 80/20 \rceil = 8$ bloques para true dual-port en lugar de los $\lceil 18/9 \rceil \times \lceil 80/40 \rceil = 4$ necesarios en simple o dual-port. Por otro lado, el uso de ambos puertos en modo lectura simultáneamente no permite realizar actualización y búsqueda simultáneamente, por lo que el pipeline de lookup se interrumpe durante una actualización de reglas. Otra característica explotada en trabajos previos es el uso de distRAM para implementar pequeñas secciones de memoria donde el uso de BRAM es ineficiente, por ejemplo para segmentos de $m << 9$. Sin embargo, la memoria distRAM no soporta modo true dual-port, por lo que no puede combinarse en igualdad de condiciones con BRAM como lo proponen [133] y [117]; además este tipo de memoria no es igualmente implementable en las distintas FPGAs. Considerando estos factores, en este trabajo se utilizan BRAMs en modo simple dual-port, destinando un puerto a lookup y el restante a actualización. Como ya se explicó en el Cap. 3, estos esquemas permiten lookup en un ciclo mientras que la actualización puede tomar múltiples ciclos dependiendo del intervalo (scope) de la nueva regla.

5.4.2. Árbol de búsqueda binario (BS)

Para la evaluación de este esquema, se adaptaron las arquitecturas presentadas en [133] y [147] para el caso de un campo. En la Fig. 5.11(a) se muestra la arquitectura general del árbol implementada como un pipeline de etapas, mientras que la Fig. 5.11(b) muestra la arquitectura interna de cada etapa. El control de decisiones en cada etapa se logra mediante el bloque de memoria *ctrl_mem*. Para una etapa i conteniendo 2^{i-1} nodos, este bloque de memoria consta de 2^i posiciones; dos posiciones consecutivas de memoria mapean dos ramas de un nodo en dicha etapa. El puerto *key_in* propaga el valor del key a través de las etapas del árbol; el puerto *ctrl_in*, en tanto, agrupa (1) el límite *key_value* a comparar contra el key, (2) un campo *next_or_urid*, y (3) la bandera *fnd* que indica si se ha encontrado ya una coincidencia para el key entrante. En general, cualquier nodo puede generar

valores de offset para la siguiente etapa *next* e identificadores de UR *urid*, ya que se pueden encontrar múltiples coincidencias a lo largo del árbol. En nuestro caso, sin embargo, las decisiones se toman en base a límites de UR en lugar de límites de UV, por lo que los nodos intermedios generan exclusivamente valores de offset y los nodos hoja generan exclusivamente valores de UR.

Para actualización del ruleset, el bloque *ctrl_mem* cuenta con un segundo puerto configurado para escritura; mientras que las señales correspondientes *wdata_in/wdata_out*, *waddr_in/waddr_out* y *wen_in/wen_out* se propagan por el pipeline a fin de actualizar el nodo adecuado. Los paquetes que arriban antes de iniciarse la actualización se propagan por el pipeline un ciclo antes de verse afectadas las tablas respectivas; mientras que el paquete entrante un ciclo después de iniciada la actualización y los posteriores a él se procesan a lo largo del pipeline con el ruleset actualizado; de este modo se logra sincronización en los datos utilizados. En tanto, las señales *key_in*, *waddr_in*, *wdata_in* y *wen_in* son registradas en cada etapa para mantener sincronización con las señales *ctrl_in*/*ctrl_out*, las cuales están naturalmente defasadas un ciclo por la operación de lectura en memoria.

Uno de los principales problemas de las arquitecturas basadas en árboles es que el consumo de memoria puede variar drásticamente de una etapa a otra, especialmente en rulesets grandes. Esto conduce a utilización ineficiente de bloques RAM en las etapas cercanas a la raíz, y necesidad de grandes bloques de memoria en las últimas etapas del pipeline. Existen varias técnicas para mitigar este problema mediante la combinación de registros, memorias DistRAM, BRAM y DRAM externa según los requerimientos en cada etapa [133]. Para nuestros objetivos, se indicó a la herramienta de síntesis mantener un balance de velocidad vs. área, lo que en la mayor parte de los casos resultó en el uso de memoria BRAM. La principal limitación de velocidad está dada por el comparador de magnitud necesario en cada etapa; este módulo puede ser especialmente lento para keys anchos. Una posible optimización a este problema, no considerada aquí, sería una combinación de pipelining horizontal y vertical [119].

5.4.3. Comprobación explícita de rangos (ERM)

ERM adopta naturalmente el método de búsqueda por UVs, comprobando independientemente el valor del key contra los límites de rango de cada UV; y su formato natural de salida es un bitmap de ancho N. En consecuencia, este esquema escala con N sin importar los patrones de solapamiento de reglas, y requiere una etapa de conversión adicional en el caso de necesitar una salida basada en etiquetas. Para un key de ancho M, esta arquitectura requiere acceso concurrente a $2 \cdot M \cdot N$ registros, así como lógica adicional para comparación de magnitud.

Nuestra primera implementación fue basada en la arquitectura [119]; en la Fig. 5.12(a) se muestra el pipeline sistólico general donde el key se propaga verticalmente y los resultados horizontalmente. En [119] se incluyen codificadores de prioridad a la salida de cada fila; estos elementos se indican en nuestro caso como módulos de conversión general de BV a UR, ellos se abarcan desde los simples codificadores de prioridad para el caso de BM hasta la compleja implementación que requiere el caso MM. En la Fig. 5.12(b) se muestra el elemento de procesamiento (PE) implementado en [119]; en ese trabajo se asegura que dicho elemento es capaz de soportar ARs propagando sólo un bit; nosotros

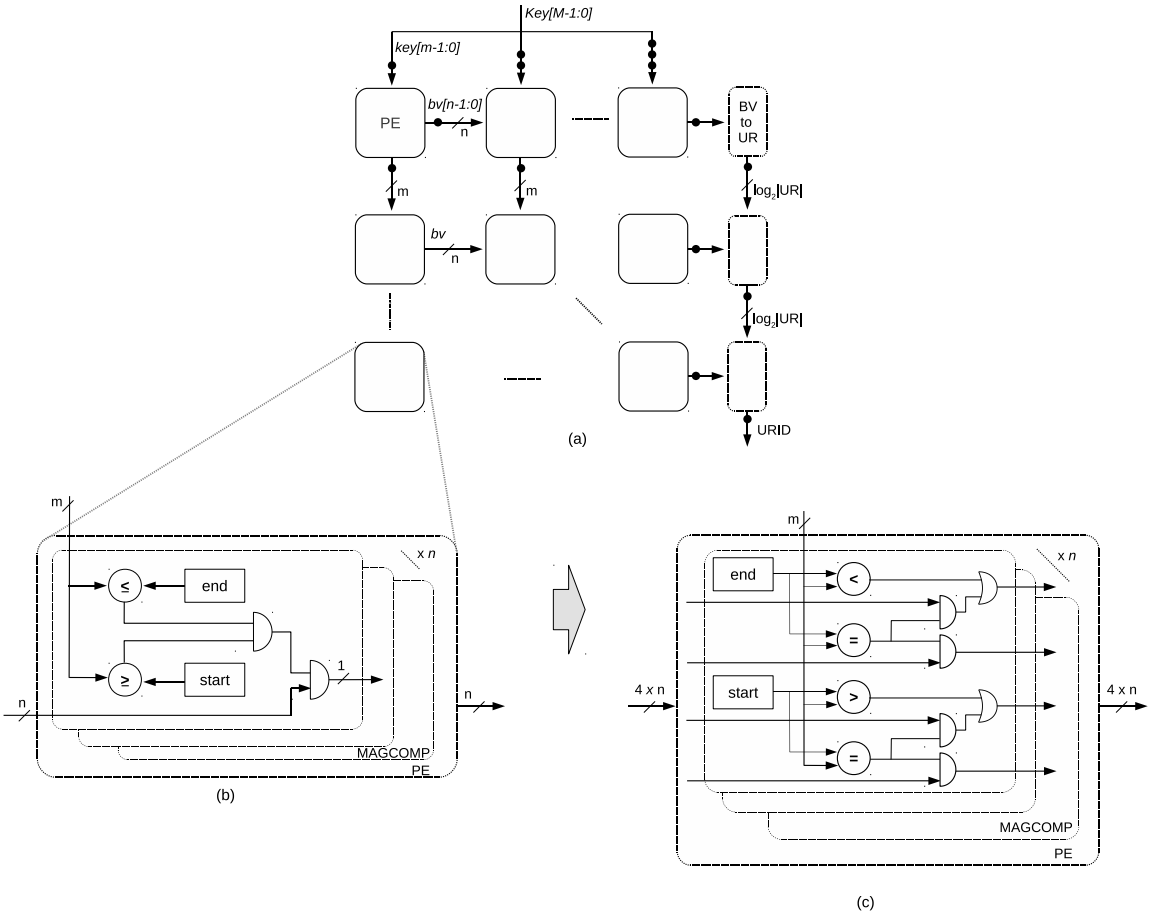


Figura 5.12: Comprobación explícita de rangos: (a) pipeline general, (b) nodo para PX, (c) nodo propuesto para soporte de AR

sin embargo comprobamos que sólo el caso PX es soportado con este esquema por lo que se propone una arquitectura modificada. Esto será justificado a continuación.

En las Figs. 5.13(a) y 5.13(b) se consideran dos rulesets basados en PX y AR respectivamente, los cuales se dividen en dos secciones. Adoptaremos el orden $A_1, A_2, ...A_{M/m}$ para representar secciones que agrupan desde los bits mas significativos *(Most Significant bits, MSbits)* hasta los bits menos significativos *(Least Significant bits, LSbits)*. El caso de prefijos, como se observó en la Fig. 5.8, no presenta mayores inconvenientes y puede implementarse propagando un bit entre secciones tanto en ERM como en IND. El caso de AR, sin embargo, requiere propagar más de un bit entre secciones. La Fig. 5.13(b) muestra dos ARs $[s, e]$ que resultan en intervalos $[s_1, e_1]$ y $[s_2, e_2]$ en A_1 y A_2 respectivamente. Para UV1 tenemos $s_2 < e_2$, mientras que $s_2 > e_2$ para UV2. Para UV1, ilustrado en la Fig. 5.8(c), observamos que si se comprueban independientemente los rangos $[s_1, e_1] = [1, 3]$ y $[s_2, e_2] = [2, 2]$ y se toma su conjunción se obtiene un rango discontinuo $(A = [6]) \cup (A = [10]) \cup (A = [14])$. Para resolver este problema, observamos que los rangos en segmentos de mayor peso abarcan múltiplos enteros de los espacios de búsqueda restantes $0 \leq A_i \leq 2^m (i = 2...\lceil M/m \rceil)$. En nuestro caso, esto significa que el rango del segmento de mayor peso $s_1 < A_1 < e_1$ abarca múltiplos enteros del rango de menor peso $0 \leq A_2 \leq 4$ (rectángulo sombreado en la Fig. 5.13(e)).

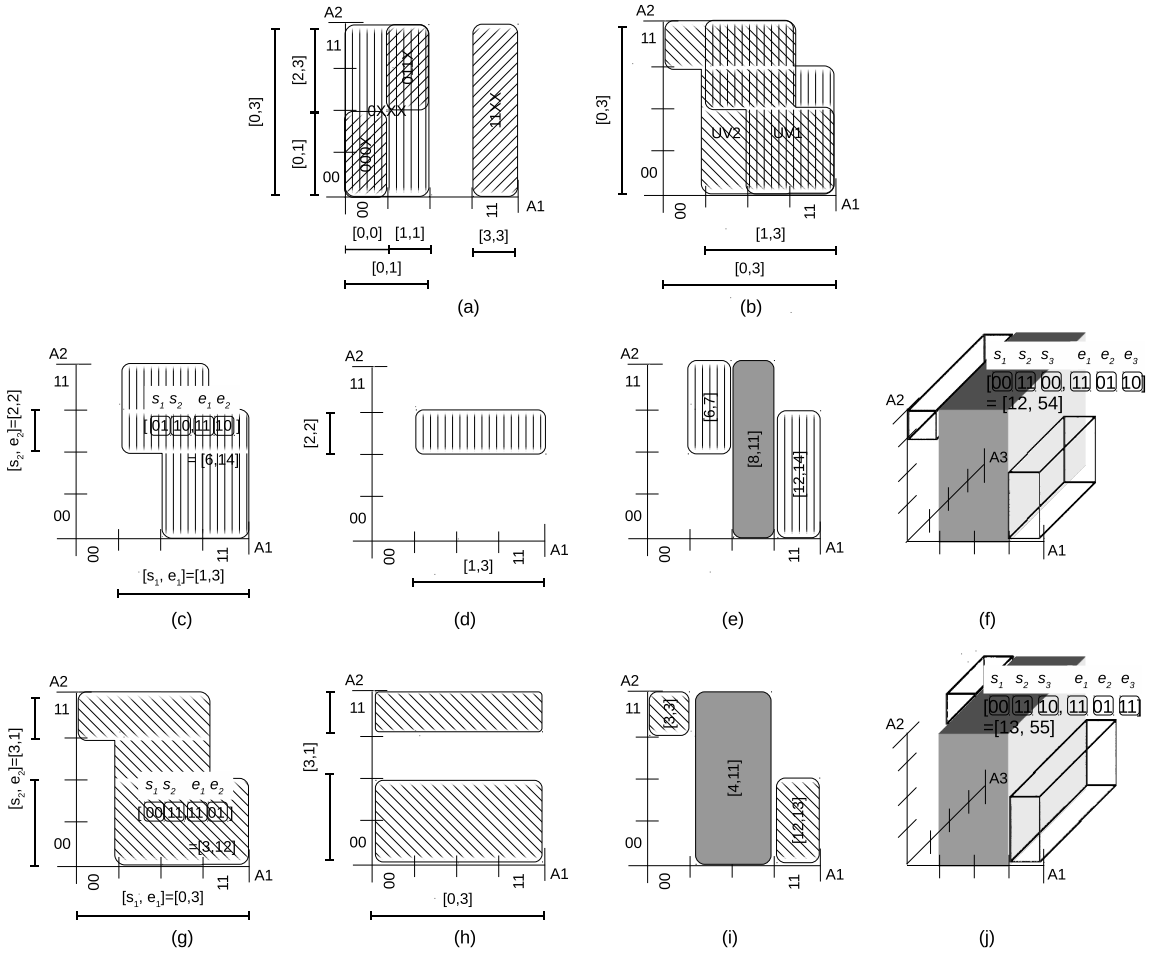


Figura 5.13: ERM segmentado: (a) ruleset PX, (b) ruleset AR, (c)(d)(e) AR donde $s_2 < e_2$, (f)(g)(h) AR donde $s_2 > e_2$, (i)(j) extensión de (h) al caso de 3 campos

En base a esta observación, se puede implementar AR propagando los resultados $A_i = s_i$, $A_i = e_i$ y $s_i < A_i < e_i$ en forma separada para $i = \{1, ..., \lceil M/m \rceil\}$, es decir 3 bits por regla. Para el caso de ERM, los resultados de $s_i < A_i$ y $A_i < e_i$ deben a su vez propagarse en forma separada en la cascada de comparadores de magnitud, lo que resulta en 4 bits por regla. En la Fig. 5.13(f) se muestra el caso de UV2, ilustrando los resultados incorrectos de propagar 1 bit en la Fig. 5.13(g) y los resultados correctos para nuestra arquitectura en la Fig. 5.13(h). La arquitectura del nuevo PE para propagar los cuatro bits necesarios se muestra en la Fig. 5.12(c). A fin de generalizar nuestras observaciones, en las Figs. 5.13(i) y 5.13(j) se agrega un tercer campo A_3 con dos rangos de ejemplo $A_3 = [00, 10]$ y $A_3 = [10, 11]$. Como se observa, el rectángulo sombreado se extiende ahora a un paralelogramo que abarca múltiplos enteros tanto de $A_2 = [00, 11]$ como de $A_3 = [00, 11]$ para el rango de MSbits $s_1 < A_1 < e_1 = [01, 10]$; mientras que los resultados para $A_1 = s_1 = 00$ y $A_1 = e_1 = 11$ deben propagarse en forma separada. En la Fig. 5.12(c) se muestra el PE modificado según estas observaciones, mientras que en la Fig. 5.14(a) se muestra el pipeline completo de lookup para una regla. Cada columna de esta última figura corresponde al bloque indicado como *MAGCOMP* en la Fig. 5.12(c).

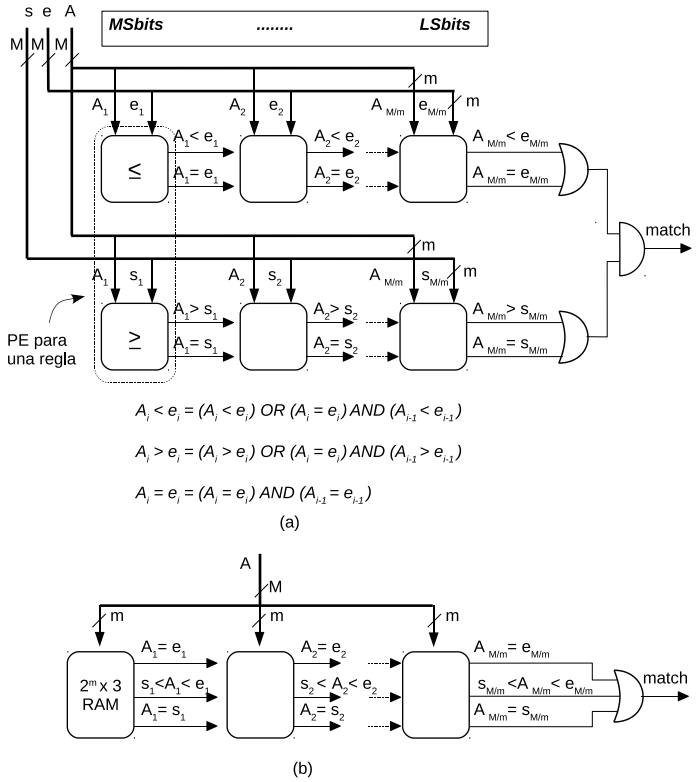


Figura 5.14: Lookup para una regla: (a) ERM segmentado, (b) IND segmentado

Volviendo a lo estudiado para IND, vemos que los resultados para $A_i = s_i$, $A_i = e_i$ y $s_i < A_i < e_i$ pueden ser pre-computados y almacenados en memorias de profundidad 2^m y tres bits de ancho para cada regla; el pipeline correspondiente se muestra en la Fig. 5.14(b) y realiza la misma función que su homólogo ERM. Es importante destacar que, mediante esta modificación, el esquema IND no depende de la expansión de direccionamiento para implementar ARs como se estudió en el Cap. 3, por lo que se puede seleccionar libremente el factor de forma más conveniente para aprovechamiento de memoria. Si bien en este caso el ancho de memoria se incrementa por un factor de 3, para rangos de gran scope esto puede ser mucho más económico en memoria que utilizar expansión de direccionamiento. Aún más, este aumento en ancho, acotado por un máximo de 4 bits/regla y 3 bits/regla para IND y ERM respectivamente, es mucho menor que el aumento necesario para implementar ARs en TCAM nativa, el cual como se vio puede ser w bits/regla para un campo con rangos de ancho w.

Los esquemas de las Figs. 5.14(a) y 5.14(b) representan ambos una fila de lookup completa para $n = 1$, la que se replica para implementar un esquema de lookup $M \times N$ como el de la Fig. 5.12(a). Como efecto neto de las modificaciones introducidas al PE, se requerirán $4 \cdot \lceil M/m \rceil \cdot N$ registros para implementar el pipeline horizontal, además de requerirse lógica adicional en cada PE; en definitiva se requerirán más recursos para soportar N reglas en AR que para soportar las mismas N reglas en PX. En el caso de IND, el efecto neto consiste en mayor ancho de memoria y mayor cantidad de registros para soportar el mismo ruleset.

Se debe observar que, en un contexto de clasificación multi-dimensional, estos bits adicionales deben propagarse sólo dentro del campo donde se definen ARs. En los puntos de interfaz con la etapa de agregación inter-campo, es decir a la salida de las etapas de lookup, se combinan los resultados en un bit único para cada regla. Esto significa que, por ejemplo, si una regla definida sobre 4 campos considera ARs sobre los campos A y B, y PX sobre los campos C y D, los campos C y D pueden aún resolverse con 1 bit/regla. Esta es una diferencia fundamental con las arquitecturas basadas en TCAM nativa, donde las múltiples reglas resultantes de la expansión de rangos de una regla en múltiples campos deben agregarse mediante su producto combinacional, generando w^k pseudo-reglas para un rango de ancho w definido sobre k campos.

5.5. Actualización dinámica

Como se introdujo en el Cap. 3, en un esquema de clasificación kD el ruleset resulta afectado en dos momentos:

1. durante la construcción del ruleset con un conjunto inicial de reglas, llamamos a esta operación *almacenamiento*
2. durante la operación normal, al agregar o borrar reglas; llamamos a esta operación *actualización incremental o dinámica*

La complejidad de ambas operaciones depende fundamentalmente del método de búsqueda empleado, es decir, basado en URs o en UVs. Como se discutió en la Sec. 5.3, la actualización consiste esencialmente en la adición o sustracción de un UV, el cual puede afectar a múltiples URs. En consecuencia, los esquemas de búsqueda basada en UV tienden a presentar menor complejidad de actualización ya que no se deben computar las URs afectadas. De las posibles operaciones de actualización, que son *set, clear y modification*, se considerará en este apartado la operación *set*; las demás pueden luego relacionarse con ésta.

En los párrafos siguientes se analizan requerimientos de actualización para ERM como representativo de búsqueda por UVs, y IND como representativo de búsqueda por URs. Las métricas involucradas son tiempos de pre-cómputo y consumos de memoria asociados, tanto en la plataforma de actualización (GPP) como en la plataforma de búsqueda (FPGA). En particular, se proponen optimizaciones para el caso de búsqueda por URs, el cual se ve actualmente muy limitado por sus requerimientos de actualización.

En ERM, cada regla se asocia a 2 palabras de M-bits, resultando en $2 \cdot N \cdot M$ registros para el ruleset completo. Además, se asocia un bit *valid* a cada par de límites $[start, end]$; este bit enmascara los resultados de comparación indicando si esa regla es válida dentro del ruleset. Durante el almacenamiento o actualización de reglas, se provee en primer término un *identificador de regla (Rule ID, RID)* que es decodificado para determinar qué posición ocupa la regla en el ruleset; RID esencialmente define el peso de las reglas. Una operación de *set* almacena luego los dos límites

$[start, end]$ y activa el bit *valid* correspondiente; mientras que la operación *clear* desactiva el bit *valid*. La operación *modification*, finalmente, cambia los límites asociados a una RID ya existente (es decir con su bit *valid* activado). En [119] se analizan en detalle estos conceptos.

Observamos que la etiqueta RID es un concepto fundamental en un esquema de búsqueda basada en UVs tal como ERM. Cada RID se asocia a un intervalo de valores del key y a una prioridad o peso específico. En general, múltiples RIDs pueden resultar de la búsqueda, sin embargo en el caso BM, sólo un RID y su peso asociado determinan de manera directa la *acción* a tomar sobre el paquete. En esquemas basados en UVs, el almacenamiento de RIDs es directo mientras que un codificacor de prioridad se encarga de seleccionar un RID durante la búsqueda. En el caso MM, sin embargo, la acción a tomar está determinada por la *combinación* específica de RIDs coincidentes, resultando en $|UR|$ posibles resultados de lookup para el *mismo* conjunto de reglas. En este caso no se puede utilizar un codificador, sino que se debe de alguna manera mapear RIDs a URIDs. Surgen así dos opciones:

1. almacenar RIDs (o UVIDs) individuales, y reemplazar el codificador de prioridad por hardware de compresión N (o $|UV|$) a $|UR|$; esta etapa adicional debe realizar intensivo procesamiento con alto costo de implementación y limitado desempeño, afectando seriamente la arquitectura de lookup.

2. *durante la actualización*, considerar todos los UVs existentes que se cumplen en el intervalo del nuevo UV, asignando URIDs a cada una de las combinaciones particulares. Estos URIDs se almacenan en la memoria de lookup. Este método esencialmente desplaza el procesamiento de URIDs desde la plataforma de lookup hacia la plataforma de actualización; dado que esta última no posee limitaciones en cuanto a recursos de cómputo y memoria, resulta más conveniente para el procesamiento requerido. Esta opción será evaluada a continuación.

5.5.1. Optimización del algoritmo de actualización

Como se mencionó previamente, tanto el almacenamiento del ruleset como la actualización de reglas son más complejos en esquemas de lookup por URs que en sus contrapartes por UVs. Aun así, los esquemas de lookup por URs son especialmente veloces por su reducido ancho de memoria y simplicidad de implementación; además son capaces de implementar tanto BM (RIDs o UVIDs) como MM (URIDs), lo cual es especialmente útil en arquitecturas de red heterogéneas basadas en flujos; por ello es de interés optimizar en particular su actualización dinámica.

El esquema típico para búsqueda por URs es RFC [106]; el pre-cómputo necesario para almacenamiento de reglas en este esquema puede tomar muchas horas para rulesets con solapamientos complejos tales como los de *IPC1* [134]. Esta complejidad de almacenamiento se redujo en [96] y [108]. Sin embargo, en ninguno de estos esquemas se diferencia el caso de almacenamiento del caso de actualización, por lo que la adición de una regla puede presentar la misma complejidad que la construcción del ruleset completo. Nosotros consideramos ambos casos por separado, con lo que la actualización se puede simplificar significativamente.

Algoritmo 1 Algoritmo de almacenamiento de ruleset para IND

Require: Limites de rango $[UV.s, UV.e]$ para todas las reglas en cada segmento, numero de reglas N

```
1:  for chunk = 0 to ⌈M/m⌉ do
2:    p ← 0
      // simular búsquedas para cada regla y construir el vector BV
3:    for i = 0 to 2^m do
4:      for j = 0 to N do
5:        if i ≥ UV[j].s and i ≤ UV[j].e then
6:          BV ← BV or 2^j
7:        end if
8:      end for
        // comprobar si el BV ya existe para otro valor de key
9:      j ← 0
10:     while j ≤ p and BV ≠ bv_mem[j] do
11:       j ← j + 1
12:     end while
        // si no se encontró el BV, incrementar el contador de URs
13:     if j ≥ p then
14:       p ← p + 1
15:     end if
        // almacenar UR y UV para el valor actual de key. Sólo ur_mem se transferirá al motor de
        lookup (FPGA)
16:     bv_mem[p] ← BV
17:     ur_mem[i] ← p
18:   end for
19:   chunk.|UR| ← p
20: end for
```

El Algoritmo 1 es el utilizado para almacenar un ruleset en un esquema RFC con segmentación, tal como el de la Fig. 5.9(a). Como consecuencia de la segmentación horizontal en múltiples bloques de memoria, los rangos requeridos $[UV.s, UV.e]$ son en general diferentes para cada uno de los $\lceil M/m \rceil$ segmentos o *chunks*. En las líneas 3-8 se recorren los 2^m posibles valores de key y se construyen los BVs correspondientes. Luego de construir cada BV, éste se compara en las líneas 9-12 con los BVs ya existentes, a fin de determinar si ese BV (combinación particular de UVs) corresponde a un URID existente. Ya que cada URID se relaciona bi-unívocamente con una combinación de UVs, si este URID ya existe simplemente se repite; de lo contrario, se genera un nuevo URID como se muestra en las líneas 13-15. Finalmente, los BV y UR generados se almacenan en sus respectivos arrays a fin de utilizarlos en posteriores iteraciones. Es importante mencionar que estos arrays se mantienen en la abundante memoria del GPP; al finalizar al procesamiento sólo el vector *ur_mem* de ancho reducido se transfiere a la memoria más limitada del FPGA. Otro aspecto importante, no considerado por el momento, es que este algoritmo contempla las N reglas aún para etapas de lookup; sin embargo como se analizó en el Cap. 4 muchas reglas comparten el mismo intervalo de valores para un campo, definiendo la cantidad de UVs $|UV| << N$ en ese campo. Esto se puede utilizar para disminuir las iteraciones y anchos de memoria involucrados en el algoritmo, requiriendo para ello que se compruebe previamente si el UV a agregar existe en el ruleset. Esto es una ventaja frente a la técnica basada en comparación de BVs, la cual escala según N (sin considerar las optimizaciones propuestas en el Cap. 4).

Algoritmo 2 Primer algoritmo propuesto para actualización en IND

Require: Límites de rango $[UV.s, UV.e]$ de la nueva regla, $chunk.|UR|$, $chunk.|UV|$, próximo URID disponible nxt_urid para todos los segmentos

```
1:  for chunk = 0 to ⌈M/m⌉ do
2:    p ← chunk.|UR|
3:    v ← chunk.|UV|
      // incorporar el rango del nuevo UV al BV
4:    for i = UV.s to UV.e do
5:      BV ← bv_mem[i] or 2^(v+1)
        // comprobar si el BV ya existe para otro valor de key
6:      j ← uv.start
7:      while j ≤ i and BV ≠ bv_mem[j] do
8:        j ← j + 1
9:      end while
        // si no se encontró el BV, incrementar el contador de URs
10:     if j ≥ i then
11:       p ← p + 1
12:       ur_mem[p] ← nxt_urid
13:     end if
14:     bv_mem[p] ← BV
15:   end for
16:   chunk.|UR| ← p
17: end for
```

Consideremos ahora el caso de actualización incremental. En ésta, $|UV|$ puede aumentar o no según el intervalo introducido esté o no ya presente en el ruleset. Sin embargo, ya que el BV generado al combinar el nuevo UV con los existentes no se conoce *a priori*, se debe ejecutar completamente el algoritmo de almacenamiento para cada actualización. En el Algoritmo 2, en cambio, se introducen tres mejoras principales:

- se optimiza el ancho del array *bv_mem* considerando $|UV|$ en lugar de N, no sólo para la etapa de lookup sino para el segmento considerado dentro de ella. Para ello, se mantiene un contador v de UVs.
- se recorre *sólo el intervalo asociado al nuevo UV* $[UV.start, UV.end]$ (línea 4) en lugar del espacio completo de búsqueda $[0, 2^m - 1]$. Dependiendo del scope del nuevo UV, esto puede ahorrar gran cantidad de iteraciones.
- el valor $|UR|$ se mantiene en un contador p. Los URIDs, en tanto, deben mantenerse individualmente ya que pueden ser no consecutivos, por lo que que se almacenan en una FIFO de etiquetas disponibles. Esto permite gestionar eficientemente la re-utilización de URIDs y conocer en forma separada la cantidad de URIDs en uso.

En la Fig. 5.15 se ilustra la implementación de este algoritmo para un ruleset 1D inicialmente formado por los UV1, UV2, UV3 y UV4, el cual se actualiza mediante la suma de UV5. Mediante este esquema se reduce el ancho de memoria *bv_mem de* $O(N)$ *a* $O(|UV|)$ mientras que la complejidad en tiempo puede variar desde $O(1)$ para UVs basados en EX hasta $O(2^m)$ para UVs basados en WC; si

Algoritmo 3 Segundo algoritmo propuesto para actualización en IND

Require: Limites de rango $[UV.s, UV.e]$ de la nueva regla, $chunk.|UR|$, $chunk.|UV|$, proximo URID disponible p y ERID q para todos los segmentos

```
1: for chunk = 0 to ⌈M/m⌉ do
2:    ER_A ← er_mem[UV.s]
3:    ER_B ← er_mem[UV.e]
      // actualizar memorias de regiones y de limites
4:    for i = UV.s to ER_A.e do
5:       ur_mem[i] ← p
6:       er_mem[i] ← q
7:       bnd_mem[i] ← UV.s, ER.e
8:    end for
9:    for i = ER_B.s to UV.e do
10:      ur_mem[i] ← p
11:      er_mem[i] ← q
12:      bnd_mem[i] ← UV.s, ER.e
13:   end for
14: end for
```

bien esto representa una importante mejora respecto al algoritmo original, aún puede ser ineficiente en el peor caso.

Mediante el Algoritmo 3 y la Fig. 5.16 se muestra nuestra segunda propuesta para actualización de IND, basada en el análisis de complejidad realizado en la Sec. 5.3 y la Fig. 5.4. Veremos a continuación que esta modificación puede acelerar notablemente el proceso de actualización de esquemas de búsqueda basados en URs (IND). Como se demostró en las Figs. 5.3 y 5.4, no más de dos nuevas URs son generadas por una operación de actualización; estas URs pueden encontrarse al comienzo o fin del nuevo intervalo por lo que podemos afirmar que ellas bien pueden comenzar en el extremo $UV.s$ o finalizar en el extremo $UV.e$. Por ejemplo, en los casos 1) y 4) de la Fig. 5.4(a) y el caso 1) de la Fig. 5.4(b) se genera sólo una nueva UR, caracterizada por la presencia exclusiva de UV3, mientras que en los casos 2) y 3) de la Fig. 5.4(a) y los casos 2), 3) y 4) de la Fig. 5.4(b) se introducen dos nuevas URs definidas por combinaciones de UV1, UV2 y UV3. Los extremos opuestos de las nuevas URs, en tanto, están determinados por los extremos de las URs existentes con anterioridad en la zona afectada. Estas observaciones pueden explotarse para reducir la complejidad de actualización. Para ello, se realizan inicialmente dos lookups independientes en los puntos $UV.s$ y $UV.e$ respectivamente, los cuales de acuerdo a la Fig. 5.3 resultan en no más de dos URs diferentes que denotaremos como UR_A y UR_B. Los URIDs de ambas se reemplazarán por nuevos URIDs sólo en los intervalos $[UV.s, UR_A.e]$ y $[UR_B.s, UV.e]$. En general, existirán también URs para las cuales se cumple $((UR.s > UV.s) \wedge (UR.e < UV.e))$, por lo que resultarán totalmente abarcadas por el nuevo UV; sin embargo sus respectivos URIDs no son afectados por el nuevo UV ya que sus intervalos de acción permanecen esencialmente inalterados. Un ejemplo de tal caso sería UR3 en la Fig. 5.15, definida por el solapamiento de UV1 y UV3 y totalmente abarcada por el nuevo intervalo de UV5. En general, los intervalos afectados $[UV.s, UR_A.e]$ y $[UR_B.s, UV.e]$ serán mucho menores que el intervalo del nuevo UV $[UV.s, UV.e]$, por lo que se logra una mejora. Ésta se ponderará en la sección de resultados.

En la Fig. 5.16 se muestra la arquitectura implementada para nuestra segunda propuesta. Luego

de realizar lookup en los extremos $UV.s$ y $UV.e$, se accede a una memoria de extremos *bnd_mem*, la cual sirve para acotar la cantidad de iteraciones en la línea 4 del Algoritmo 2. En este punto, se debe observar un problema a resolver para que esta modificación trabaje correctamente. Haciendo referencia a las Figs. 5.4(a) y 5.4(b), vemos que en los casos 1), 2) y 3) las regiones UR_A, UR_B y sus respectivos rangos $[UR.s, UR.e]$ resultan bien definidos al realizar los lookups respectivos con $UV.s$ y $UV.e$ respectivamente; sin embargo, en el caso 4) de la Fig. 5.4 surge el problema de que ambas URs resultan iguales ($UR_A = UR_B = 3$) aun cuando sus rangos asociados $[UR.s, UR.e]$ son distintos. Esto demuestra que una UR puede implicar múltiples intervalos $[s, e]$, siempre que esos intervalos involucren el mismo solapamiento de UVs que caracteriza bi-unívocamente a cada UR. En el caso de la Fig. 5.16 esta situación se produce al realizar lookup en *ur_mem* para $UV.s = 6$ y $UV.e = 10$, donde resultan $UR_A = UR_B = 2$. Si al acceder a *bnd_mem* se obtuviera la misma tupla $[UR.s, UR.e]$ para ambos casos no sería posible implementar nuestra propuesta. Para resolver este problema, podemos almacenar múltiples tuples $[UR.s, UR.e]$ en cada posición de *ur_mem* y luego resolver colisiones, lo cual requeriría lógica adicional; o almacenar una cantidad mayor de regiones para cubrir las posibles colisiones. Este último método es el adoptado mediante lo que llamamos *regiones expandidas (Expanded Regions, ERs)*, almacenadas en un segundo bloque de memoria *er_mem*. Las regiones expandidas consideran el peor caso de $|UR| = 2|UV| - 1$, de este modo es posible comprobar que cada ER estará bi-unívocamentte relacionada con una única tupla $[ER.s, ER.e]$ solucionando el problema a un costo acotado y conveniente como veremos más adelante. En el ejemplo de la Fig. 5.16, el lookup en *er_mem* arroja $ER_A = 2$ y $ER_B = 4$, las cuales están asociadas a los intervalos $[5, 6]$ y $[10, 11]$ respectivamente. Ya que $UV5 = [6, 10]$, podemos determinar los intervalos a actualizar como $[UV.s, ER_A.e] = [6, 6]$ y $[ER_B.s, UV.e] = [10, 10]$ en lugar de todo el intervalo $[0, 2^m]$ de RFC (Algoritmo 1) o aún el intervalo $[UV.s, UV.e]$ del Algoritmo 2.

Tanto *er_mem* como *bnd_mem* se utilizan para propósitos de actualización; sin embargo sólo *ur_mem* se almacenará en la memoria del sistema de lookup por lo que nuestro esquema no afecta las características de desempeño originales de RFC. Aún más, la memoria *bv_mem* utilizada por RFC durante la actualización es ahora reemplazada por la combinación de *er_mem* y *bnd_mem*. Estas memorias escalan como $O(2^m \cdot log_2(2|UV|))$ y $O(4 \cdot m \cdot |UV|)$ respectivamente, mientras que *uv_mem* escala como $O(|UR| \cdot N)$ (o en el mejor de los casos $O(|UR| \cdot |UV|)$), por lo que se reduce el consumo de memoria, tal como se demostrará en la Sec. 5.6.

Nuestra propuesta puede combinarse con las de [108] para obtener un esquema de búsqueda altamente optimizado. Las contribuciones de [108] son: i) se reduce el ancho de operaciones AND para comprobar $BV \neq bv\_mem[j]$ en la línea 7 del Algoritmo 2 mediante el uso de vectores agregados (Aggregated Bit Vectors, ABVs), ii) se reduce el número de iteraciones de las líneas 4-15 en el Algoritmo 2 mediante el acceso simultáneo a múltiples palabras de *bv_mem* a través de hashing, y iii) se comprime el espacio de direccionamiento de *ur_mem* agrupando palabras que contienen el mismo URID. Nuestra propuesta elimina la necesidad de i) y ii) ya que se prescinde del bloque de memoria *bv_mem*; inclusive nuestra solución es más simple y determinística ya que no genera falsos positivos como lo hace ABV ni requiere el uso de hashing. La contribución iii) de [108], en tanto, es una técnica efectiva para reducir en particular *ur_mem*, por lo que se adoptará como complemento a nuestra solución.

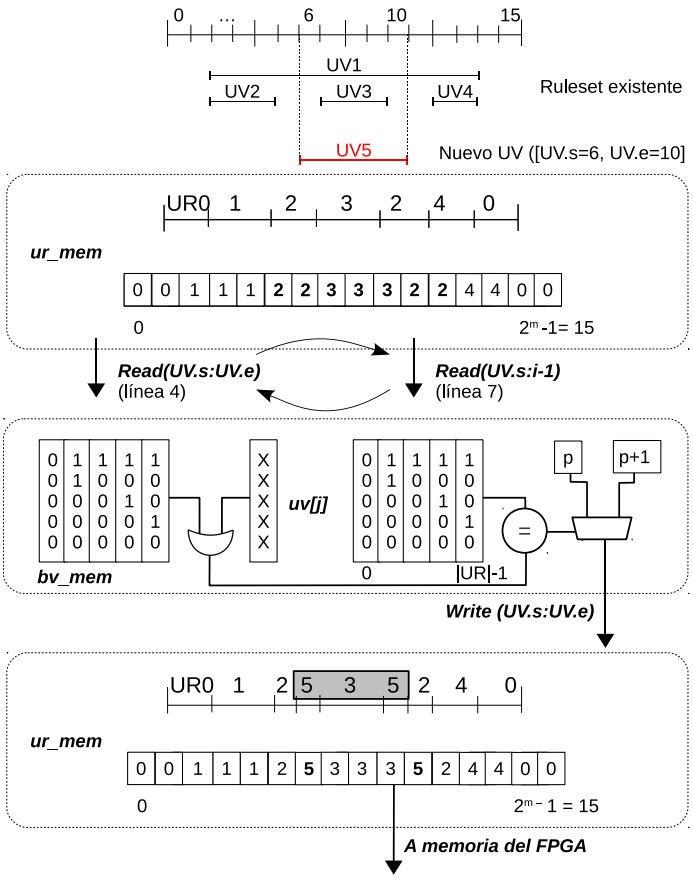


Figura 5.15: Primer esquema de actualización propuesto para búsqueda IND (metadato: etiqueta UR)

Aun cuando estas mejoras disminuyen el pre-procesamiento necesario para actualizar esquemas basados en URs, desde el aspecto tecnológico persiste el inconveniente de requerir múltiples ciclos para actualizar *ur_mem* en la memoria del FPGA. Esto se puede mitigar observando que *ur_mem* normalmente utiliza modos profundos y angostos ya que su ancho escala como $O(log_2|UR|)$ en lugar de $O(N)$ de *bv_mem*. Por otro lado, las memorias BRAM pueden explotar configuraciones mixtas de puertos, de modo que se pueden escribir múltiples palabras seleccionando modos de lectura profundos/angostos y modos de escritura poco profundos/anchos. Por ejemplo, la arquitectura de la Fig. 5.10 podría utilizar modo de lectura $4K \times 4$ y de escritura 512×40, escribiendo 10 palabras (40/4) por ciclo y soportando $2^4 = 16$ URs con un solo bloque de memoria o $2^8 = 256$ URs con dos bloques. Este aspecto, en conjunto con el estudio de scopes típicos de ruleses reales [134], se pueden utilizar para establecer compromisos en la elección del factor de los modos de memoria a utilizar.

5.6. Resultados

En esta sección se evalúan las arquitecturas de lookup consideradas (IND, BS y ERM), comparando resultados de estimación con los de síntesis en FPGAs. Como resumen de nuestro análisis

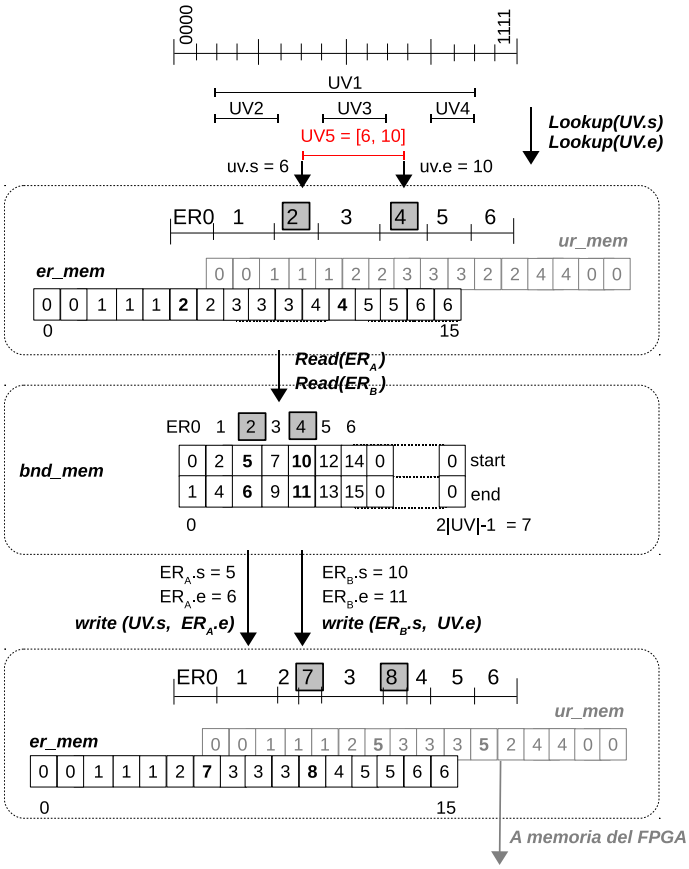


Figura 5.16: Segundo esquema de actualización propuesto para búsqueda IND (metadato: etiqueta UR)

previo, y a fin de introducir la presente evaluación, se presentan en el Cuadro 5.1 las estimaciones de costo para cada caso involucrado. Las métricas analizadas son almacenamiento (memoria/registros), consumo de lógica combinacional y ancho de memoria (Mem BW). Los metadatos generados, que constituyen la interfaz entre la etapa de lookup y la etapa de agregación en esquemas de clasificación por agregación, se indican entre paréntesis (BV) o (UR). Se consideran asimismo los costos de posibles etapas de adaptación de metadatos; para el caso particular $BV \rightarrow UR$, que representa la contraparte del codificador de prioridad para un caso general MM, no se proveen estimaciones ya que su implementación es considerada impráctica.

Cuadro 5.1: Estimación de costos de los esquemas de lookup considerados

Esquema de lookup	Memoria	Lógica comb.	Mem BW
Logic-based ERM (BV) [119]	$O(2 \cdot N \cdot M)$	$O(3 \cdot N \cdot M)$	$O(2 \cdot N \cdot M)$
(opcional) post-encoding BV $\rightarrow$ UR	–	–	–
IND, no stitching (UR)	$O(2^M \cdot log\|UR\|)$	$O(0)$	$O(log\|UR\|)$
(opcional) post-encoding UR $\rightarrow$ BV	$O(\|UR\| \cdot N)$	$O(0)$	$O(N)$
IND, no stitching (BV)	$O(2^M \cdot N)$	$O(0)$	$O(N)$
IND, BV-based horizontal stitching (BV) [135]	$O(\lceil M/m \rceil \cdot (2^m \cdot log\|UR\| + \|UR\| \cdot N))$	$O(\lceil M/m \rceil \cdot N)$	$O(N)$
IND, UR-based horizontal stitching (UR) [106]	$O(2 \cdot 2^m \cdot log(\|UR\|/2) + 2^{2 \cdot log(\|UR\|/2)} \cdot log\|UR\|)$	$O(0)$	$O(log\|UR\|)$
IND, vertical stitching (UR) [145]	$2^X \cdot 2^{M-X} \cdot log\|UR\|$	$O(2^X \cdot (1 + log\|UR\|))$	$O(log\|UR\|)$
Binary search (UR) [105]	$O(M \cdot \|UR\| + \|UR\| \cdot log\|UR\|)$	$O(M \cdot log\|UR\|)$	$O(log\|UR\|)$
(opcional) post-encoding UR $\rightarrow$ BV	$O(\|UR\| \cdot N)$	$O(0)$	$O(N)$
Binary search (BV)	$O(M \cdot \|UR\| + \|UR\| \cdot N)$	$O(M \cdot log\|UR\|)$	$O(N)$

5.6.1. Esquemas de direccionamiento de memoria (IND)

En primer término se consideran esquemas IND por separado, ya que constituyen una generalización de la emulación de TCAM planteada en el Cap. 3. Sin pérdida de generalidad, consideraremos un caso de 32 reglas, el que es un límite típico para el ancho x de BRAM en FPGAs modernos, y ancho de key $M = 128$. En la Fig. 5.17(a) se ilustra el consumo de memoria para diferentes modos de BRAM M20K disponible en FPGAs Altera Stratix V. Se considera que los rangos son soportados mediante expansión de direccionamiento ($w < m$), propagándose sólo un bit por regla. Los bloques M20K pueden configurarse en modos desde 20Kx1 hasta 512x40; para este caso consideramos rangos y anchos que sean potencias de 2 (modo 16Kx1 o factor de forma 14/1, hasta modo 512x32 o factor de forma 9/32). Para un factor de forma dado m/x, el espacio de direccionamiento 2^m define los rangos posibles mientras que el ancho de memoria define la cantidad de UVs N o la cantidad de URs $|UR|$ según se adopten metadatos basados en bitmaps de UVs (bit vector) o etiquetas de URs. Como se observó en al Cap. 3, para rangos de bajo scope o PX/EX la mejor opción es el modo 512x32, mientras que para rangos $9 < w < 15$ se pueden considerar modos más profundos y angostos a costo de menor eficiencia en el uso de memoria. Para un determinado rango requerido, en tanto, surge la necesidad de analizar los modos que minimizan el desperdicio de memoria; para ello se define un *factor de utilización* μ. Como se observa en la Fig. 5.17(b), un rango determinado se implementa más eficientemente por el modo que soporta el rango inmediato superior. Por ejemplo, un rango $w = 5$ es soportado más eficientemente por el modo 512x32 que por modos más profundos y angostos.

En la Figs. 5.18(a) y 5.18(b) se explora el punto de vista opuesto, es decir, para un determinado soporte de rango w, cómo afecta el ancho de key M en el caso de una regla ($N = 1$). Se analizan dos casos hipotéticos $w = 1, w = 5$ así como casos $w = 9, w = 14$ que realmente afectan el factor de forma utilizado. En la Fig. 5.18(a) se observa que el consumo de memoria asciende en pasos cuando M crece en múltiplos enteros de w; mientras que los casos que soportan mayor w presentan mayor consumo intrínseco de memoria para el mismo ancho de key M. En el caso extremo de $w = 14$ y $M = 128$ se requieren alrededor de 4 Mbits de memoria para una regla, mientras que para $w = 9$ y el mismo $M = 128$ este consumo desciende a 40 Kbits. Ya que los resultados de los segmentos individuales deben ser agregados mediante un esquema de pipelining, resulta asimismo de interés la *latencia* introducida en los casos de la Fig. 5.18(a); esta es evaluada en la Fig. 5.18(b) para distintos rangos y anchos de key. Al contrario que el consumo de memoria, la latencia disminuye para mayores w, esto es así ya que cada bloque de BRAM abarca más bits del key. Por ejemplo la implementación con bloques de $w = 1$ requiere 128 ciclos para $M = 128$, mientras que el caso de bloques con $w = 9$ (modo 512x32) presenta $\lceil 128/9 \rceil = 15$ ciclos de latencia. Se debe recordar que, aún con estas latencias, se obtiene un nuevo resultado por ciclo de reloj.

De las figuras anteriores, podemos concluir que el modo 512x32 puede ser la mejor opción de compromiso entre consumo de memoria y latencia, si bien no es capaz de soportar rangos $w > 9$ sólo mediante expansión de direccionamiento. Según lo visto en la Fig. 5.14, se pueden soportar rangos mayores expandiendo en ancho; esta expansión no es igual a la necesaria en TCAMs sino que se limita a 3 pseudo-reglas por cada regla original. De este modo, se puede utilizar el modo más eficiente 512x32 a costa de mayor ancho de memoria por regla (se soportan $\lfloor 32/3 \rfloor = 10$ reglas

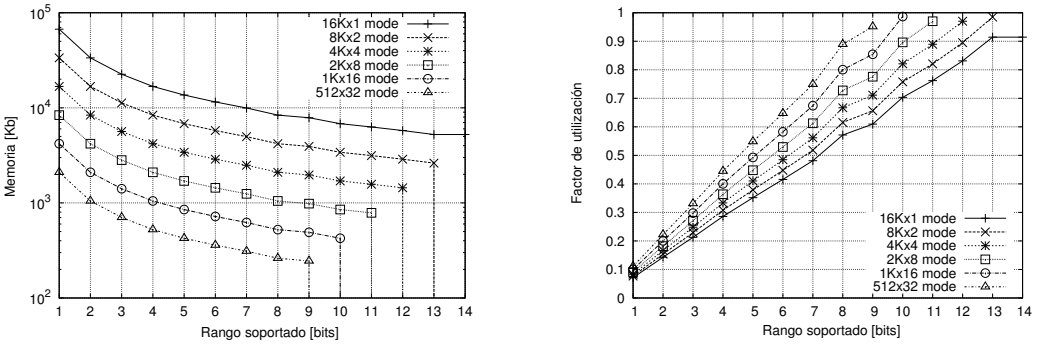


Figura 5.17: Soporte de rangos mediante expansión de direccionamiento ($M = 128$, $N = 32$): (a) consumo de memoria, (b) factor de utilización μ

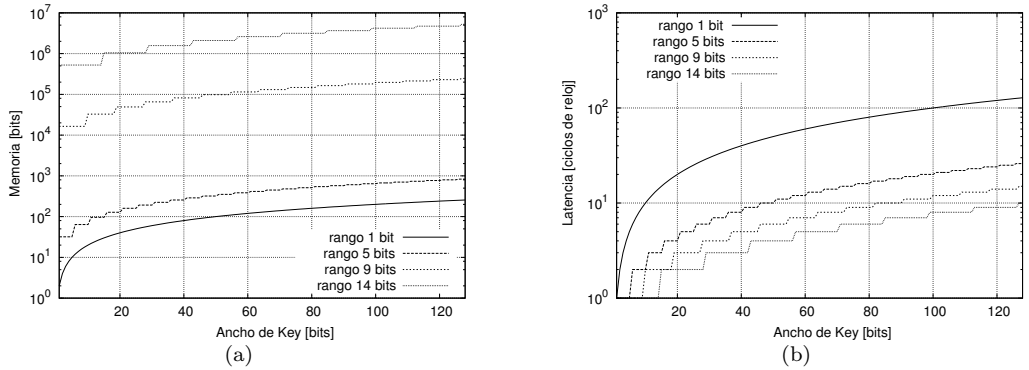


Figura 5.18: Impacto del ancho de key ($m = 9$, $N = 1$): (a) consumo de memoria, (b) latencia

por BRAM). Otra alternativa es implementar IND con metadatos basados en URIDs tal como se muestra en la Fig. 5.9(a).

5.6.2. Comparación de esquemas IND, BS y ERM

A continuación se comparan entre sí los esquemas planteados; para ello se consideran las complejidades obtenidas en el Cuadro 5.1, todas ellas ofreciendo procesamiento a velocidad de línea. De las alternativas mencionadas en este cuadro, nos concentraremos en las siguientes (líneas sombreadas):

1. ERM basado en lógica combinacional (metadato de salida: bitmap BV)
2. IND segmentado horizontalmente (stitched IND) con agregación intra-campo por BVs (metadato de salida: bitmap BV)
3. IND segmentado horizontalmente (stitched IND) con agregación intra-campo basada en URs (metadato de salida: etiqueta UR)
4. Árbol de decisión binario (BS) (metadato de salida: etiqueta UR)

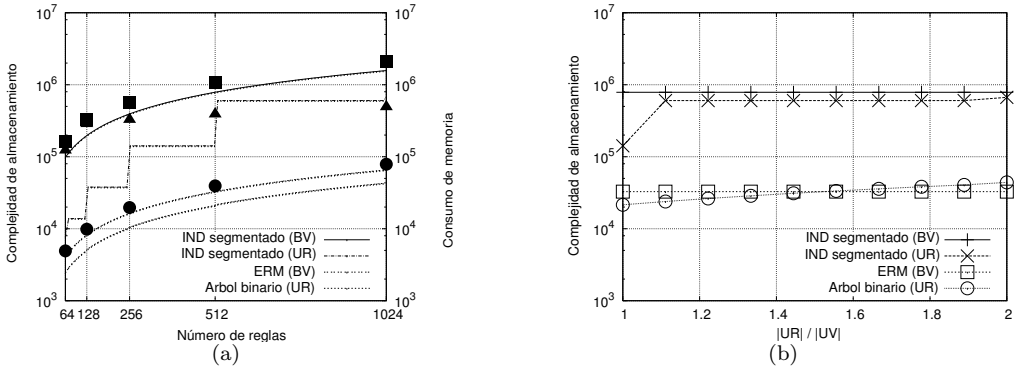


Figura 5.19: Complejidad de almacenamiento: (a) $|UR| = |UV| = N$, $64 \leq N \leq 1024$, (b) $|UV| = N = 512$, $|UV| \leq |UR| \leq 2.|UV|$

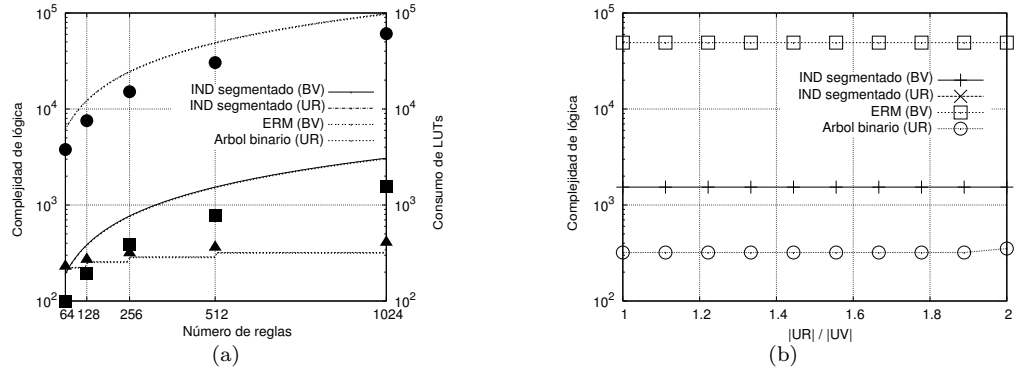


Figura 5.20: Complejidad lógica: (a) $|UR| = |UV| = N$, $64 \leq N \leq 1024$, (b) $|UV| = N = 512$, $|UV| \leq |UR| \leq 2.|UV|$

Se obtienen además resultados post-fitting de síntesis para los casos (1), (2) y (4). La arquitectura de (3) se basa exclusivamente en direccionamiento de memoria sin presentar dificultades particulares, por lo que no se analiza su implementación. En general, se puede observar que (1) y (2) son más predecibles ya que dependen de N; sin embargo su implementación plantea desafíos para obtener el mejor desempeño posible. (3) y (4) son más sensibles al patrón específico de reglas por lo que se considerará un caso medio donde $|UR| = |UV| = N$. La principal ventaja de (3) y (4) radica en su mejor escalabilidad con N. (4) reduce la expansión de memoria 2^w de (3) a $2^m = |UR|_{max} = 2|UV|$; sin embargo introduce mayor latencia y desperdicio de memoria debido al desbalance de consumo entre niveles del árbol. Debido a estos compromisos de diseño, la implementación de (4) se incluye en este análisis.

En la Fig. 5.19(a) se comparan los consumos estimados para $M = 32$, $m = 9$, $|UR| = |UV| = N$, y $64 \leq N \leq 1024$. Para el caso de (3), que es esencialmente RFC aplicado al caso de agregación intra-campo, se considera agregación de a pares de segmentos mediante múltiples instancias del esquema de la Fig. 5.9(a). Para el caso considerado $M = 32$, $m = 9$, se utilizan $\lceil 32/9 \rceil = 4$ memorias en la primera etapa de agregación, 2 bloques en la segunda y uno en la tercera obteniendo el metadato final basado en etiquetas UR. Según observaciones en rulesets reales [106], se considera

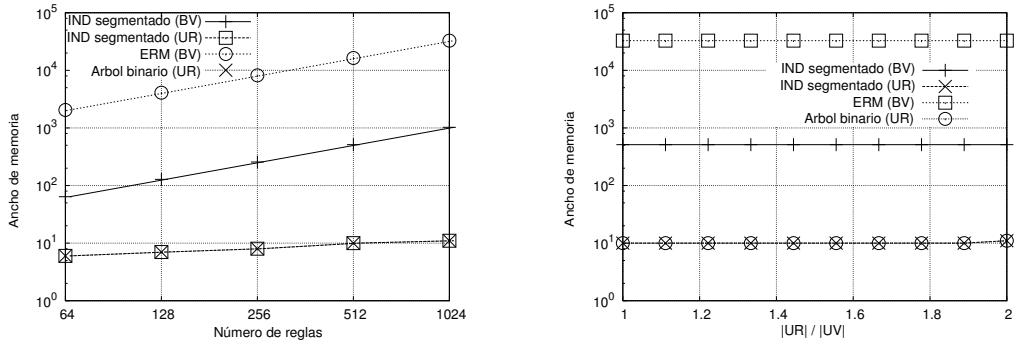


Figura 5.21: Ancho de memoria: (a) $|UR| = |UV| = N$, $64 \leq N \leq 1024$, (b) $|UV| = N = 512$, $|UV| \leq |UR| \leq 2.|UV|$

que $|UR|$ se duplica de una etapa a otra; por ejemplo si $|UR| = 256$ para $M = 32$ y el campo se divide en 4 segmentos, tendremos $256/4 = 64$ regiones para la primera etapa y $256/2 = 128$ regiones para la segunda etapa. Como se observa, el esquema BS (UR) presenta el menor consumo de memoria seguido de ERM (UR); esto es ya que BS explota la relación $N \leq |UR| \leq 2N$ que en este caso se fija en $|UR| = N$, requiriendo sólo N límites de M bits cada uno; mientras que ERM escala indefectiblemente como $O(2 \cdot N \cdot M)$ sin importar el valor de $|UR|$. Es de destacar que el almacenamiento en ERM se implementa exclusivamente mediante registros ya que se debe acceder a todos ellos concurrentemente, mientras que BS y IND pueden utilizar memoria o combinaciones de ésta con registros. El esquema IND (BV), en tanto, presenta el mayor consumo de memoria debido a las contribuciones de la expansión de direccionamiento y el metadato de ancho N.

En la Fig. 5.19(a) se muestran asimismo en puntos llenos los resultados post-fitting de síntesis para FPGA. En principio se nota que IND (BV) presenta consumo algo mayor al estimado (cuadrados llenos), esto es debido a que el factor de forma utilizado 9/40 tiene granularidad algo más gruesa que el 9/32 estimado. El consumo de registros para ERM (BV) (círculos llenos), en tanto, sigue fielmente la complejidad estimada. Para el caso BS (UR), se notan claramente los efectos del desbalance de memoria en nuestra arquitectura genérica, ya que los bloques BRAM se utilizan ineficientemente en los primeros niveles del árbol. Si bien esta implementación general es conveniente para nuestros propósitos, las optimizaciones introducidas en [133] pueden aproximarse al valor estimado a costa de bastante mayor complejidad de implementación.

En la Fig. 5.19(b) se analiza el consumo de memoria al variar la relación $|UR|/|UV|$ en el rango $1 \leq |UR|/|UV| \leq 2$, asumiendo $|UV| = N = 512$. Los resultados obtenidos confirman nuestra discusión previa; tanto Stitched IND (BV) como Stitched IND (UR) tienen alto requerimiento de memoria mientras que ERM (BV) y BS (UR) presentan consumo por debajo de un orden de magnitud; la relación entre ellos varía a su vez según la relación $|UR|/|UV|$. Además, como se observa, los esquemas que utilizan BV como metadato tienen mucho menor posibilidad de aprovechar esta relación que los que utilizan etiquetas UR.

En la Fig. 5.20(a) se consideran los recursos de lógica combinacional con respecto a variaciones en $N = |UV| = |UR|$. Para el caso de implementación, la métrica considerada es el consumo de LUTs. ERM presenta uso intensivo de lógica, mientras que los demás esquemas desplazan el cómputo a la etapa de actualización requiriendo mucha menos lógica. BS consume poca lógica para comparación de magnitud, mientras que IND (UR) no requiere lógica combinacional alguna. En la Fig. 5.20(b), en tanto, se analiza este consumo para $|UR|/|UV|$ variable. ERM en este caso consume recursos máximos y constantes sin importar $|UR|/|UV|$, mientras que BS logra un punto medio entre el consumo nulo de IND (UR) y el consumo de IND (BV).

Finalmente, las Figs. 5.21(a) y 5.21(b) muestran que ERM mantiene máximo requerimiento de ancho de memoria para las variables consideradas por su arquitectura basada en registros. Los esquemas que entregan resultados tipo URID presentan el mínimo ancho de memoria, mientras que IND (BV) se ubica en un punto medio.

De los resultados obtenidos, se destaca que ERM no sufre de expansión de direccionamiento; sin embargo sus requerimientos de lógica combinacional son elevados y se debe diseñar cuidadosamente el esquema de pipelining utilizado para no incurrir en elevados retardos de ruteo en el FPGA. Estos factores pueden a su vez significar consumo de energía elevado. Los requerimientos de ERM escalan con N sin importar la relación $|UR|/|UV|$, esta desventaja es heredada de la arquitectura TCAM/ETCAM. Aun mas; si bien no se considera aquí, ERM requiere necesariamente de un codificador de prioridad (BV-to-UR mapping en la Fig. 5.12(a)) para que los resultados de lookup o clasificación sean utilizables; esta etapa es también heredada de la arquitectura TCAM y su desempeño es crítico para el esquema de búsqueda. Los esquemas basados en direccionamiento de memoria, en tanto, pueden escalar mejor con N mediante la explotación de la relación $|UR|/|UV|$; su expansión de direccionamiento se mitiga mediante esquemas de segmentación o compresión del espacio de direccionamiento a través de BS.

5.6.3. Actualización incremental de esquemas IND

En esta sección se evalúa la efectividad de las técnicas de actualización propuestas para lookup basado en IND (UR). Sobre los Algoritmos 2 y 3, así como los esquemas de las Figs. 5.15 y 5.16, y las referencias [106] y [108], se evalúan requerimientos de espacio y tiempo de trabajos previos contra las propuestas.

En primer término, evaluamos qué tan costosa puede ser *er_mem* con respecto a *ur_mem*. Para ello, en la Fig. 5.22(a) se comparan sus consumos de memoria para $|UV| \leq |UR| \leq 2|UV|$. En este análisis se considera $|UV| = N$ ya que el algoritmo de actualización no comprueba si el nuevo intervalo ya existe sino que simplemente agrega un nuevo bit por UV a *uv_mem* (línea 5 del Algoritmo 2); se consideran tres casos $N = 64, 128, 256$. Como se comprueba, ya que *er_mem* escala como el logaritmo del peor caso de regiones $|UR| = 2|UV| = 2N$, ésta tiene un consumo tolerablemente más alto que aquél de *ur_mem*. Sin embargo, *er_mem* permite utilizar el bloque *bnd_mem*, que contiene sólo $2|UV|$ tuplas de m bits eliminando la necesidad del bloque *bv_mem* que requiere $|UV| \leq |UR| \leq 2|UV|$ palabras de N bits cada una. De este modo, las memorias *ur_mem*, *er_mem* y *bnd_mem* escalarán

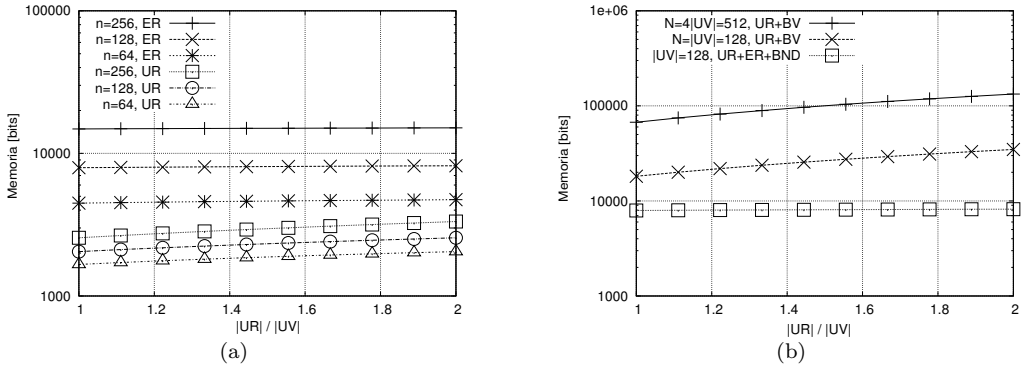


Figura 5.22: Actualización de esquemas IND: (a) consumo de memoria para *ur_mem* vs. *er_mem*, (b) consumo de memoria total para nuestra propuesta

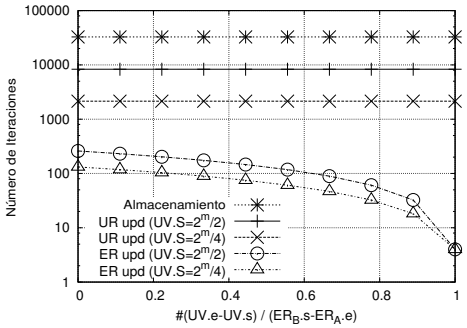


Figura 5.23: Actualización de esquemas IND: iteraciones requeridas

como $2^m log_2 N$, $2^m log_2(2N)$ y $2Nm$ respectivamente. Los consumos totales respectivos se analizan en la Fig. 5.22(b), donde se demuestra que utilizar memorias *ur_mem + er_mem + bnd_mem* puede ser efectivamente más económico que sus contrapartes *ur_mem + bv_mem*. Además, ya que *bv_mem* escala con N en lugar de $|UV|$, su consumo es en relación aún mayor ya que el esquema propuesto mantiene su costo son respecto a la relación $|UV|/N$. Esto se ilustra para dos casos distintos $N = |UV|$ y $N = 4|UV|$.

En segundo término, interesa evaluar la ganancia obtenida en ciclos de cómputo al considerar sólo el intervalo $[UV.s, UV.e]$ (Algoritmo 2) para la actualización en lugar del intervalo completo de almacenamiento $[0, 2^m]$ (Algoritmo 1), así como los intervalos de actualización por ERs $[UV.s, UR_A.e]$ y $[UR_B.s, UV.e]$ (Fig. 3) en lugar del intervalo original por URs $[UV.s, UV.e]$ (Algoritmo 2). Como se muestra en la Fig. 5.23, el algoritmo de almacenamiento tiene máximo costo constante, por lo que su ejecución debería limitarse al momento de creación de la tabla de lookup. Asimismo, se consideran los tiempos de actualización para esquemas basados en URs y ERs para dos casos de intervalos (Scopes S) $UV.S = 2^m/2$ (la mitad del espacio de búsqueda) y $UV.S = 2^m/4$ (un cuarto del espacio de búsqueda); a fin de mostrar las ventajas del primer algoritmo propuesto. Finalmente, se consideran distintas relaciones entre el scope del nuevo UV ($UV.e - UV.s$) y el scope de ERs existentes ($ER_B.s - ER_A.e$), a fin de demostrar la ventaja del segundo algoritmo de actualización propuesto. Se debe notar que el *Offset O* particular no es de relevancia para este análisis, siempre

que se cumplan las condiciones $UV.s \leq UR_A.e$ y $UV.e \geq UR_B.s$.

5.6.4. Resultados de síntesis en FPGA

En los Cuadros 5.2, 5.3 y 5.4 se observa el reporte de síntesis post-fitting con balance de área vs. velocidad para los tres esquemas principales IND (BV), ERM (BV) y BS (UR) implementados. Las arquitecturas hardware fueron descriptas en Verilog HDL y simuladas mediante la herramienta *Mentor ModelSim*; la síntesis se realizó mediante *Quartus II* de Altera utilizando la herramienta de análisis estático de tiempos *Timequest* para comprobación y optimización de caminos críticos. El dispositivo utilizado es un FPGA Altera Stratix V SGXMB6R2F43C2, con rango comercial de temperatura y speed grade 2 (fast, 1=fastest). Se evaluaron anchos de key $M = 16, 32, 128$ y número de reglas $N = 256, 512, 1024$ (se considera $N = |UV|$). Confirmando lo analizado en nuestras estimaciones, IND presenta mínimo consumo de LUTS y Registros basando su desempeño en el uso de memoria BRAM. ERM utiliza abundantes registros para almacenamiento de reglas y pipelining, y LUTs para comparadores de magnitud. BS, en tanto, utiliza BRAM para *ctrl_mem*, LUTs para comparadores de magnitud, y registros para pipelining. Como se comentó anteriormente, los factores críticos de BS resultaron ser los comparadores de magnitud y el aprovechamiento de memoria BRAM. A fin de realizar una comparación equitativa con los demás esquemas, se buscó el comparador con el mejor desempeño posible. Luego de evaluar múltiples alternativas, se adoptó la primitiva *lpm_compare* ofrecida por Altera, la cual mostró la mejor escalabilidad. El consumo de BRAM es mayor al estimado ya que (a) la granularidad de BRAM es mucho menor que la requerida en los primeros niveles del árbol, y (b) el ancho requerido no es sólo M como se estimó sino la suma de los anchos de *key_value+next_or_urid+fnd* según el esquema de la Fig. 5.11(b).

Según lo recomendado en [133], se adoptó $m = 4,\ n = 40$ para ERM, mientras que en base a las observaciones de la Fig. 5.17 se seleccionó $m = 9, n = 40$ para implementar IND. Se observa que el desempeño de IND escala mejor con N y M que el de ERM. En base a este resultado y aplicando las optimizaciones de [108], se puede predecir que el desempeño de IND (UR) debería escalar aún mejor que IND (BV) por su ancho de memoria fuertemente reducido de $O(N)$ a $O(log|UR|)$; mientras que mediante las optimizaciones propuestas en este trabajo se mitigan sensiblemente sus problemas de complejidad de actualización. BS es otra alternativa interesante, ya que aún con su ineficiencia en el aprovechamiento de memoria el consumo total de BRAM es notablemente inferior al de esquemas IND. Queda por optimizar la escalabilidad de sus comparadores de magnitud, la cual es el mayor limitante de su desempeño (Mlps). Si bien no se considera en este trabajo, una posible optimización al respecto sería utilizar pipelining 2D. Esto demanda cierto análisis previo ya que, a diferencia de IND y ERM, los resultados de los últimos niveles son dependienes de aquéllos en los niveles superiores.

5.7. Conclusiones

En los últimos trabajos publicados sobre lookup y clasificación en FPGAs, tales como [118] y [119], se adoptan esquemas similares a ERM para solucionar definitivamente el problema de lookup AR. Sin embargo, todos ellos asumen implícitamente el uso de metadatos tipo bitmap basado en BV, los que presentan problemas de escalabilidad con N y carecen de flexibilidad alguna para explotar la relación $|UR|/|UV|$. Aún más, ninguno de estos esquemas tienen posiblilidad de explotar la relación $|UV|/N$, mientras que de hecho $|UV| << N$ para la mayoría de los rulesets reales [83] [134]. Por ejemplo para el ruleset *IPC1*, uno de los más complejos según [134], se tiene $N = 1550$, $|UV|_{SrcIP} = 152$ y $|UV|_{SrcPort} = 34$. Esta característica tiende a mantenerse válida para rulesets actuales y futuros [148]. Los esquemas basados en IND, en tanto, pueden utilizarse en conjunto con metadatos tipo bitmap (UV) o label (UR); en este último caso es donde se obtiene su máximo potencial; ya que los URIDs escalan como $O(log_2|UV|)$, la arquitectura resultante es altamente escalable. El costo por esta escalabilidad es mayor pre-cómputo para mapear UVs a URIDs y mayor consumo de memoria debido a la expansión de direccionamiento; estos factores fueron analizados exhaustivamente en este trabajo y en base a las conclusiones alcanzadas se proponen soluciones a ellos. Para obtener este tipo de metadatos, los esquemas ERM requieren una etapa adicional de mapeo, ésta es simplemente implementada por un codificador de prioridad en BM pero su implementación es excesivamente compleja para el caso general MM. Los esquemas IND, en tanto, pueden entregar directamente este metadato ya que es su método natural de búsqueda, por lo que se adaptan naturalmente a los requerimientos actuales de lookup para clasificación basada en flujos MM.

El trabajo detallado en el presente Capítulo dio lugar a la quinta contribución de la Tesis de Doctorado, el artículo de revista titulado *Optimization of lookup schemes for flow-based packet classification on FPGAs* publicado en el International Journal of Reconfigurable Computing del año 2015.

Cuadro 5.2: Síntesis de IND, segmentación horizontal por BV, pipeline 2D ($m = 9$, $n = 40$)

M	Reglas (N)											
	256				512				1024			
	M20K	LUTs	Regs	Mlps	M20K	LUTs	Regs	Mlps	M20K	LUTs	Regs	Mlps
16	14	260	108	589	26	533	216	592	52	1078	450	555
32	28	388	216	588	52	775	432	588	104	1543	900	514
128	105	1149	810	509	195	2280	1620	450	390	4627	3375	377

Cuadro 5.3: Síntesis de ERM, pipeline 2D ($m = 4$, $n = 8$)

M	Reglas (N)											
	256				512				1024			
	M20K	LUTs	Regs	Mlps	M20K	LUTs	Regs	Mlps	M20K	LUTs	Regs	Mlps
16	0	7721	9984	437	0	15433	19968	467	0	31235	39899	392
32	0	15145	19712	453	0	30464	39351	384	0	60902	78775	352
128	0	34845	77791	321	0	69815	155871	336	0	139676	312031	320

Cuadro 5.4: Síntesis de BS, pipeline 1D entre niveles del árbol ($N \leq |UR| \leq 2N$)

M	Reglas (N)											
	256				512				1024			
	M20K	LUTs	Regs	Mlps	M20K	LUTs	Regs	Mlps	M20K	LUTs	Regs	Mlps
16	8	179	1254	328	10	206	1463	320	13	235	1684	293
32	16	316	2038	246	19	361	2343	229	24	407	2660	245
128	32	1054	6742	191	39	1193	7623	172	53	1332	8516	174

Capítulo 6

Conclusiones

6.1. Resumen

La presente Tesis Doctoral aporta nuevas arquitecturas y criterios de diseño para el procesamiento eficiente de paquetes en redes de datos. Para ello, se analizan en primer término las características que presentan estas redes en la actualidad y las tendencias de su evolución. En base a este análisis, se definen módulos de procesamiento fundamentales y se consideran las opciones tecnológicas actuales para su implementación. En particular, se realizan implementaciones sobre tecnología de lógica programable (FPGAs), la cual es especialmente adecuada a esta aplicación. A partir de estas primeras implementaciones se identifican módulos críticos para el desempeño del sistema, concentrándose en especial en el módulo de *clasificación* de paquetes. Este módulo y sus desafíos de diseño son extensivamente analizados, arribando a una completa taxonomía que permite identificar áreas de investigación. Sobre esta base, se proponen nuevas aproximaciones al problema de clasificación especialmente orientadas a su eficiente implementación en dispositivos FPGA. Los resultados obtenidos no pretenden superar a los diseños existentes, que según la aplicación pueden resultar más o menos convenientes; sino que permiten explorar nuevas alternativas de diseño especialmente orientadas a satisfacer las necesidades de las redes futuras sobre tecnologías FPGA.

En términos generales, la evaluación de las arquitecturas planteadas se realizó para distintas complejidades de clasificación, considerando una implementación totalmente embebida en el dispositivo FPGA y un procesamiento de paquetes a velocidad de línea. La velocidad de las redes involucradas se define en base a bits por segundo (bps), encontrándose actualmente en el rango de 10^{11}bps (100 Gbps); mientras que la unidad de procesamiento involucrada es el paquete, cuyo tamaño mínimo se establece en 40 bytes. De este modo, se fijan distintas velocidades de operación mínimas según la máxima velocidad de transferencia de paquetes requerida.

6.2. Aportes realizados

El primer aporte de este trabajo, detallado en el Capítulo 2 del presente informe, consiste en la definición de un modelo de procesamiento de paquetes general que permite cubrir luego casos más específicos presentes en redes de datos reales. Mediante este modelo se define el concepto fundamental de *flujo* de paquetes, el cual permite establecer *caminos de procesamiento* en forma abstracta. Se identifican módulos de procesamiento fundamentales, llamados *primitivas*, a fin de analizar sus requerimientos computacionales y sus roles en el conjunto. Se implementan arquitecturas básicas para cada uno de ellos, ensayando luego su funcionamiento mediante una plataforma de red real de 1 Gbps. Asimismo, se comprueba mediante resultados de síntesis hasta qué punto estos módulos pueden escalar manteniendo la velocidad de operación requerida.

En base a esta primera experiencia se logran identificar los módulos críticos en cuanto a escalabilidad, seleccionando la primitiva de *clasificación* por su importancia para la evolución de las redes de datos. La complejidad de esta primitiva se asocia al problema de localización de un punto, que representa un paquete; entre múltiples objetos solapados multi-dimensionales, que representan reglas o filtros definidos en base a múltiples campos de búsqueda. El resultado a obtener consiste en los objetos intersectados por este punto, que representan los múltiples filtros coincidentes con el paquete. En cuanto a esta primitiva particular, se realizan aportes dentro de dos métodos existentes: memorias accesibles por contenido (TCAMs) y clasificación por descomposición. Las memorias TCAM, muy difundidas para ruteo en base a direcciones IP jerárquicas, brindan un método de muy alta velocidad para clasificación sin depender de las características del grupo de reglas (ruleset) utilizado; sin embargo, presentan serios problemas de escalabilidad con el tamaño (ancho y profundidad) del ruleset y no son adecuadas para búsqueda según rangos arbitrarios. A partir de estas observaciones sobre las TCAMs nativas, se brinda un análisis completo de las capacidades de un FPGA para emular TCAM mediante memorias RAM, mitigando sus problemas dentro de límites fijados. Se brinda así un criterio de diseño para seleccionar configuraciones adecuadas de memoria según los requerimientos del ruleset. Los resultados de implementación obtenidos demuestran que las TCAMs emuladas pueden soportar hasta 4K reglas de 128 bits cada una con rangos arbitrarios de hasta 9 bits, disminuyendo este tamaño al soportar rangos mayores de hasta 14 bits. Estos límites son establecidos principalmente por el tamaño de la memorias RAM incluidas en los FPGAs y los recursos de ruteo involucrados con sus retardos de propagación asociados. En particular, se muestran resultados de implementación para cuatro familias sucesivas de FPGAs, demostrando las posibilidades crecientes que ofrece esta tecnología para implementar casos de mayor complejidad manteniendo el desempeño requerido.

El segundo método, clasificación por descomposición, surge como una optimización de las memorias TCAM para el caso de clasificación sobre múltiples campos, y es especialmente adecuado para implementación en FPGAs por su natural concurrencia. Este método explota el hecho de que la búsqueda de coincidencias locales a cada campo presenta complejidad reducida respecto a la búsqueda en el espacio definido por todos los campos. Para ello implementa dos etapas sucesivas, una de *búsqueda* (lookup) en cada campo particular y otra para posterior *agregación* de los resultados obtenidos. Nuestro trabajo realiza aportes en ambas etapas, como se discute a continuación.

En cuanto a la etapa de agregación, se observó que los *metadatos* utilizados en trabajos previos son mayormente basados en *etiquetas* o en *bitmaps*; el primer caso requiere en general arquitecturas basadas en direccionamiento de memoria y extensivo pre-cómputo, lo que se traduce en alto consumo de memoria y compleja actualización incremental; mientras que el segundo utiliza ancho de palabra $O(N)$, donde N es la cantidad total de reglas, con agregación basada en simple concatenación de resultados. Se propone entonces utilizar un formato intermedio, que permita relajar los requerimientos de memoria y pre-cómputo de los esquemas basados en etiquetas mientras que a la vez reduzca el ancho de palabra requerido por los esquemas basados en bitmaps. Para realizar a agregación basada en este formato, se diseña una arquitectura basada en pipelines sistólicos, especialmente adecuados para implementación en FPGAs. Se exploran diferentes alternativas respecto a granularidad de agregación y recursos computacionales utilizados. Como resultado, se obtiene una serie de arquitecturas que explotan tanto los recursos de lógica combinacional como los recursos de memoria del FPGA, a diferencia de propuestas anteriores que utilizan mayormente uno de ambos. De esta forma, los diseños propuestos pueden adaptarse a los recursos disponibles en una sistema que involucre otros módulos en el mismo dispositivo. En particular, la actualización incremental del ruleset para el esquema de agregación propuesto es directa, por lo que su demanda de pre-cómputo es muy baja.

Respecto a la etapa de búsqueda (lookup), se observó que los trabajos previos en clasificación por descomposición no consideran en general su interacción con la etapa de agregación, utilizando esquemas de lookup sin justificación concreta de su elección. Existen diversos esquemas para esta función; algunos de ellos implementados en software, otros en hardware, y otros sin implementación publicada; por lo que se propone analizar y comparar extensivamente estas alternativas. Sobre esta base, se proponen cuatro arquitecturas genéricas específicamente orientadas a implementación en FPGAs. En particular, se aborda el análisis desde los criterios de *tipo de búsqueda* (valor exacto, prefijos, rangos arbitrarios) y de *metadato de agregación* (etiquetas o bitmaps). Se brindan resultados analíticos y de implementación que demuestran la importancia de utilizar el esquema de lookup más adecuado según los criterios citados, manteniendo siempre la velocidad de procesamiento requerida para redes de 100 Gbps. De este modo, se aportan criterios de diseño para la etapa de lookup en el contexto de arquitecturas de clasificación por descomposición. Como aporte adicional, se identifica entre estas alternativas la más adecuada para realizar clasificación basada en flujos, la cual actualmente presenta serios problemas en cuanto a consumo de memoria y actualización incremental. Para mitigar el primer problema se adopta una propuesta previa, mientras que para el segundo problema se proponen y demuestran nuevas optimizaciones.

Durante la presente Tesis se consideran dos aspectos generales, transversales al desarrollo de la misma. El primer aspecto es de carácter analítico. Si bien el problema de clasificación ha sido ampliamente explorado en la década pasada, es actualmente muy difícil establecer criterios de selección para una necesidad particular. Este problema es aún más notable dada la evolución de aplicaciones, desde los primeros esquemas de enrutamiento uni-dimensional basado en direcciones IP jerárquicas, pasando por las necesidades crecientes de calidad de servicio, hasta los modernos esquemas de clasificación multi-dimensional en redes virtualizadas. El último relevamiento completo del estado del arte en el área, donde se analizan ya 19 diseños muy heterogéneos, data del año 2005; sin embargo, en los últimos años esta área se ha visto notablemente ampliada mediante numerosas propuestas

especialmente orientadas al aspecto tecnológico. De esto, se observó la necesidad de actualizar el análisis del estado del arte, haciendo especial énfasis en los aspectos metodológicos y considerando luego la tecnología particular utilizada. Durante este análisis, realizado en cada una de las etapas de la Tesis, se abordan distintos aspectos del problema, siendo algunos de los más relevantes: la cantidad de campos considerados (uno o múltiples), el tipo de filtros considerados (prefijos, valores exactos, rangos arbitrarios), el tipo de resultado necesario (mejor o múltiples coincidencias), el formato de resultado conveniente o necesario (bitmaps o etiquetas), complejidad de actualización dinámica y pre-cómputo requerido.

En segundo lugar, se considera el aspecto de implementación en FPGAs, en particular para el caso de clasificación. Este aspecto involucra el diseño de sistemas digitales capaces de explotar el paralelismo ofrecido por los FPGAs, disponiendo convenientemente de los recursos de hardware de estos dispositivos. Para ello, se incluyen pruebas de concepto para cada una de las arquitecturas propuestas, demostrando su desempeño real mediante resultados de implementación. En términos generales, el consumo de recursos se mantiene por debajo del 30 % del total disponible en el dispositivo utilizado, tanto para el caso de lógica combinacional (LUTs) como de registros para todas las arquitecturas ensayadas. El consumo de memoria RAM, en tanto, se aproxima al 60 % sólo para los casos más grandes de TCAM emulada, manteniéndose por debajo del 20 % para las demás arquitecturas evaluadas. Esto es debido a que los requerimientos de velocidad impuestos son bastante exigentes, por lo que generalmente los límites de escalabilidad se alcanzan por disminución de velocidad (debido mayormente a retardos de ruteo) antes que por consumo de recursos.

Cabe mencionar que, si bien se utilizaron dispositivos del proveedor Altera por razones de disponibilidad, se tuvo especial consideración en utilizar recursos disponibles con mínimas variantes en FPGAs de otras empresas tales como Xilinx; de modo que las arquitecturas propuestas son portables a otros dispositivos.

Nuestras propuestas explotan mínimamente los patrones de reglas actuales, lo cual les aporta flexibilidad para adaptarse a distintas aplicaciones. Sin embargo, su escalabilidad puede aún mejorarse aprovechando más intensivamente las características del ruleset a utilizar. Para ello, y a modo de trabajo futuro, se analizan en la Sec. 7.1 técnicas de *reducción de complejidad* mediante *sectorización* y *estratificación* del ruleset. A partir de este análisis se logra comparar el trabajo existente sobre estas técnicas y detectar sus problemas para proponer mejoras. Se plantean asimismo las modificaciones necesarias a la arquitectura de agregación planteada para ser capaz de incorporar estas técnicas. Finalmente, se repasan en la Sec. 7.3 características de los rulesets actuales que pueden ser aprovechadas para estos objetivos, brindando resultados de análisis sobre rulesets reales que permiten apreciar tales características.

Capítulo 7

Trabajo Futuro

7.1. Generalización e integración de las propuestas

7.1.1. Disminución de complejidad

Si bien el esquema DCFV logra reducir el costo de BV manteniendo el uso de bitmaps y su bajo costo de actualización, aún se puede optimizar sin llegar a los extremos de RFC o DCFL. Esta oportunidad de optimización se identifica al observar que, si bien los bitmaps de DCFV tienen ancho $|UV| < N$, sólo $|UV|_{max} << |UV|$ bits darán resultados de match positivos para un determinado valor del key. Sin embargo, ya que DCFV codifica UVs según los pesos de cada bit, este ancho no se puede reducir directamente.

En el extremo opuesto, las técnicas basadas en etiquetas tales como DCFL o RFC tienen su principal inconveniente en que requieren agregación por *direccionamiento* para garantizar desempeño independiente del set de reglas. Si bien se pueden *comprimir* los espacios de direccionamiento mediante técnicas de hashing o filtros Bloom [97] [108], tales técnicas complican considerablemente el diseño y son fuertemente dependientes del patrón de reglas, por lo que no garantizan desempeño.

Situándose en un punto medio, se pueden lograr beneficios al utilizar técnicas que *reduzcan la complejidad* de clasificación, tanto en las etapas de lookup como de agregación. Estas técnicas no afectan el proceso de clasificación en sí mismo, sino que dividen el espacio de clasificación según algún criterio; de este modo las búsquedas locales o sus pre-procesamientos asociados pueden ser simplificados. En resumen, las técnicas a analizar no buscan optimización intensiva tal como lo hacen por ejemplo las técnicas basadas en árboles, sino una reducción general de la complejidad de clasificación.

Esta tendencia hacia métodos que agrupan reglas en *clusters de complejidad reducida* se observa

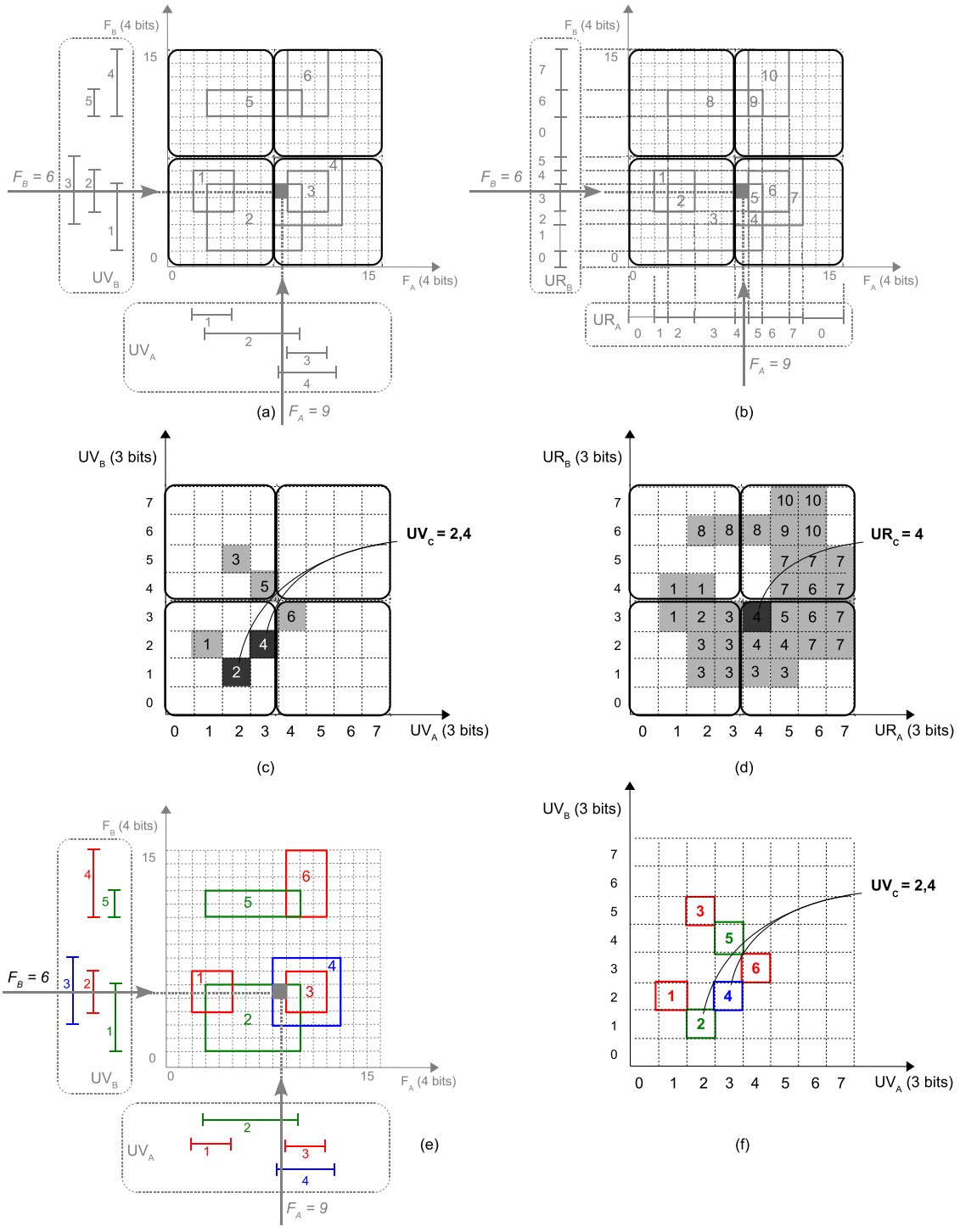


Figura 7.1: Estratificación vs. sectorización: (a),(b) sectorización 1D/2D para UVs y URs respectivamente, (c)(d) sectorización 2D para UVs y URs respectivamente, (e) estratificación de UVs, (f) mapa de capas de UVs

en numerosos trabajos recientes, los cuales esencialmente buscan de esta forma paralelizar algoritmos existentes de la forma más efectiva. En nuestro caso, se busca explotar estos métodos para mejorar

los aspectos negativos de las técnicas basadas en URs (complejidad de actualización, consumo de memoria) y en UVs (múltiples ciclos de procesamiento, dependencia de los solapamientos), potenciando sus aspectos positivos (velocidad independiente del ruleset para URs, y reducido consumo de memoria/complejidad de actualización para UVs), explotando para ello los recursos que ofrece un FPGA.

En primer lugar definiremos los métodos generales a aplicar, en primera instancia ellos se pueden dividir en:

- *Estratificación (layering)*: estas técnicas buscan reducir complejidad definiendo *capas* en el espacio de búsqueda. Estas capas se definen en base a algún criterio de optimización general, por lo que requieren un mínimo pre-procesamiento para asignar elementos a capas de forma mínimamente eficiente.
- *Sectorización (clustering)*: a diferencia del caso de layering, en este caso se divide el espacio de búsqueda en base a conjuntos de valores, ya sean éstos del key, de URIDs o de UVIDs; por ello deberían ser implementables mediante circuitos lógicos simples. En general, estas técnicas deberían ser más simples que las de layering, si bien los beneficios reportados pueden ser menores.

Esencialmente, las técnicas de estratificación consideran *todo el espacio de búsqueda* (ya sea en 1D o kD) para cada capa definida. En las técnicas de sectorización, en cambio, cada sector considera *sólo una porción del espacio de búsqueda* (1D o kD). Durante la operación de clasificación, las técnicas de estratificación *derivan en parelelo* el key de búsqueda hacia todas las capas existentes, mientras que las técnicas de sectorización *conmutan* parte del key hacia uno u otro sector según los bits de selección, basados en la parte restante del key. Estos conceptos permiten identificar la presencia de una u otra técnica.

En segundo lugar, se observa que el uso de metadatos basados en UVs, tales como los utilizados en DCFL o DCFV, requiere reportar múltiples resultados a causa de los solapamientos presentes. Por ello es que los bitmaps son naturalmente adecuados para representar tales UVs en forma totalmente concurrente, mientras que el uso de etiquetas requiere múltiples ciclos de clasificación (DCFL) o en su defecto replicación del esquema (DCFLE). Los metadatos basados en URs, en cambio, contienen todos los resultados en una sola etiqueta, por lo que se puede asegurar que sólo un URID resultará del proceso de clasificación.

En base a lo anterior, intuitivamente vemos que las técnicas de estratificación serían más adecuadas para metadatos basados en UVs, ya que éstos presentan naturalmente solapamientos que sería deseable reducir. Introduciendo estratificación de UVs, cada capa reporta un UV local y en conjunto todas las capas entregan el resultado de clasificación. Para metadatos basados en URs, en cambio, el uso de estratificación no reportaría beneficios significativos. Por un lado, se debe buscar concurrentemente en todas las capas, ya que no se sabe a priori en cuál de ellas reside el UR, pero por otro lado sólo una de ellas reportará el resultado. En este caso, parece más adecuada la técnica de sectorización, donde se puede dirigir la búsqueda exclusivamente al grupo que contiene el URID

resultante, sin activar los demás grupos de URIDs.

A fin de apreciar estos conceptos, se muestran en la Fig. 7.1 ejemplos de aplicación para el ruleset ya presentado en la Sec. 4.3. Las Figs. 7.1(a) y 7.1(b) muestran la división en sectores según valores del key; como se observa esta simple sectorización puede aún resultar en URs replicados en mútiples sectores. Las Fig. 7.1(c) muestra una segunda etapa de sectorización que permite separar efectivamente UVs; sin embargo como se observa los múltiples UVs coincidentes pueden en general encontrarse en un mismo sector. En la 7.1(d), en tanto, se muestra una segunda sectorización según URIDs que, colocada a continuación de la sectorización 1D por key, permite obtener exactamente una URID por sector. Ya que las búsquedas en cada sector pueden realizarse concurrentemente, es posible en el mejor caso efectuar dos operaciones de clasificación a la vez; sin embargo para ello los respectivos keys deben encontrarse en sectores diferentes. Por otro lado, como se aprecia en la Fig. 7.1(d), la sectorización permitiría disminuir tanto la profundidad como el ancho de memoria en cada etapa de agregación; sin embargo esto no necesariamente aporta beneficios en implementaciones FPGA debido a que las BRAMs disponibles tienen una mínima granularidad por debajo de la cual se reduce el rendimiento de memoria.

En las Figs. 7.1(e) y 7.1(f) se muestra el resultado de aplicar estratificación al mismo ruleset, identificando capas mediante distintos colores. En este caso se pueden obtener beneficios al reducir o eliminar los solapamientos en cada capa, disminuyendo con ello la complejidad de búsqueda en cada una de ellas. El valor de $|UV|$ en cada capa es reducido, y se puede lograr que $|UV| = |UR|$, de este modo se pueden utilizar UVIDs de ancho reducido que facilitan asimismo la agregación.

7.1.2. Técnicas de sectorización

A fin de comprender mejor el principio de estas técnicas y apreciar su conveniencia, se consideran a continuación esquemas generales de implementación. A partir de lo discutido, se analizarán asimismo los metadatos correspondientes propagados en cada esquema. En la Fig. 7.2(a) se muestran las etapas de agregación sectorizada para el caso de la Fig. 7.1(b). Entre corchetes se indica el sector 1D o 2D asociado a cada resultado. En cada nodo de agregación, se muestran los URIDs locales definidos, mientras que entre paréntesis se muestran los URIDs originales asociados (sin sectorización). En este caso, se requieren 4 bloques independientes de memoria a fin de mapear keys a URIDs 1D. asimismo, son necesarios 4 bloques adicionales para mapear los resultados 1D a URIDs 2D, implementando efectivamente los cuatro sectores 2D de la Fig. 7.1(d). Los URIDs resultantes de cada bloque serán locales a sus respectivos sectores, por lo que para obtener un URID global se afecta por su ID de sector. Finalmente, ya que sólo uno de estos bloques arroja el URID válido, el resultado de clasificación se puede obtener mediante una operación OR. En este esquema, es teóricamente posible realizar dos operaciones de clasificación a la vez, sin embargo esto requiere que los respectivos sectores no compartan acceso al mismo bloque de memoria; por ejemplo se podrían obtener resultados simultáneos en los sectores 2D $[0;0]$ y $[1;1]$ o en los sectores 2D $[0;1]$ y $[1;0]$, pero no en los sectores $[0;0]$ y $[0;1]$ o $[1;0]$ y $[1;1]$ ya que éstos comparten bloques 1D $[0]$ y $[1]$ respectivamente. El consumo de memoria y la complejidad de actualización pueden reducirse con respecto a la de RFC dependiendo de las características del ruleset; en general los mayores beneficios se pueden obtener cuando los

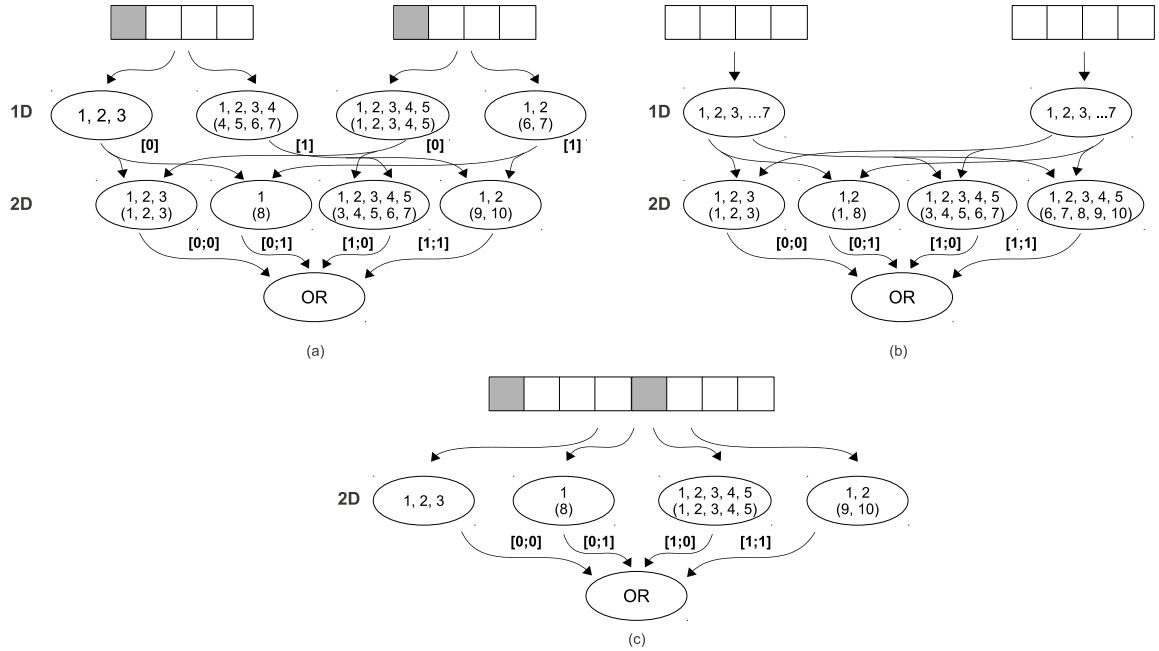


Figura 7.2: Sectorización: (a) aplicada al key, (b) aplicada a URIDs 1D, (c) directa

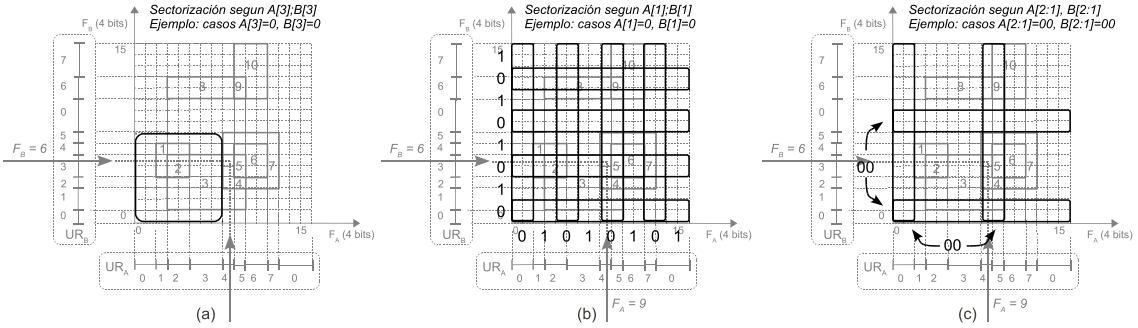


Figura 7.3: Sectorización directa 2D: (a) según $A[3], B[3]$, (b) según $A[1], B[1]$, (c) según $A[2:1], B[2:1]$

sectores se distribuyen equitativamente los URIDs sin compartir muchos URIDs.

En la Fig. 7.2(b), en tanto, se observa la implementación del mapa de la Fig. 7.1(d). Este caso se diferencia del anterior en que la sectorización se realiza sobre los URIDs 1D, es decir que el lookup 1D no se sectoriza. De este modo, para poder realizar múltiples clasificaciones a la vez las etapas de lookup deberían trabajar a múltiplos de la velocidad de línea, en este caso al doble; tranformándose en un cuello de botella. Los beneficios en cuanto a consumo de memoria y complejidad de actualización son más controlables que en la Fig. 7.2(a) ya que los URIDs son asignados de la forma más conveniente, a diferencia de los valores de key que deben ser consecutivos.

Los esquemas de las Fig. 7.2(a) y 7.2(b) implementan *sectorización aplicada* a un esquema de agregación como lo es RFC. El esquema de la Fig. 7.2(c), en cambio, aplica *sectorización directa* a partir de valores del key, y *luego* realiza clasificación 2D en cada cluster resultante. En otras

palabras, en este caso sólo se busca agrupar reglas (y sus respectivos URIDs 2D), para luego realizar clasificación 2D en cada sector. El beneficio en este caso reside en que los grupos definidos presentarán en general menor complejidad de clasificación y consumo de recursos a costa de una simple etapa intermedia basada en multiplexores. Se pueden obtener múltiples resultados de clasificación siempre que los respectivos keys accedan a bloques distintos de memoria según los bits seleccionados para realizar la sectorización.

Concentrándonos ahora en la técnica de la Fig. 7.2(c), vemos en la Fig. 7.3 tres posibles mapas de sectorización para un key formado por dos campos A y B de cuatro bits cada uno. Las variables a considerar son i) la cantidad de bits del key utilizados para sectorizar, y ii) la posición de dichos bits dentro del key. La Fig. 7.3(a) muestra la sectorización correspondiente a la Fig. 7.2(c), donde se sectoriza de acuerdo a los MSbits $A[3]$ y $B[3]$ de cada campo respectivamente. En la Fig. 7.3(b), en tanto, se explora el mapa resultante de seleccionar los bits $A[1]$ y $B[1]$; mientras que la Fig. 7.3(c) muestra el mapa resultante de sectorizar según cuatro bits $A[2:1], B[2:1]$. En este último caso se definen 16 sectores en lugar de los 4 anteriores. Eligiendo cuidadosamente los bits a considerar, se pueden obtener beneficios en cuanto a velocidad, consumo de memoria y complejidad de actualización.

Los trabajos [149] y [150] exploran particionado basado en selección de ciertos bits del key, llamados bits *eficientes (E-bits)*, a partir del análisis de rulesets tradicionales basados en quíntuplas IP [134]. En estos trabajos se aplican dos mecanismos heurísticos, *Fast Growth e Intelli-Evolution*, a fin de lograr optimización en el consumo de memoria y latencia de clasificación. Para determinar las *posiciones* de los bits a utilizar y su *cantidad* se utilizan dos funciones, *Judge (J) y Performance (P)* respectivamente. En cada uno de los sectores así definidos se realiza clasificación mediante algún esquema multi-dimensional como búsqueda lineal o árboles. Cuanto más bits se utilicen, la complejidad de cada grupo disminuye pero se tiene mayor consumo de recursos. De acuerdo a los resultados reportados, este esquema puede disminuir sensiblemente el consumo de memoria y latencia respecto a otros mecanismos como RFC, HiCuts o HyperSplit; sin embargo el procesamiento necesario para determinar una estructura óptima debe justificar los beneficios obtenidos. Además, la optimización obtenida puede degradarse al realizar actualizaciones frecuentes. En particular, los autores hallan que los E-bits más adecuados se encuentran entre los bits 0-3 (comienzo del campo SrcIP), 26-30 (fin del campo SrcIP), 32-39 (comienzo del campo DstIP), y 101 (protocolo) en rulesets tipo Lista de Control de Acceso *(Access Control List, ACL)* y tipo Cadena IP *(IP Chain, IPC)*. Para los rulesets tipo Firewall (FW), en tanto, el campo DstPT también aporta E-bits. Este efecto se atribuye a que los rulesets ACL e IPC típicamente se concentran en restringir o permitir el acceso de ciertos usuarios a un recurso computacional, mientras que los rulesets FW se concentran asimismo en proveer control avanzado de aplicaciones con mayor granularidad. Ya que los E-bits se concentran mayormente al inicio o al final de los campos, se propone una sectorización en dos niveles, dividiendo primero el espacio de búsqueda e grandes sub-espacios continuos tales como los de la Fig. 7.3(a), y luego separando en pequeños sub-espacios locales como los de la Fig. 7.3(b).

Otro trabajo más reciente [151] analiza extensivamente rulesets reales basados en quíntuplas IP, concluyendo que se puede obtener una sectorización eficiente basándose en los bytes más significativos de DstIP y SrcIP respectivamente. De este modo, este trabajo explora una simple *sectorización en*

dos dimensiones de forma directa, brindando implementaciones eficientes tanto en software como en FPGAs. En términos generales, se utiliza una primera tabla de sectorización conteniendo $2^{(8+8)} = 64K$ índices; estos índices direccionan estructuras que contienen las reglas coincidentes para cada sector así definido. Asimismo, las reglas que involucran wildcards (" * ") en alguno de los dos campos se almacenan en tablas separadas, que de otro modo deberían almacenarse para cada valor del campo sobre el que se define el wildcard. El valor de este esquema radica en su simplicidad y efectividad para los rulesets analizados, brindando pruebas concretas de su factibilidad de implementación real. Los principales problemas de esta arquitectura se relacionan con su adaptación a rulesets que se alejen de estos patrones en sus filtros IP, ya que las estructuras de datos utilizadas pueden variar sustancialmente su costo.

7.1.3. Técnicas de estratificación

7.1.3.1. Aspectos generales

Las técnicas de estratificación afectan esencialmente el formato y significado del metadato propagado en el esquema de agregación, a fin de optimizar la complejidad de dicho esquema. En general, podemos encarar un estudio de estas técnicas dividiendolas en dos grandes grupos:

1. Técnicas que buscan obtener capas *sin solapamiento interno*, es decir que dentro de cada capa los intervalos definidos son disjuntos. En este caso se obtiene complejidad de clasificación óptima en cada capa, ya que se llega a un caso local basado en UVIDs.
2. Técnicas que buscan obtener capas con *solapamiento máximo controlado*. Estos métodos disminuyen la complejidad de esquemas como RFC, pero la clasificación en cada capa aún se basa en URIDs locales a esa capa.

A su vez, como veremos, ambas técnicas se pueden aplicar considerando una dimensión o múltiples dimensiones a la vez.

A fin de introducir nuestro análisis, se muestra en la Fig. 7.4 una serie de posibles metadatos de agregación, basada parcialmente en [121]. Estos metadatos son referidos en [121] como *estilos de codificación* para *clasificación paralela de paquetes (Parallel Packet Classification, PPC)*. A fin de mostrar sus funcionamientos, se considera como ejemplo el ruleset y valores de key ya analizados en la Fig. 4.4, separando sus campos ortogonales A y B. Se comienza por el extremo que minimiza pre-cómputo a costa de ineficiencia en ancho (BV), ilustrado en a Fig. 7.4(a), finalizando en el extremo que minimiza el ancho de palabra a costa de intensivo pre-cómputo (URID), mostrado en la Fig. 7.4(g). El caso de la Fig. 7.4(b) utiliza bitmaps disjuntos de valores de campo (DBFVs), y es el utilizado en nuestra propuesta. En las Figs. 7.4(c) a 7.4(e), se agrupan UVs en *capas sin solapamiento*, buscando de esta forma reducir el ancho de palabra sin recurrir al uso de URIDs. Cabe destacar que, ya que no más de $|UV|_{max}$ UVs coinciden a la vez para un determinado valor de key, podemos deducir que existirán no más de $|UV|_{max}$ capas sin solapamiento; mientras que cada

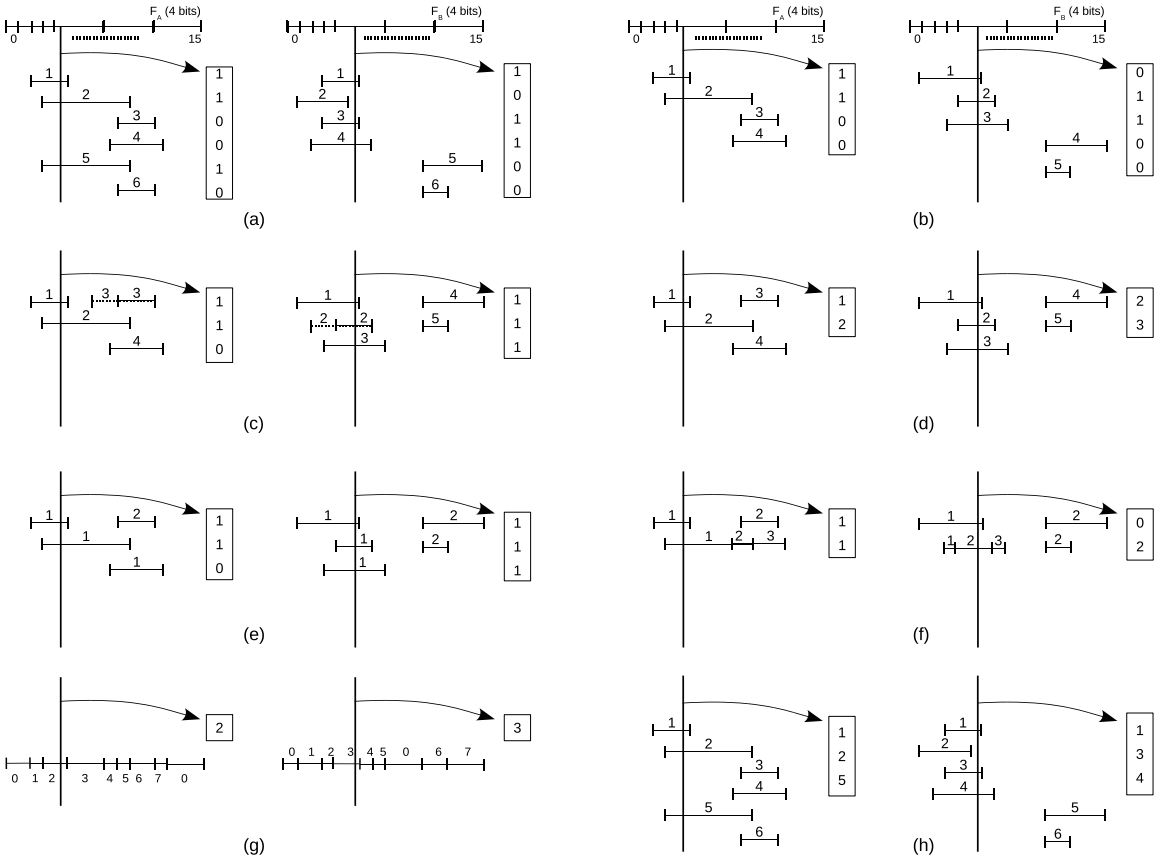


Figura 7.4: Metadatos de agregación: (a) BV, (b) DBFV, (c) PPC-2, (d) UVIDs, (e) LUVIDs (PPC-1), (f) PPC-3, (g) URID (PPC-4), (h) RIDs ([120])

capa contendrá sólo una porción de la complejidad total de clasificación; esto sugiere la conveniencia de explorar estas alternativas.

La Fig. 7.4(c) muestra el caso referido como estilo 2 en [121] (PPC-2); este caso utiliza una etiqueta donde cada bit representa simplemente el *estado de coincidencia (1 o 0)* de cada capa. La *combinación* particular de estos estados será el resultado de lookup en cada campo. Este método, propuesto en [121], tiene el problema de que no permite más de $2^{|UV|max}$ combinaciones diferentes de entre las $|UR|$ combinaciones posibles. En este ejemplo particular existen $7 < 2^3$ URs por lo que este esquema es válido, sin embargo si modificáramos los intervalos de los UV 3 (A) y 2 (B) como se muestra en trazos, no sería posible diferenciar por ejemplo las combinaciones de UVs 1-2 y 3-2 en A (ambas darían como resultado la etiqueta 110) o las combinaciones 1-2 y 4-5 en B (ambas darían la etiqueta 110).

En la Fig. 7.4(d), en tanto, se muestra un caso donde se utilizan UVIDs en capas sin solapamiento interno. Esto es esencialmente un caso totalmente paralelizado de lo que utiliza DCFL, tal como lo implementa DCFLE [83]. Sin embargo, se debe notar que en el caso de DCFLE sólo se *replica* la estructura de DCFL, que contiene *todos* los UVs posibles en el espacio de lookup o agregación, a fin de obtenerlos en un mismo momento. Podemos mejorar este esquema como se muestra en la

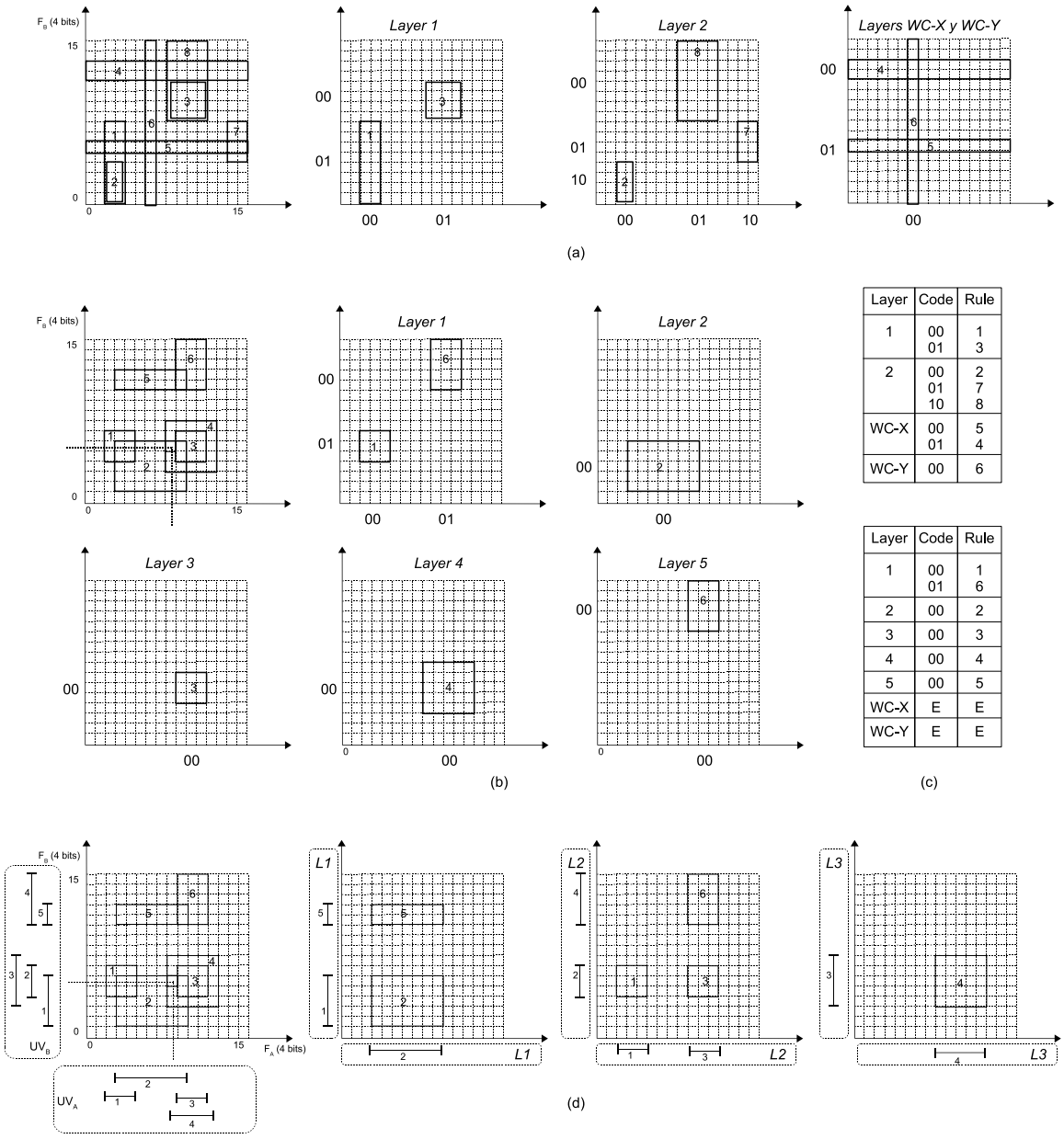
Fig. 7.4(e), donde los UVIDs son completamente *locales* a su capa de pertenencia (*Local* UVIDs o LUVIDs), referido como *estilo 1* en [121] (PPC-1). De este modo, el significado de cada etiqueta estará dado por i) su valor y ii) su capa de pertenencia; pudiendo reducirse el ancho de etiquetas según los LUVIDs presentes en cada capa.

En la Fig. 7.4(f) se muestra un esquema donde se permite un *solapamiento máximo* controlado en cada capa; de esta forma se reduce el número de capas a costo moderado de pre-cómputo. Este es efectivamente un caso intermedio entre UVs y URs, referido como *estilo 3* (PPC-3) en [121]. En el extremo opuesto a BV, en la Fig. 7.4(g) se muestra el caso de URIDs tal como lo utiliza RFC. En este esquema se define una capa única que considera una nueva etiqueta URID por cada combinación posible de UVs; estas etiquetas son pre-computadas durante la carga de reglas con lo que se logra muy alta velocidad de clasificación a costa de mayor complejidad de actualización. Finalmente, la Fig. 7.4(h) muestra el metadato utilizado en [120], basado en múltiples etiquetas que representan números de reglas kD (RIDs); esto es equivalente a utilizar UVIDs kD en todas las etapas de lookup, por lo que en este caso no es necesario realizar agregación sino comparación cruzada de todas las etiquetas resultantes. Si bien este último método reduce notablemente la compljeidad de agregación, requiere que se mantengan N etiquetas en cada etapa de lookup; esto puede resultar muy ineficiente dado el intensivo re-uso de UVs en el rulesets kD.

Al momento existen diversos trabajos que adoptan técnicas de estratificación, si bien no todos ellos dejan en claro este hecho. Por ello los repasaremos brevemente, destacando los aspectos relevantes para nuestro trabajo. Comenzaremos considerando el caso de capas sin solapamiento interno. A su vez, esta estratificación se puede definir i) directamente en kD, ii) en $1D$ para cada campo considerado, o iii) en $d < k$ dimensiones. Estos casos son fundamentalmente diferentes ya que la combinación de layers independientes no conducirá necesariamente a resultados de agregación independientes; esto es debido a que la cantidad de capas necesarias y la complejidad para determinarlas es en general mayor a medida que se agregan dimensiones. Finalmente, se puede considerar una combinación de ambas técnicas, aplicando por ejemplo la primera para agregación 2D donde la complejidad es moderada y adoptando a partir de allí la segunda.

7.1.3.2. Capas independientes sin solapamiento interno

Las técnicas [152] y [153] implementan estratificación sin solapamiento en kD, si bien sólo consideran el caso de BM sobre reglas bi-dimensionales basadas exclusivamente en prefijos, y rulesets típicos como los reportados en [103] y [109]. Este caso no sólo es muy limitado con respecto a un caso general como por ejemplo OpenFlow, sino que los solapamientos para el caso de prefijos son mucho más simples de resolver que los producidos por AR, debido a que los prefijos no pueden presentar solapamientos parciales (se profundiza este aspecto en el Cap. 5). En particular, [153] pondera complejidades de ruleset mediante la probabilidad β de que un prefijo sea a su vez prefijo de otro. Se considera el factor promedio $\beta = 10^{-4}$ que conduce a un máximo de 14 layers en 2D, contrastando resultados con las técnicas BV y ABV que son las únicas que permiten hacer agregación por simple intersección de resultados 1D (por lo que en rigor no utilizan agregación sino concatenación de resultados).



Layer	Code	Rule
1	00 01	1 3
2	00 01 10	2 7 8
WC-X	00 01	5 4
WC-Y	00	6

Layer	Code	Rule
1	00 01	1 6
2	00	2
3	00	3
4	00	4
5	00	5
WC-X	E	E
WC-Y	E	E

Figura 7.5: Técnicas de estratificación: (a) LCV aplicado a prefijos 2D, (b) LCV aplicado a rangos arbitrarios 2D, (c) capas resultantes para (a) y (b) respectivamente, (d) LFL aplicado a rangos arbitrarios 1D

En general, el problema de descomponer un espacio de búsqueda multi-dimensional en sub-espacios máximos multi-dimensionales sin colisión interna puede referirse como un problema del tipo de *Coloración de grafos* [104], el cual es del tipo NP-hard; esta afirmación se relaciona con la redundancia de valores de campos y consecuente necesidad de pseudo-reglas como puede observarse en la Fig. 4.9. [153] ataca este problema mediante un algoritmo del tipo *greedy*, el cual sigue la heurística de buscar optimizaciones locales y confiar en que estas optimizaciones llevarán a una solución global satisfactoria si bien no óptima. Llevado a nuestro caso, esto significa determinar capas independientes sin pretender que el número de capas resultante sea un mínimo global; esto

puede resultar en número de capas mayor al óptimo a cambio de reducir el tiempo de procesamiento y permitir actualizaciones más simples. La dificultad principal de la técnica de estratificación kD reside en que, para cada regla en el ruleset, se deben comprobar posibles solapamientos *en todas las dimensiones* para las capas definidas.

En la Fig. 7.5(a) se considera esta técnica aplicada a un ruleset simple 2D basado en prefijos, tal como lo hace [153]. Por ejemplo, la regla 1 involucra los prefijos $F_A = 001X, F_B = 0XXX$, la regla 4 se define como $F_A = XXXX, F_B = 111X$ (WC-X) y la regla 6 es $F_A = 0110, F_B = XXXX$ (WC-Y). Como se observa, LCV define capas *comunes a ambas dimensiones* (Capas 1 y 2 en este caso), así como dos capas específicas para reglas que involucran wildcards en los ejes X (reglas 4 y 5) e Y (regla 6) llamadas $WC-Y$ y $WC-X$ respectivamente. Ahora bien; si llevamos este esquema al caso AR considerado anteriormente en este trabajo, como se muestra en la Fig. 7.5, surgen solapamientos parciales lo que puede aumentar sensiblemente la complejidad. Para este ejemplo, son necesarias 5 capas, 4 de las cuales (2, 3, 4 y 5) contienen sólo una regla. Esto se resume en las tablas de la Fig. 7.5(c), donde se observan los IDs de capa, índices dentro de cada capa y reglas relacionadas para los casos de las Figs. 7.5(a) y 7.5(b) respectivamente. De este ejemplo, se puede observar la reducida eficiencia de LCV para procesar rulesets donde se involucran muchos ARs.

Abordamos ahora propuestas de estratificación 1D. El trabajo aquí referido como *Sets Independientes (ISET)* [154] define capas sin solapamiento interno para cada campo y luego utiliza para estra-tificación sólo el campo que generó capas óptimas, es decir, aquél campo para el cual la relación (tamaño de capa) / (número de capas) es máxima. Cabe destacar que, para optimizar la generación de tales capas, los intervalos definidos por las proyecciones de reglas en cada campo (es decir los intervalos de UVs) se ordenan previamente según sus *puntos finales*. Las estructuras de búsqueda se generan posteriormente para este campo según los *extremos iniciales* de cada intervalo; de esta forma se reduce notablemente el almacenamiento con respecto a las estructuras utilizadas para definir UVIDs o URIDs. El costo por esta reducción es la pérdida de sensibilidad respecto a los puntos finales, lo que se resuelve en una segunda etapa por comparación directa de campos. Para cada capa se almacena el índice de la regla kD asociada; de este modo el resultado de lookup será un conjunto de índices utilizados para realizar una búsqueda lineal *reducida* en la tabla de reglas original. El mecanismo de lookup 1D adoptado se basa en árboles de búsqueda de rangos. En las Figs. 7.6(a) y 7.6(b) se observa la aplicación de este método a nuestro ruleset de ejemplo; en el caso de la Fig. 7.6(a) se analiza el caso donde se utiliza el campo A mientras que la Fig. 7.6(b) ilustra el caso donde se utiliza el campo B; si bien como dijimos sólo se utilizará para la clasificación el que resulte óptimo. Como se observa, en este caso el campo B es el más conveniente desde el punto de vista del consumo de memoria ya que genera menos capas. Se ilustra asimismo un caso de clasificación $F_A = 9, F_B = 6$. Vemos que, en el caso de utilizar el campo B, el resultado son los índices $2, 1, 3, 4$, los cuales se utilizan para búsqueda lineal en las respectivas entradas de la tabla de reglas. Esto puede hacerse en forma secuencial o concurente (por ejemplo mediante una TCAM). El trabajo no aporta mayores detalles de implementación, limitándose al análisis de su consumo optimizado de memoria.

Se deben notar algunos puntos importantes de este esquema. Por un lado, se observa que no se utiliza el concepto de agregación, sino que se utiliza una búsqueda 1D en el campo de capas óptimas

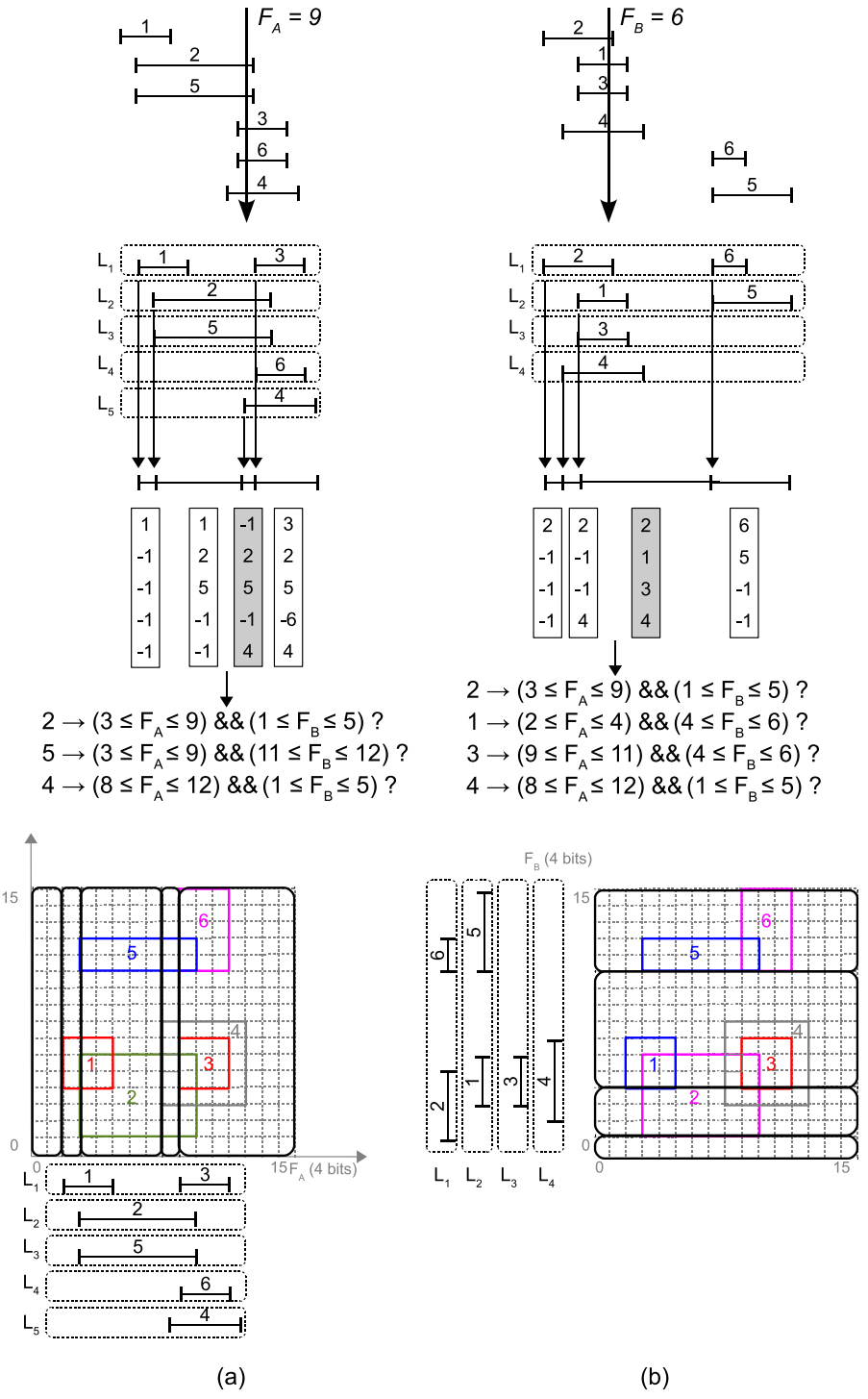


Figura 7.6: Técnica de Sets Independientes: (a) utilizando el campo A, (b) utilizando el campo B

para *filtrar* la cantidad de accesos de búsqueda lineal en la tabla de reglas; vale decir entonces que la búsqueda según capas 1D se utiliza para posterior sectorización directa de las demás dimensiones. Se puede interpretar el proceso como dos etapas; la primera de ellas consiste en sectorización y consecuente reducción del espacio de búsqueda en base a *puntos de inicio* de UVs (no a selección de

bits como vimos anteriormente). Luego la búsqueda en cada sector se realiza en forma lineal sobre el sector de reglas correspondiente. En la Fig. 7.6(a) puede apreciarse este mecanismo para el caso de búsqueda en el campo A, mientras que la Fig. 7.6(b) considera búsqueda 1D sobre el campo B. El reducido consumo de memoria de este esquema depende fuertemente del procesamiento del ruleset para buscar el óptimo agrupamiento de reglas en un campo; sin embargo como se destaca en el trabajo esta optimización puede perderse luego de múltiples actualizaciones incrementales por lo que se debería re-generar la estructura cada cierto tiempo con el costo computacional que esto implica. Por otro lado, se observa que los intervalos definidos en cada campo *no son UVs 1D sino UVs kD*; es decir, dos intervalos 1D iguales (definiendo un solo UV 1D) generan igualmente dos capas distintas; esto se puede observar por ejemplo en la Fig. 7.6(a) para las reglas 2 y 5 o 3 y 6 del campo A. Esto puede limitar seriamente la escalabilidad de este esquema con respecto a las técnicas basadas en UVIDs/URIDs; de hecho en el trabajo original se define para cada campo un *factor de repetición* $f = N/|UV|$ para ponderar este aspecto. Finalmente, podemos relacionar esta técnica con las de agregación argumentando que los múltiples índices que devuelve el lookup 1D actúan esencialmente como los múltiples UVIDs que devolvería una técnica basada en UVs. Po rejemplo, en un esquema como DCFV el campo A devolvería un solo UV para las reglas 2 y 5, pero éste debería agregarse con dos UVs del campo B para obtener el resultado de casificación. En ISET, en cambio, al campo A devuelve 2 UVIDs directamente en kD (es decir, IDs de regla) para acceder a la tabla de reglas en la segunda etapa. Esta diferencia puede apartar notablemente este esquema de los demás cuando el factor de repetición aumenta.

Consideremos ahora las técnicas de estratificación 1D que forman capas completamente locales a su campo, es decir que se basan exlusivamente en UVs 1D para definir estas capas. Estas propuestas reducen la complejidad de procesamiento respecto a las técnicas kD, sin embargo requieren de una etapa de agregación más que simple concatenación de resultados. En la Fig. 7.5(d) se muestra un caso de estratificación 1D en UVs independientes según el esquema de la Fig. 7.4(e). Este caso puede ser fácilmente comparado con el de la Fig. 7.5(b), observando que en general son necesarias menos capas ya que los solapamientos considerados son en cada dimensión particular. Asimismo, se nota la diferencia respecto a la técnica ISET ya que se generan sólo 3 capas en lugar de 5, si bien se requiere un lookup independiente por campo. El primer trabajo en analizar estas técnicas es *Clasificación paralela de paquetes (Parallel Packet Classification, PPC* [121], el cual considera tanto casos de capas sin solapamiento interno como con solapamiento interno controlado adoptando agregación mediante TCAMs. Los intervalos definidos en cada capa se denominan en ese trabajo *rangos primitivos*. Mediante estas técnicas, el objetivo específico de PPC es generar una etapa de *compresión del key* capaz de reducir el ancho de la memoria TCAM utilizada. Trabajos posteriores, como [155], exploran diferentes variantes de estratificación aplicadas en forma local a cada campo, diferenciándose en el grado de independencia buscado y la técnica de agregación utilizada. Estas opciones permiten explorar diferentes compromisos entre complejidad de cómputo y consumo de recursos. El Algoritmo 4 ilustra un procedimiento general de estratificación en 1D aplicado a un ruleset kD.

Finalmente, [156] evalúa estratificación de rulesets en base a un número de dimensiones $d < k$, definiendo así capas cuyos UVs en d dimensiones no presentan solapamiento (si bien sí lo pueden presentar en las demás $k - d$ dimensiones). [156] utiliza esta técnica con el objetivo específico de

Algoritmo 4 Algoritmo de separación en capas independientes 1D

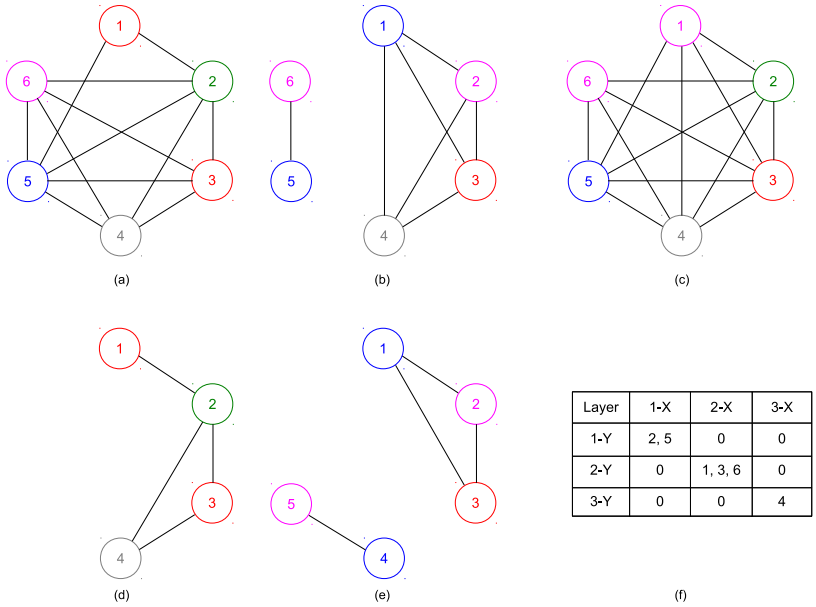
Require: Ruleset C containing N rules

$j \leftarrow 0$
while C is not empty **do**
  $C1 \leftarrow C$
  $d \leftarrow field\ with\ maximal\ |UR|$
  $rp[0 : |UV| - 1] \leftarrow sort\_right\_points(C, d)$
  $prev\_rp \leftarrow 0$
  // find maximum independent set on this field
  for $i = 1$ to $|UV|$ **do**
    $curr\_rp \leftarrow rp[i]$
    $r \leftarrow least\ used\ UV\ among\ [prev\_rp, curr\_rp]$
    if r == NULL **then**
      break
    end if
    Insert r into $I[j]$
    Remove r from C
    Remove r from $C1$
    Remove UVs that overlap with r from $C1$
    $prev\_rp \leftarrow curr\_rp$
  end for
  $j \leftarrow j + 1$
end while

disminuir el problema de duplicación de reglas en árboles multi-dimensionales, ocasionado por los solapamientos entre las mismas. En particular, en ese trabajo se evalúan estratificaciones basadas en campos individuales de rulesets basados en quíntuplas (SrcIP, DstIP, SrcPrt, DstPrt, Prot), así como estratificación basada en duplas $< SrcIP, DstIP >$. En cada sub-grupo así definido se aplica clasificación basada en árboles del tipo *quarter-cut*, los cuales resultan simples debido a la eliminación de solapamientos internos. En el caso de prefijos como SrcIP y DstIP, el particionado se realiza en base a *niveles de prefijo*, mientras que para los campos basados en rangos como SrcPrt y DstPrt la estratificación se realiza en forma similar a [121]. Sobre esta base, se observa que la estratificación 1D puede resultar insuficiente para reducir el número de reglas en las hojas del árbol multi-dimensional. Para mejorar esto, se considera la estratificación en base a duplas $< SrcIP, DstIP >$. Según los análisis de [90] y [95], la cantidad de reglas coincidentes para el 99.99 % de los productos cruzados de duplas $< SrcIP, DstIP >$ no es más de 5, mientras que en el peor caso no se superan las 20 reglas coincidentes al considerar el espacio bi-dimensional definido por estas duplas. Esto significa que, para casos normales, existirán no más de 5 capas en el espacio $< SrcIP, DstIP >$, con complejidad de clasificación reducida en cada una de ellas. En base a estas observaciones, [156] particiona el ruleset según este espacio bi-dimensional y luego aplica árboles multi-dimensionales a cada uno de los sub-grupos de reglas resultantes.

7.1.3.3. Análisis mediante grafos

A fin de comparar objetivamente estas técnicas, podemos expresar las relaciones involucradas mediante *grafos*. Para ello, recordamos que un grafo G consiste en dos grupos (V, E) donde V son los



Layer	1-X	2-X	3-X
1-Y	2, 5	0	0
2-Y	0	1, 3, 6	0
3-Y	0	0	4

Figura 7.7: Representación mediante grafos: (a)(b) sets correspondientes los sectores de las Figs. 7.6(a) y 7.6(b) respectivamente, (c) sets independientes en 2D, (d)(e) sets independientes basados en UVs 1D, (f) productos cruzados entre sets independientes 1D

Regla	F_A	F_B
R1	45000:50000	10000:15000
R2	23000:65535	0:45000
R3	40000:50000	50000:55000
R4	0:30000	40000:65535
R5	20000:25000	51000:52000
R6	30000:35000	20000:25000
R7	10000:15000	5000:10000

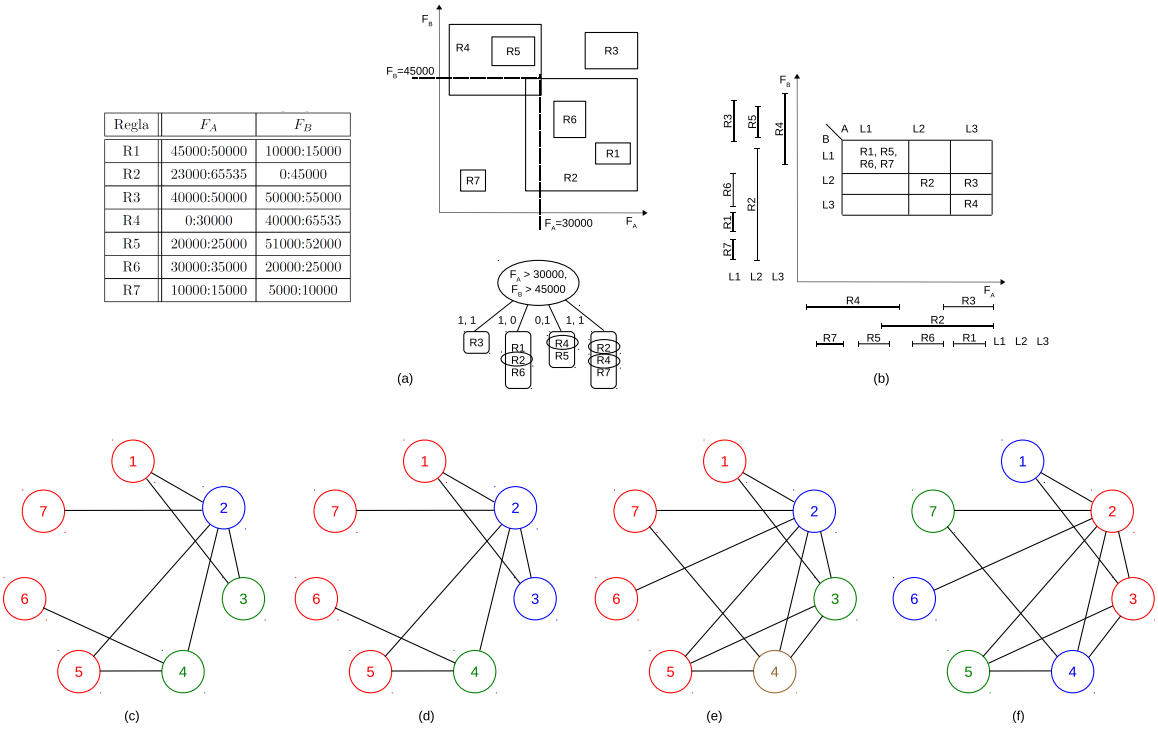


Figura 7.8: Sets independientes en 1D y 2D: (a) ruleset de ejemplo, (b) productos cruzados de sets independientes 1D, (c)(d) grafos correspondientes a las dimensiones X e Y respectivamente, (e) grafo 2D resultante de los productos cruzados de UVs estratificados 1D, (f) grafo 2D resultante de la estratificación 2D

nodos (vértices) del grafo y E son las aristas (edges) interconectando dichos nodos por pares; de este modo una arista $e \in E$ es un sub-grupo conteniendo dos elementos de $V : e = u, v$ donde $u, v \in V$ [157]. Para nuestro caso, los nodos representan UVs en las dimensiones consideradas, mientras que las aristas representan solapamientos entre estos UVs. El objetivo de los algoritmos de particionado es, entonces, el de obtener el mínimo número de grupos independientes de reglas (grupos no-solapados de reglas) conteniendo cada uno el máximo número de nodos (reglas) posible. De este modo, se reduce la complejidad de clasificación en cada grupo, obteniendo a la vez la combinación de grupos más conveniente; lo que resulta en definitiva en el compromiso más conveniente de consumo de memoria. Este problema, común a varias aplicaciones en computación, es llamado de los *Grupos Independientes Máximos*, y su versión 1D se trata como problema de *planificación de intervalos* en [157]. En general, dado un grafo $G = (V, E)$ donde nodos $v \in V$ representan UVs y las aristas $e \in E$ representan conflictos o solapamientos entre ellas, decimos que un grupo de nodos $S_1 \subseteq V$ es *independiente* de otro grupo de nodos $S_2 \subseteq V$ si ningún par de nodos $V_1 \in S_1,\ V_2 \in S_2$ está unido por una arista; el problema consiste entonces en hallar los grupos independientes tales que su cantidad sea mínima y su tamaño sea máximo.

En las Figs. 7.7(a) y 7.7(b) se muestran los grafos para los casos de las Figs. 7.6(a) y 7.6(b) respectivamente, los cuales utilizan sectorización en forma *local a sus respectivas dimensiones*. Cada nodo, representa en este caso las múltiples reglas kD que comparten su UV asociado tal como lo realiza [154]. Asimismo, como se observa claramente, estos grupos obtenidos por sectorización no son en general independientes entre sí sino el resultado de una división basada en otras métricas generales. De este modo, los resultados de lookup 1D obtenidos arrojan directamente un conjunto de reglas kD no requiriendo agregación con los demás campos; este es un esquema similar al de *resolución de colisiones por encadenamiento* en tablas hash, requiriendo una búsqueda lineal auxiliar dentro de cada nodo (o el uso de URIDs pre-calculados para cada nodo). Por otro lado, cada set independiente en las dimensiones X e Y se indican mediante un color distintivo en las Figs. 7.7(a) y 7.7(b) respectivamente. En este ejemplo, las capas independientes (sin solapamientos internos) en la dimensión X podrían ser $\{1, 3\}; \{2\}; \{4\}$; $\{5\}$; y $\{6\}$; mientras que en la dimensión Y podrían ser $\{1, 5\}; \{2, 6\}; \{3\}$; y $\{4\}$. Por otro lado, en la Fig. 7.7(c) se ilustra la separación en grupos independientes según kD, es decir considerando en el grafo las dos dimensiones a la vez tal como lo propone [153]; vemos que en este caso se generan 5 capas (colores) sin solapamientos internos en $2D$ de acuerdo a lo ya analizado geométricamente en la Fig. 7.5(b). Finalmente, las Figs. 7.7(d) y 7.7(e) ilustran agrupación en 1D pero considerando los UVs definidos en cada dimensión; de esta forma se puede reducir sensiblemente el número de capas (colores) a 3, como ya se analizó geométricamente en la Fig. 7.5(d). En este último caso, a diferencia de los anteriores, los nodos involucrados y sus solapamientos son completamente locales a sus respectivas dimensiones; en consecuencia es necesaria una etapa de agregación de campos. Dadas dos dimensiones A y B conteniendo I_A y I_B grupos independientes respectivamente, existen en general $I_A \times I_B$ tablas de agregación, de las cuales $I_{AB} \leq I_A \times I_B$ serán no-nulas. Es decir, si bien los grupos I_A e I_B son independientes para sus respectivos campos, existen en general productos cruzados *entre intervalos de distintas capas 1D* que generan nuevas capas en el espacio 2D. Es en este punto donde advertimos la diferencia fundamental entre definir grupos en d dimensiones $(1 < d \leq k)$ o en 1 dimensión.

En el ejemplo particular de la Fig. 7.7(f), los productos cruzados de las tres capas en X y en Y

respectivamente dan como resultado exactamente tres capas 2D resultantes de la combinación de sus respectivas capas 1D, por lo que no se aprecia el efecto antes mencionado. En la Fig. 7.8(a), en tanto, se ilustra otro ruleset mencionado en [156] basado en duplas $< SrcPrt, DstPrt >$. La Fig. 7.8(b) muestra el resultado de los productos cruzados entre UVs 1D en X e Y respectivamente, donde se observa un producto cruzado adicional no-nulo entre las capas X-3 e Y-2, resultando en cuatro capas independientes en 2D. En las Figs. 7.8(c) y 7.8(d) se observan los grafos correspondientes a los campos X e Y respectivamente, mientras que la Fig. 7.8(e) muestra el grafo del producto cruzado donde se aprecian cuatro capas independientes (colores) $\{1, 5, 6, 7\}; \{2\}; \{3\}$ y $\{4\}$. Sin embargo, si la división en capas se hiciera directamente en 2D y no como resultado de un producto cruzado de capas 1D la cantidad de grupos independientes se puede disminuir de 4 a 3, resultando en los grupos de reglas $\{1, 4, 6\}; \{5, 7\}$ y $\{2, 3\}$ como ilustra la Fig. 7.8(f). En definitiva, en el primer caso la división en capas se realiza en 1D tomando luego su producto cruzado; mientras que en el segundo caso la división de capas se realiza en base al espacio 2D directamente. En el primer caso el costo computacional es menor ya que se consideran intervalos 1D; sin embargo se deben realizar los productos de *todas* las capas 1D para obtener los resultados 2D. En el segundo caso el costo computacional es mayor, obteniendo beneficios en cuanto al grado de optimización de las capas 2D.

7.1.3.4. Capas independientes con solapamiento interno máximo controlado

Otro conjunto de trabajos analizados adoptan técnicas de sectorización *con solapamiento interno controlado* para disminuir los principales problemas de los esquemas basados en división en espacios ortogonales (descomposición) y no-ortogonales (árboles). Ambos esquemas pueden llegar a tener consumo de memoria $O(N^k)$, en el primer caso debido a la existencia de pseudo-reglas y en el segundo por el efecto de duplicación de reglas; ambos factores provenientes del problema de solapamiento en rulesets. Como se demuestra en estos trabajos, la aplicación de estratificación a estos esquemas junto con la relajación de la exigencia de que los intervalos sean disjuntos en cada capa puede reportar un compromiso positivo costo-beneficio.

En [116] se considera el esquema de cross-producting [96] el cual ofrece máxima velocidad de clasificación; sin embargo se observa que la necesidad de pseudo-reglas produce explosión de memoria. Se ataca entonces este problema mediante estratificación (referida en ese trabajo como *particionado*) del ruleset en *grupos independientes*. Los autores observan que enfoques previos como [154] o [156] llevan a un número de capas independientes muy impredecible, por ejemplo entre 43 y 61 capas para el caso de [154], lo que es una seria limitación para implementaciones en hardware. En consecuencia, se plantea fijar un máximo número de grupos P, definidos en 1D, permitiendo en cada uno de ellos solapamientos que involucren un máximo de B reglas. $P-1$ de estos grupos son grupos independientes del tipo de [154] (Fig. 7.6) con solapamiento interno controlado, mientras que el grupo restante es una tabla de cross-producting genérica que contiene las reglas restantes. La búsqueda en cada campo se realiza mediante árboles de decisión del tipo *quadtree* (cuatro hijos por nodo). Para $P = 4, B = 2$ se logra reducir el consumo de memoria de [154] en un orden de magnitud para rulesets del tipo ACL (listas de control de acceso).

EffiCuts [93] y ParaSplit [61] realizan particionado del ruleset en kD, aplicando luego el esque-

ma basado en árboles HyperSplit a cada sub-grupo así definido; donde cada árbol utiliza todo el encabezado. El consumo de memoria de esquemas basados en árboles se atribuye a la existencia de solapamientos en el espacio kD; en especial en rulesets complejos como FW o IPC y en menor medida ACL. Esto fue discutido en referencia a la Fig. 4.1 en la Sec. 4.2. Los autores observan que aún utilizando árboles optimizados HyperSplit [92], por ejemplo el ruleset $FW1\_10K$ de [134] puede producir hasta 7000 replicaciones de reglas. Para disminuir este consumo mejorando la escalabilidad de tales esquemas, se busca asignar las reglas que se solapan a sub-grupos distintos. De este modo, dentro de cada sub-grupo el grado de solapamiento y consecuente consumo de memoria deberían ser menores. Esto es similar a lo realizado en [116], pero definiendo los grupos directamente en k dimensiones. Persiguiendo este objetivo, EffiCuts agrupa reglas en árboles independientes, reduciendo significativamente la replicación en un factor de 57 respecto a HyperSplit; sin embargo el algoritmo utilizado puede generar una cantidad excesiva $O(2^k)$ de grupos, por ejemplo 32 grupos para rulesets basados en quíntuplas IP y hasta 4000 grupos para rulesets Openflow de 12-tuplas. ParaSplit busca mejorar el algoritmo de particionado, minimizando los solapamientos en cada grupo sin conducir a un número de grupos excesivo. Encontrar una solución óptima a este problema en kD es muy difícil, por lo que ParaSplit utiliza dos etapas de procesamiento. En primer término, se convierten reglas en el espacio kD a puntos en el espacio $2kD$ y se aplica el algoritmo de clustering *k-means* con diferentes heurísticas para agrupar de estos puntos. En segundo lugar, se aplica al algoritmo de *Simulated Annealing* a fin de optimizar aún más el particionado. Respecto a EffiCuts, esta técnica logra reducciones de $20\,\% - 500\,\%$ en la memoria requerida. Se aplica HyperSplit [92] a cada uno de los grupos en forma paralela, agregando finalmente los resultados de cada grupo. Mediante técnicas de pipelining en FPGAs se logran desempeños de 120 Gbps; además, dado que el consumo de memoria se reduce, se pueden utilizar hasta 10 instancias de ParaSplit en un FPGA conduciendo a velocidades totales de 1 Tbps para 10K reglas. Esto es por supuesto muy dependiente del ruleset considerado y requiere extensivo pre-cómputo para obtener el particionado adecuado.

La agrupación de reglas se aproxima en ParaSplit como un problema de matemática combinatoria. ParaSplit define que, dadas N reglas diferentes definidas sobre k campos ortogonales, el solapamiento existente entre ellas puede llegar a generar $|UR_{1...k}|_{max} = (2.N + 1)^k$ regiones diferentes en el peor caso, de modo que la cantidad de regiones UR aumenta exponencialmente con la cantidad de campos y linealmente con la cantidad de reglas (es decir UVs kD). ParaSplit busca disminuir este límite y la complejidad asociada mediante la agrupación inteligente de reglas. En términos generales, los autores argumentan que dividiendo adecuadamente el ruleset en G grupos, cada grupo debería contener en promedio N/G reglas siendo ahora la complejidad del ruleset la suma de sus partes, es decir $O(G.(N/G)^k) = N/(G^{k-1})$. Como resultado, la complejidad de clasificación debería reducirse por un factor G^{k-1}. Existen múltiples maneras de agrupar N reglas en G grupos, dadas por $S(N,1) + S(N,2) + ... + S(N,M)$ si $N \geq M$ y $S(N,1) + S(N,2) + ... + S(N,N)$ si $N \leq M$, donde $S(N,x)$ es el número de Sterling. De estas múltiples opciones, deberían seleccionarse las que den como resultado grupos con mínimo solapamiento interno en k dimensiones, de modo que al aplicar clasificación basada en árboles en cada uno de ellos la replicación de reglas sea reducida. Dada la cantidad de combinaciones posibles, seleccionar estas agrupaciones más adecuadas es muy difícil. Para buscar dichas agrupaciones en un tiempo razonable con cierto margen de aproximación, ParaSplit evalúa dos alternativas: i) mapeo rango-punto y aplicación del algoritmo de clustering

k-means al mapa de puntos obtenido, en base a ciertas heurísticas; y ii) el algoritmo de simulated annealing. k-means es más efectivo en la relación overlap (o #de hojas)/iteraciones, si bien no llega a al reducción deseada. Simulated Annealing, en tanto, por sí solo no obtiene resultados satisfactoris en un tiempo razonable. Por ello, ParaSplit finalmente opta por aplicar k-means y *sobre esos resultados* Simulated Annealing.

ParaSplit introduce algunos conceptos que creemos necesario re-considerar para nuestro trabajo actual. Por un lado, es importante corregir que el límite $|UR| = 2|UV|-1$, válido para una dimensión, no se puede simplemente extrapolar al caso kD para obtener el límite mencionado $|UR_{1...k}|_{max} = (2.N+1)^k$, ya que para obtener este peor caso se deben compartir UVs 1D. Este tema se profundiza en la Sec. 7.3. Por otro lado, y aún más importante, las heurísticas adoptadas en k-means no son eficientes para el objetivo planteado. Ya que por sí solo k-means obtiene mejores resultados que Simulated Annealing, consideramos que reemplazándolas por otras se podría llegar a resultados satisfactorios sin necesidad de aplicar Simulated Annealing. Repasaremos brevemente la metodología de ParaSplit a fin de justificar nuestras observaciones.

ParaSplit realiza la transformación de rangos a puntos a fin de agrupar posteriormente estos puntos mediante un algoritmo k-means. En la Fig. 7.9(a) se observa un conjunto de reglas en 1D, que deben ser separados en clusters, mientras que en la Fig. 7.9(b) se ilustra la transformación rango-punto para el caso de la reglas $R1$. Esta regla está definida por el rango 1D $(s_1 < x < e_1)$ o lo que es lo mismo $(s_1 < F_A), (-e_1 < F_A)$, por lo que el punto asociado en el plano 2D $(s; -e)$ será $(s_1; -e_1)$. Es de destacar que la zona de este plano donde $s > e$, rayada a 45º en la Fig. 7.9(b), no es válida para rangos reales, por lo que cualquier punto válido estará en la zona limitada por el eje $s = 0$, la recta $s = -e$ y $e = 2^m$. Por otro lado, cualquier key de entrada es un valor exacto por lo que los valores de key se encuentran sobre la diagonal $e - s = 0$ o lo que es lo mismo $s = e$. De este modo, todos los paquetes cuyo campo de encabezado F_A coincida con R_1 se encuentran en el segmento indicado de la Fig. 7.9(b). Ahora bien, teniendo en cuenta que para cualquier regla válida en la realidad se debe cumplir $s \leq e$, podemos afirmar que una regla R_2 *NO* se solapará con R_1 si se cumple $(e_1 < s_2)$ || $(e_2 < s_1)$, es decir que el rango de la segunda regla inicie luego de finalizar el rango de la primera ó finalice antes de comenzar la primera (se profundiza esto en el Cap. 5). En la Fig. 7.9(c) las áreas rayadas a -45º definen el espacio 2D donde se cumplen estas condiciones. En el espacio sombreado en gris, en tanto, ninguna de las dos condiciones se cumple, por lo que las reglas ubicadas en ese espacio se solaparán con R_1. En la Fig. 7.9(c) se ilustran estos hechos para los rangos $R_2 - R_{10}$ de las Fig. 7.9(a), donde se puede corroborar la validez de nuestras afirmaciones. Asimismo, en las Figs. 7.9(d) y 7.9(e) se ilustra el mapeo rango-punto para los campos F_A y F_B del ruleset de ejemplo de la Fig. 4.4(b), mientras que la Fig. 7.9(f) muestra el mapeo rango-punto del campo F_A para dicho ruleset.

Sobre esta base ParaSplit busca dividir el ruleset en grupos con mínimo solapamiento interno de reglas; para ello utiliza el algoritmo de clusterización k-means basado en tres heurísticas principales, a saber:

1. *minimización de la distancia promedio entre puntos en cada grupo*, es decir que cada regla analizada se incorpora al cluster con rangos mas similares al suyo

2. *maximización de la distancia promedio entre puntos en cada grupo*, donde para cada regla analizada se la incorpora al grupo con rangos mas lejanos a ella.

3. *distancias similares al origen en cada grupo*, donde se busca un balance entre scopes en cada grupo.

Los autores argumentan que la primera heurística produce el menor consumo de memoria al aplicar HyperSplit, por lo que es adoptada. La segunda heurística, en tanto, se indica como la que produce los menores solapamientos dentro de cada grupo. Sin embargo, de acuerdo a nuestro razonamiento anterior respecto a los solapamientos, ninguna de estas heuristicas reducen por definición los solapamientos; ya que éstos no dependen directamente de la amplitud de rangos ni de la distancia al origen. Por ejemplo, en nuestro ejemplo de la Fig. 7.9(c) se podría pensar que agrupar las reglas $R2 - R3$, $R4 - R6$, $R7 - R9$, y $R10$ por sus distancias una a otra reduce el consumo de memoria, mientras que en la Fig. 7.9(a) observamos que estas agrupación no produce beneficios. Según la heurística 2), en tanto, se buscaría agrupar reglas por sus mayores distancias, lo que no reduce necesariamente su solapamiento. Esto no significa que las heurísticas utilizadas no puedan reportar beneficios dados los patrones de reglas utilizados para evaluación; sin embargo tales beneficios pueden aun ser mayores y más previsibles atacando directamente el problema de solapamiento entre reglas. De esta manera, se espera asimismo eliminar la necesidad de aplicar Simulated Annealing en una segunda etapa, el cual consume excesivas iteraciones para el mismo resultado.

Del análisis anterior se extraen las siguientes observaciones generales:

- las técnicas de estratificación analizadas consideran (a) dividir en grupos kD sin solapamiento y luego aplicar en cada grupo alguna técnica multi-dimensional tal como árboles (Efficuts, Parasplit), (b) particionar en 1D y luego aplicar XPROD entre las capas definidas (ppc), o (c) particionar en dD ($1 \leq d \leq k$) y luego buscar en cada sub-grupo en k-d dimensiones mediante alguna técnica multidimensional (indsets (d=1), prepart (d=2), coarsesets (d=1, sets con solapamiento limitado y tabla reducida de XPROD), controlled XPROD (d=1, sets limitados sin solapamiento y tabla reducida de XPROD)).

- la estratificación se puede usar para (a) *filtrar* búsquedas multi-dimensionales, a ejecutarse mediante alguna técnica común como búsqueda lineal o trees sobre rulesets reducidos en las dimensiones restantes; o (b) ingresar a *tablas de XPROD* entre capas. El caso (b), si bien aún requiere de tablas XPROD, puede reportar ventajas respecto al XPROD o distXPROD original ya que en cada capa la cantidad de URs es exactamente igual a la cantidad de UVs.

- la estratificación se puede aplicar en 1D o en dD ($1 < d \leq k$). El primer caso tiene mínima complejidad de pre-cómputo pero la disminución de complejidad de búsqueda puede no ser significativa. A medida que aumentamos d puede reducirse notablemente la complejidad de búsqueda pero se requiere mayor pe-cómputo (se deben considerar múltiples dimensiones a la vez para definir los grupos).

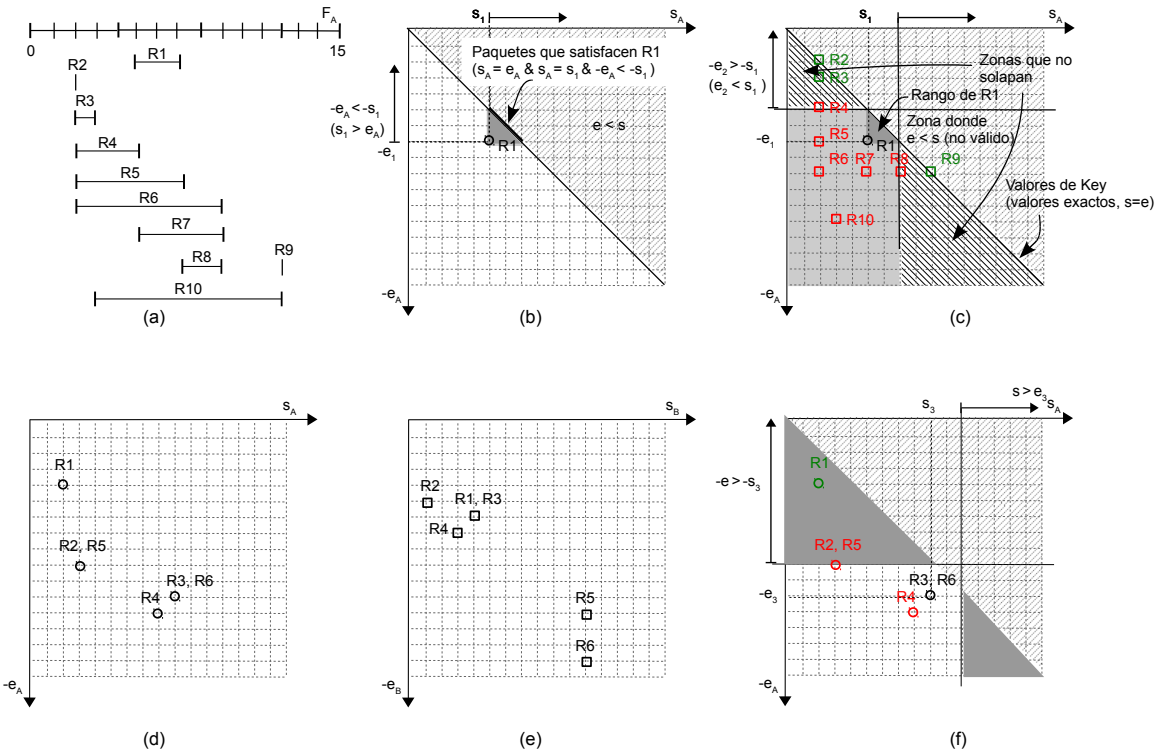


Figura 7.9: (a) Rangos en el espacio 1D F_A, (b) Conversión de un rango $[s, e]$ en 1D a un punto (s; -e) en 2D, (c) casos de solapamiento en el espacio 2D $\{s_A, -e_A\}$, (d)(e) puntos correspondientes a los rangos en F_A y F_B del ruleset en la Fig. 4.4(a) respectivamente, (f) problemas de solapamiento en el espacio $\{s_A, -e_A\}$

7.2. Opciones de implementación para el caso general

Habiendo repasado y analizado las técnicas generales para disminución de complejidad del ruleset, nos concentramos ahora en los posibles esquemas para implementación de clasificación por descomposición de complejidad reducida, ya sea por sectorización o estratificación, uni- o multi-dimensional. Si bien son aplicables numerosas variantes, tomaremos como ejemplo los estilos de estratificación propuestos en [121]. En general, las arquitecturas de agregación que se analizarán son:

- mediante TCAM, según lo propuesto en [121]
- la arquitectura propuesta en esta Tesis, utilizando en particular elementos de procesamiento híbridos memoria-lógica
- mediante bloques de memoria RAM, como se propone en [106] y [83] entre otros

El trabajo [121], también referido aquí como Clasificación Paralela de Paquetes (Parallel Packet Classification, PPC), propone el uso de memoria TCAM para agregar resultados de lookup, con el propósito original de reducir el ancho del key que ingresa a la TCAM. La memoria TCAM recibe como key la concatenación de los resultados de lookup de los múltiples campos, y los mapea

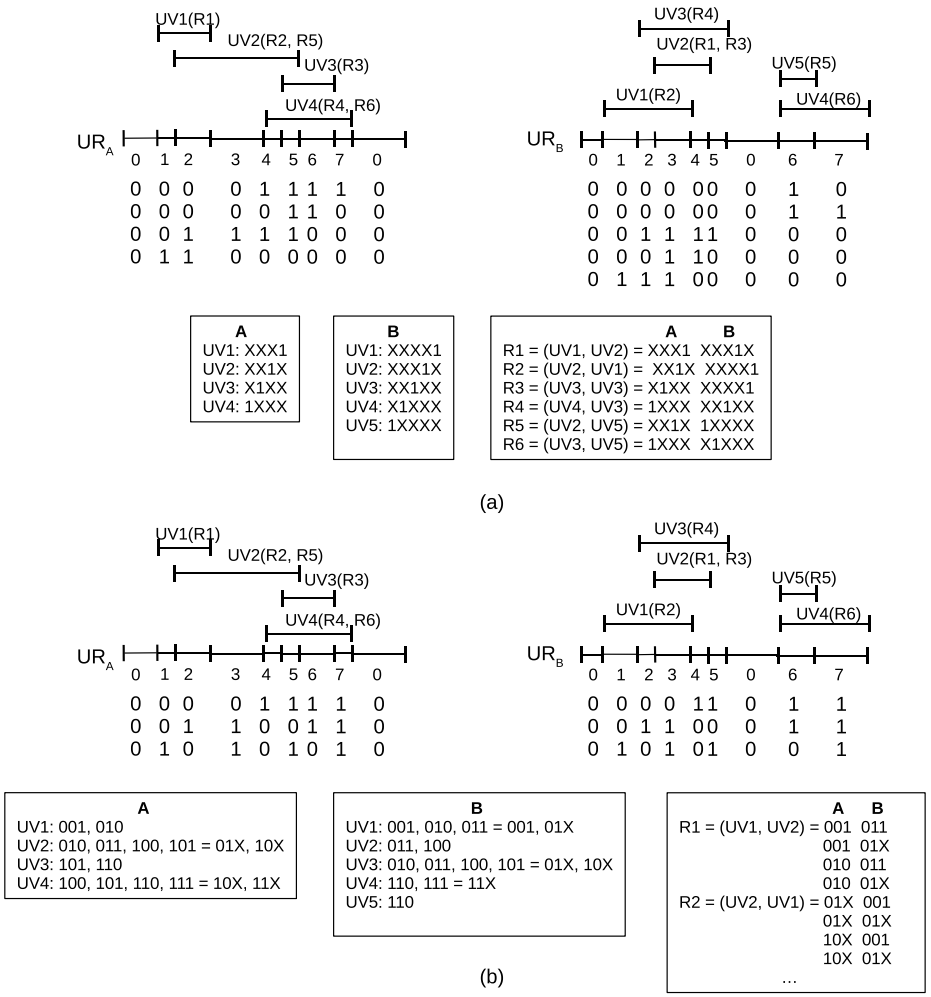


Figura 7.10: Implementación de PPC mediante TCAM: (a) uso de metadato DBFV, (b) uso de metadato URID

directamente a las reglas kD. Su ventaja con respecto a un esquema de direccionamiento directo en memoria es que puede comprimir múltiples direcciones mediante el uso de bits "don't care" (X). Este esquema puede en principio utilizarse para implementar cualquiera de los esquemas de estratificación mencionados, si bien su rendimiento se ve sensiblemente afectado. Para ver este efecto, se ilustra en la Fig. 7.10(a) un ejemplo de mapeo del ruleset de la Fig. 4.4(b); en este caso se utiliza el metadato DBFV (Fig. 7.4(b)) al igual que lo hacemos en nuestro esquema DCFV del Cap. 4. Como se ve en la Fig. 7.10(a), para este caso cada regla corresponde a una entrada en la TCAM. En la Fig. 7.10(b) en tanto, se muestra el extremo opuesto donde se utiliza el metadato URID (Fig. 7.4(g)), esencialmente utilizado en RFC. En este caso observamos que PPC no es eficiente ya que requiere de múltiples entradas para representar cada regla. Esto es debido a que la memoria TCAM sólo es capaz de mapear prefijos a bits individuales (keys a reglas), de forma similar a lo que sucede en la expansión de rangos. Entre ambos metadatos extremos (es decir DBFV y URID) se pueden implementar tambien otras opciones como las mostradas en la Fig. 7.4. En general, se observa que PPC requiere múltiples entradas cuando i) existe solapamiento interno en las capas definidas (ya que una regla involucra un rango arbitrario de keys en lugar de un prefijo) (por ejemplo en PPC-3

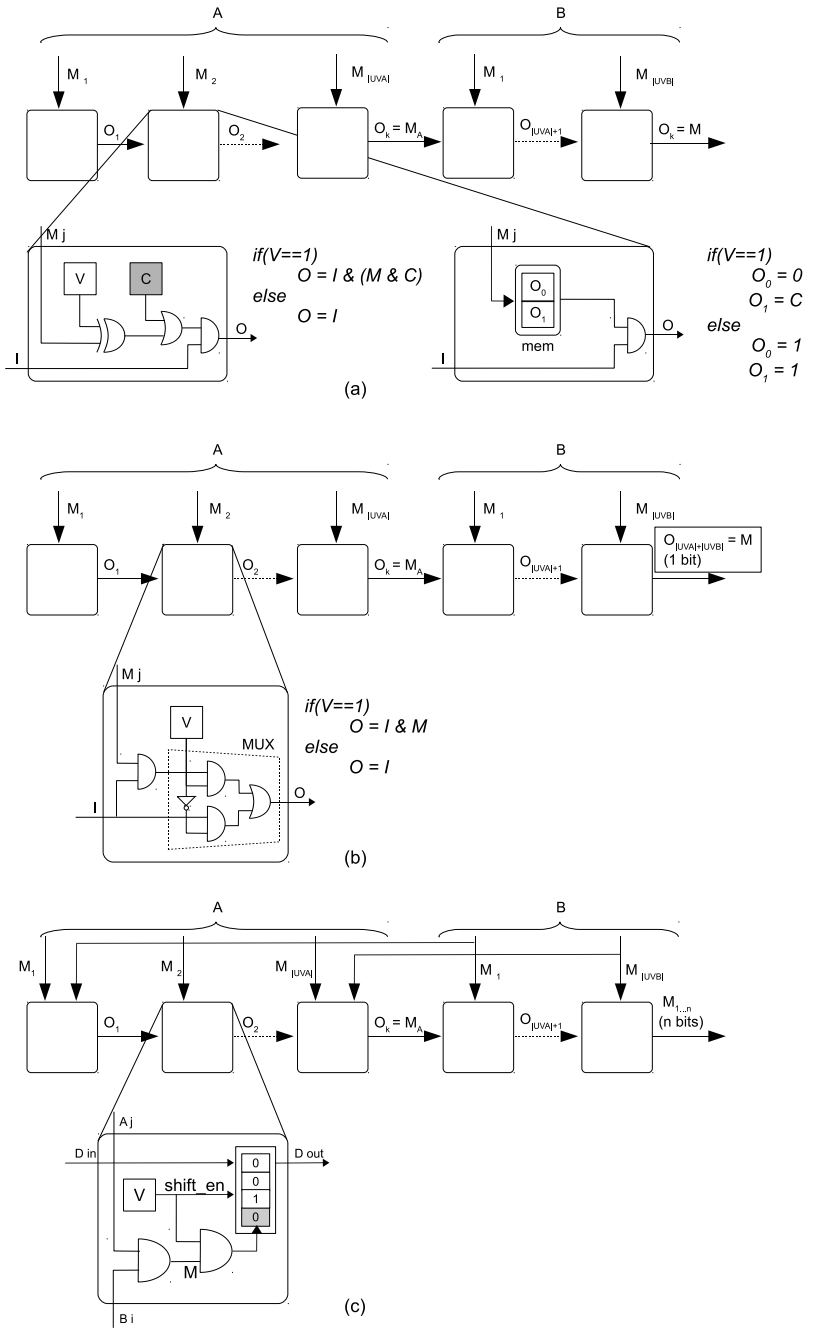


Figura 7.11: (a) Línea de match PPC mediante celdas TCAM emuladas mediante lógica y mediante memoria, (b) línea de match PPC implementada mediante PE de 1 bit, (c) línea de match DCFV mediante PE de 4 bits

o PPC-4), o cuando ii) el metadato se basa en etiquetas en lugar de bitmaps, ya que la salida de la TCAM no representará reglas sino bits afectados por múltiples keys (por ejemplo UVIDs, PPC-3, o PPC-4).

Nos preguntamos ahora si los casos contemplados en PPC pueden ser implementados mediante nuestra arquitectura de PEs híbridos. Para ello, se comparará el concepto de la memoria TCAM con

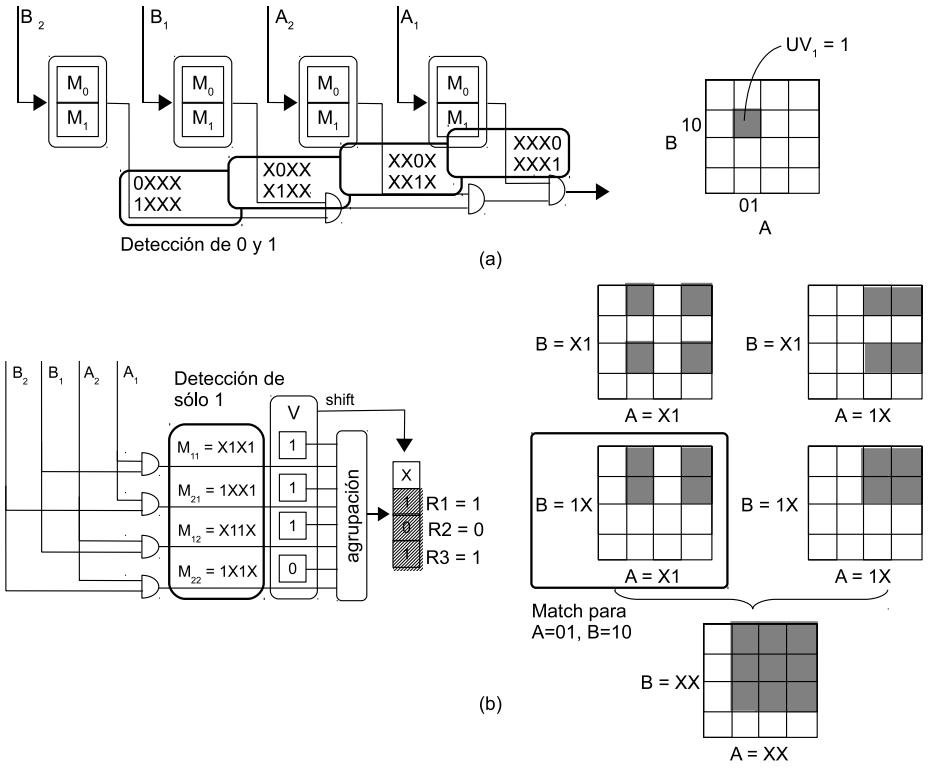


Figura 7.12: (a) Mapeo de múltiples bits (key) a 1 bit (regla) en línea de TCAM, (b) mapeo de múltiples bits (resultados lookup 1D) a múltiples bits (resultados agregación) en DCFV

el de nuestro PE. En la Fig. 7.11(a) se ilustra la estructura de una celda TCAM emulada tanto en lógica como en memoria (ver Cap. 3), y su aplicación dentro de una línea de match para una regla. Como se observa, cada celda recibe el resultado de la anterior, toma su conjunción con el resutado propio y lo propaga hacia la siguiente celda en su línea. A su vez, recibe un bit del key (A_j) que evaluará. Internamente, dos bits determinan el resultado propio de la celda: el bit de *valid (V)*, que indica si la celda almacena un valor exacto o un valor "don't care" (X), y el bit de *contenido (C)*, que indica, en caso de ser $V = 1$, el valor (0 o 1) contra el cual se compara el bit A_j. Como vemos, no existe posibilidad de propagar más de 1 bit con esta celda. Para los casos PPC donde no existe solapamiento interno a cada capa, es decir donde cada regla es una entrada de TCAM, se observa que las celdas TCAM desperdician el bit C, ya que sólo tiene relevancia *qué UVs se combinan* para la regla (línea TCAM) considerada. Para los UVs válidos en esta línea, interesará si esos UVs matchean o no, es decir el estado de M_j, mientras que C se debe fijar en 1 para dejar pasar este estado. De este modo, en la versión lógica de la Fig. 7.11(a) se puede eliminar el bit C y su lógica asociada, mientras que en la versión que utiliza el bloque *mem* ambas posiciones almacenarían sus propias direcciones (0 o 1) por lo que se puede propagar directamente M_j. Utilizando FPGAs, se puede eliminar este almacenamiento innecesario para reducir la memoria a la mitad, manteniendo todas las demás características de PPC. Esto se ilustra en la Fig. 7.11(b). Mas aún, se pueden ahora propagar *múltiples resultados de match* por línea implementando PEs de dos entradas, como muestra la Fig. 7.11(c). Los factores claves para la implementación de estas arquitecturas con el bit V y el uso de bitmaps.

Ahora bien, el costo que se paga por este ahorro de memoria y propagación de múltiples matches por línea es que los PEs son ahora capaces de determinar sólo si los matches de interés están en uno, pero no el estado individual de cada match en la línea. En otras palabras, el esquema es capaz de individualizar el estado de los bits de interés, pero ignora el estado de los bits que no son de interés; esto es debido a que está diseñado para trabajar estrictamente con bitmaps. En el caso de trabajar con etiquetas, la información consta de combinaciones de todos los bits involucrados, por lo que la arquitectura se debe modificar. En la Fig. 7.12(a) se muestra efectivamente que, dados dos campos A y B de dos bits cada uno, el esquema TCAM es capaz de determinar la ubicación exacta de la combinación de matches en el plano (A, B). Por ejemplo, para el caso $A_1 = 1,\ A_2 = 0,\ B_1 = 0,\ B_2 = 0$ se puede determinar con certeza la ubicación sombreada. En la Fig. 7.12(b), en tanto, vemos el resultado de ingresar con el mismo set de matches a una arquitectura DCFV 1d1b. En este ejemplo, el vector V indica que las combinaciones válidas de matches son $M_{11} = (A_1 = 1, B_1 = 1) = R1$, $M_{21} = (A_1 = 1, B_2 = 1) = R2$, y $M_{12} = (A_2 = 1, B_1 = 1) = R3$, mientras que la combinación $M_{22} = (A_2, B_2)$ no existe en este ruleset. Las cuatro posibles combinaciones válidas cubren en conjunto las regiones sombreadas en el plano de la Fig. 7.12(b). Cabe mencionar que las combinaciones $(A_1 = 1, A_2 = 1)$ y $(B_1 = 1, B_2 = 1)$ no son consideradas ya que corresponden a un mismo campo. Con este esquema, una combinación entrante $A = 01, B = 10$ como la de la Fig. 7.12(a) se podría encontrar en múltiples puntos del plano ya que los valores $A_2 = 0, B_1 = 0$ no son de interés para un esquema de bitmaps. Esto se ilustra en el plano (A, B) recuadrado en la Fig. 7.12(b), donde si bien se encuentra la combinación buscada existen otras tres posibles ubicaciones que no se pueden descriminar.

Finalmente, se puede utilizar memoria RAM para realizar agregación de resultados de lookup. Este esquema permite implementar cualquiera de los casos de la Fig. 7.4 a costa de mayor costo por su expansión de direccionamiento. Sin embargo, las técnicas mencionadas de sectorización o estratificación pueden mitigar este problema.

7.3. Estudio de patrones de clasificación

7.3.1. Peores casos

A fin de introducir nuestro estudio de patrones de solapamiento, comenzaremos por considerar casos teóricos que dan una noción de las máximas complejidades posibles. Los factores esenciales que impactan en la complejidad de clasificación son (a) el número de regiones únicas ($|UR|$) definidas, que determina la cantidad de acciones diferentes a considerar; y (b) el número máximo $|UV|_{max}$ de valores únicos involucrados en tales regiones, que determina el procesamiento necesario para tomar una decisión. Sobre esta base, los peores casos se pueden fijar como aquéllos que logran máximos en las relaciones $|UR|_{max}/|UV|$ y $|UV|_{max}/|UV|$, es decir que logran las peores condiciones con el mínimo necesario de UVs. Los límites teóricos de $|UR|$ están afectados, a su vez, por (i) el número de bits m de los campos analizados, (ii) el tipo de especificación (valores exactos, prefijos, rangos arbitrarios) en cada uno de ellos, y (iii) la cantidad de reglas.

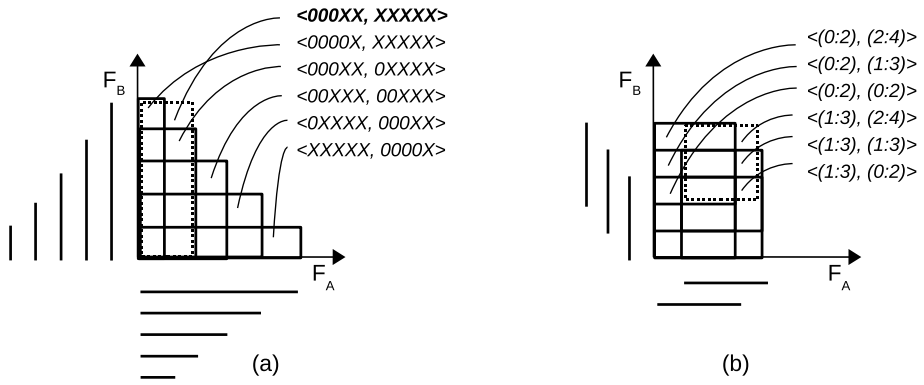


Figura 7.13: Peores casos de solapamiento 2D: (a) Reglas especificadas según prefijos, (b) reglas especificadas según rangos arbitrarios

Para un campo de ancho m, el máximo número posible de regiones es $|UR|_{max} = 2^m$, es decir, cada valor del campo se encuentra en una región diferente. La cantidad de UVs necesarios para producir este $|UR|_{max}$ depende del *scope máximo* S_{max} que abarcan los UVs, mientras que el *desplazamiento (offset) mínimo* O_{min} entre ellos debe ser 1 para producir este caso. Esto se muestra en la Fig. 7.14(a) para el caso de rangos arbitrarios, donde se muestran diferentes compromisos entre los factores involucrados. En particular, se observa que si $S_{max} > (2^k)/2$ los scopes deben ser heterogéneos para lograr $|UR|_{max}$. Asimismo, como se muestra en los dos últimos casos de la Fig. 7.14(a), si $S_{max} > |UV|$ se logran mayores valores de $|UV_{max}|$ manteniendo el caso $|UR|_{max}$. En la Fig. 7.14(b) se muestra la situación para especificaciones de prefijo. En este caso, se observa que el límite $|UR|_{max} = 2^m$ sólo puede lograrse involucrando prefijos de longitud máxima (valores exactos) como en los dos primeros casos de la Fig. 7.14(b), ya que los prefijos no son capaces de generar solapamientos parciales. Los tres últimos casos de la Fig. 7.14(b) muestran otros compromisos de solapamiento que sin embargo producen $|UR|_{max} \leq (2^m)/2$.

Para evaluar peores casos en filtros kD se diferencian dos casos; a saber, aquél donde partimos de campos independientes y analizamos los peores casos resultantes de su combinación, y aquél donde analizamos un número pre-definido de filtros kD. En el primer caso, al combinar filtros 1D en filtros kD, $|UR|_{max}$ estará determinado por sus combinaciones más desfavorables. El peor caso teórico proviene de la combinación de los peores casos teóricos 1D, es decir $|UR_{1...k}|_{max} = \prod_{j=1}^{k} |UR_j|_{max}$, lo cual es $|UR_{1...k}|_{max} = \prod_{j=1}^{k} 2^{m_j}$ para el caso de rangos arbitrarios / prefijos incluyendo valores exactos y $|UR_{1...k}|_{max} = (\prod_{j=1}^{k} 2^{m_j})/2$ para prefijos que no incluyen valores exactos. Los peores valores de $|UV|$ y $|UV|_{max}$, en tanto, resultan $|UV_{1...k}| = \prod_{j=1}^{k} |UV|_j$ y $|UV|_{max} = \prod_{j=1}^{k} |UV|_{max}j$ respectivamente. Esto se aprecia en la Fig. 7.15(a) para combinaciones 2D de los rangos arbitrarios ilustrados en la Fig. 7.14(a), y en la Fig. 7.15(b) para combinaciones 2D de los prefijos ilustrados en la Fig. 7.14(b). Si bien se podrian considerar muchas otras combinaciones, estas se extraen como las más críticas.

A la inversa, cuando se parte de un número pre-definido N de filtros kD y se buscan sus peores casos de solapamiento en kD y 1D, el análisis es diferente. Como se profundizará más adelante al hablar de rulesets reales, para el caso de prefijos las regiones kD definidas por n filtros kD pueden alcanzar valores $|UR_{1...k}| = \sum_{i=1}^{N} i = [N \cdot (N+1)]/2$ como se muestra en la Fig. 7.13(a) para 2 dimensiones y campos de 5 bits. Este patrón de reglas, donde los UVs 2D no comparten UVs 1D en absoluto, es el peor caso de solapamiento para reglas basadas en prefijos; por ejemplo vemos que si se suma un nuevo UV 2D $< 000XX, XXXXX >$ re-utilizando uno de los UVs 1D (línea de trazos), las URs 2D no continúan escalando según este factor. Para el caso de N reglas 2D basadas en rangos arbitrarios, en tanto, el peor caso es $|UR_{1...k}| = 3N - 3$ como se ilustra en la Fig. 7.13(b) para un ruleset 2D.

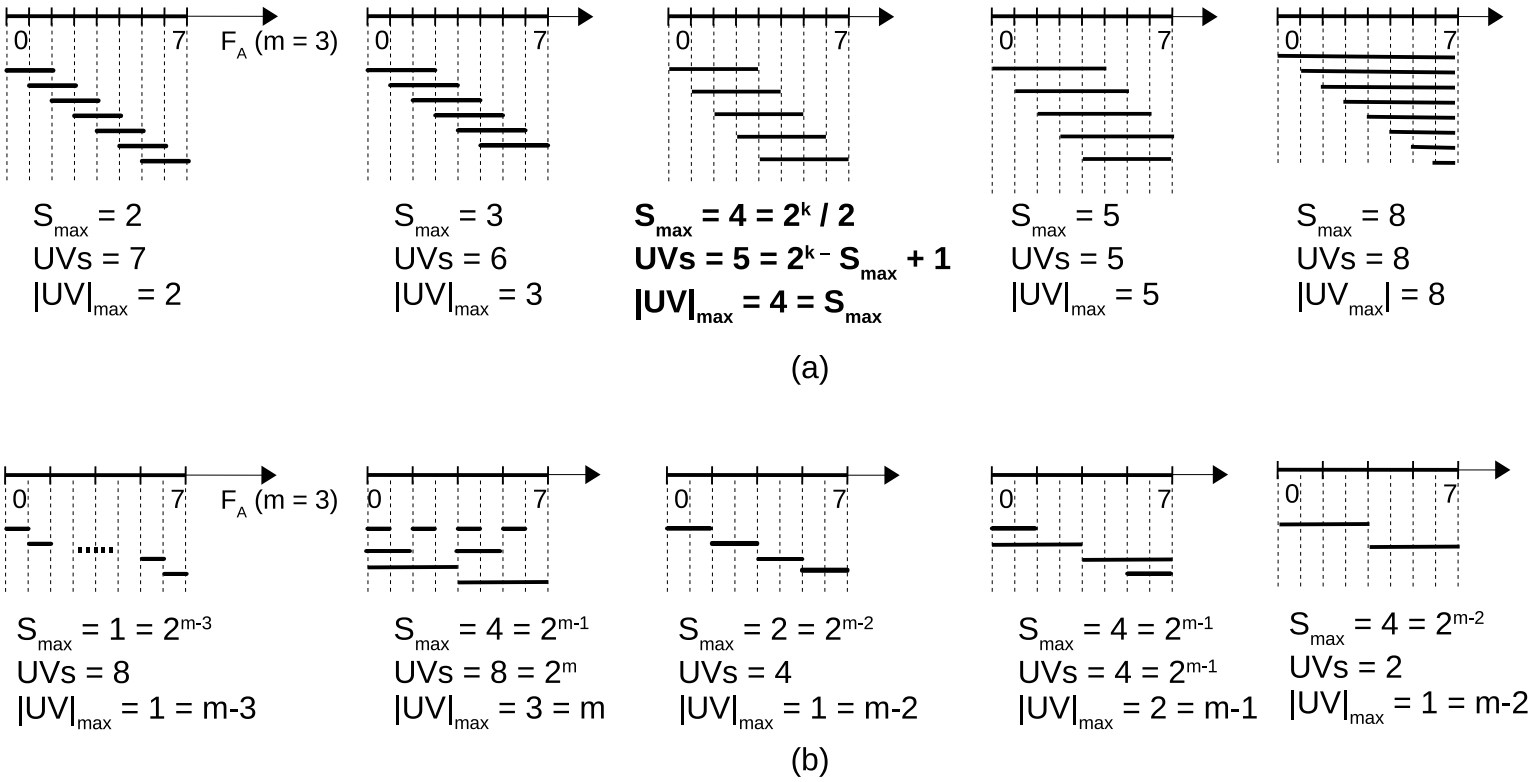


Figura 7.14: Peores casos para filtros en un campo: (a) especificaciones de rango arbitrario, (b) especificaciones de prefijo

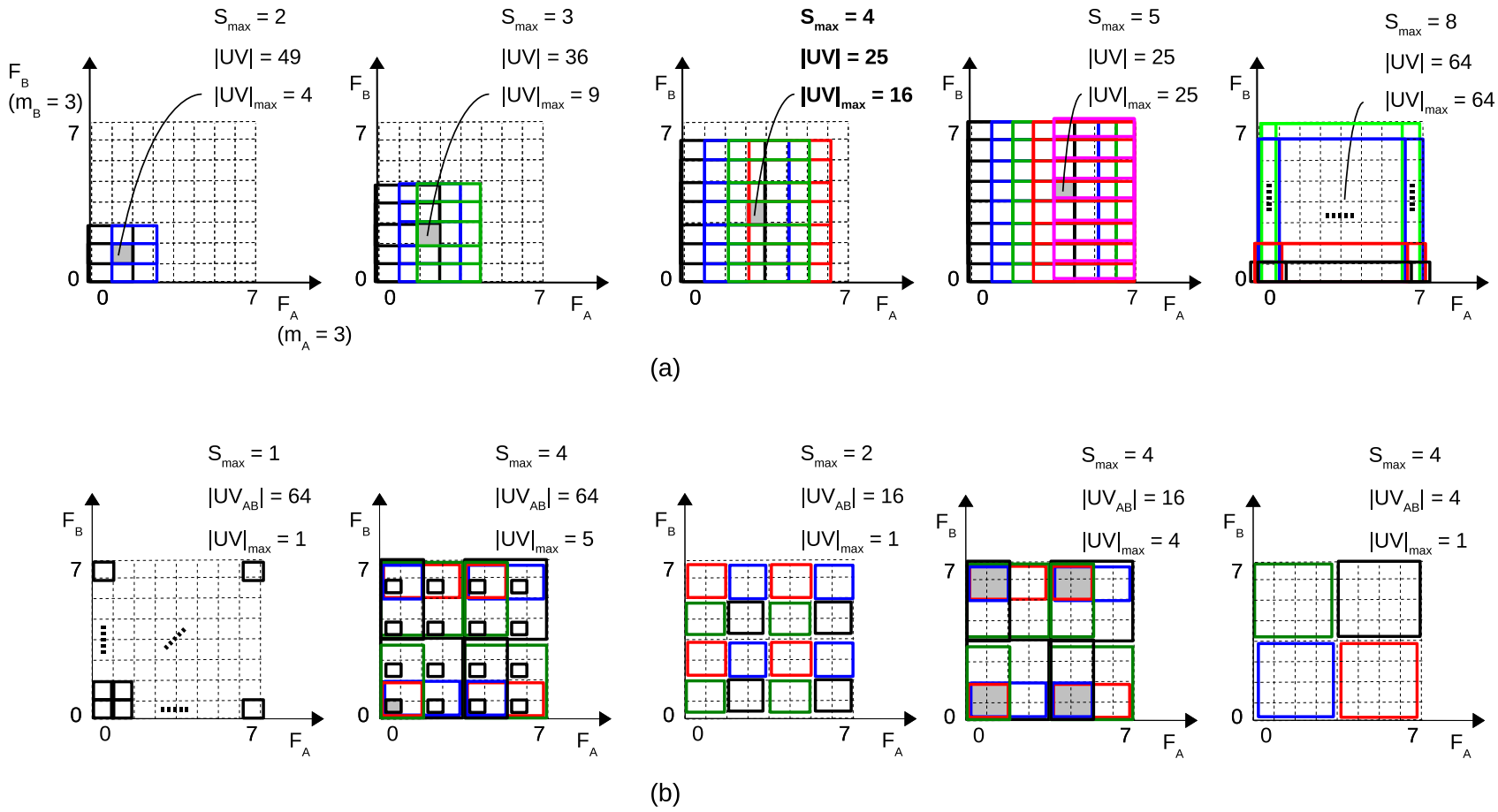


Figura 7.15: Peores casos para filtros bi-dimensionales: (a) especificaciones de rango arbitrario, (b) especificaciones de prefijo

7.3.2. Análisis y explotación de rulesets reales

Habiendo planteado los peores casos teóricos de solapamiento, surge la necesidad de compararlos con la complejidad de rulesets reales a fin de determinar su relación con aquéllos y elaborar estrategias adecuadas para su explotación. Sobre esta base, se plantea la probable evolución de estos rulesets y las necesidades que éstos plantearán en aplicaciones futuras. [103] es uno de los primeros trabajos en analizar tablas de reglas reales y extraer su estructura para explotarla en algoritmos de clasificación. Sobre esta base, los autores analizan el costo de mantener reglas derivadas de las originales que representen directamente sus solapamientos, es decir sus URs. El acceso a rulesets es muy restringido por motivos de seguridad y confidencialidad, por lo que este tipo de análisis estadístico resulta muy útil al diseñar nuevos esquemas de clasificación. Los autores accedieron a cuatro listas de control de acceso (Access Control Lists, ACLs) utilizadas en firewalls, tres de ellas provenientes de proveedores de servicio internet (Internet Service Providers, ISPs) de gran tamaño, y la restante de una intranet corporativa pequeña. En particular, se observó que la especificación de una regla se puede dividir en dos entidades lógicas o etapas. La primera de ellas es una tupla en dos dimensiones, formada por direcciones IP de origen y destino (SrcIP/DstIP), que define caminos de red; mientras que la segunda está formada por un conjunto de campos de transporte (números de puerto de origen y destino SrcPT/DstPT, protocolo Prot) en (k-2) dimensiones. Debido a la naturaleza jerárquica de las redes IP, la primera involucra búsquedas basadas en prefijos, mientras que la segunda involucra busquedas de tipo general como prefijos, rangos arbitrarios de valores o valores exactos. En general, la cantidad de caminos de red de la primera etapa es mucho mayor que las posibles especificaciones de aplicación. Si bien el análisis se realiza sobre estos rulesets, las características observadas responden a prácticas standard de los administradores de red, por lo que las conclusiones obtenidas son generales.

Un aspecto importante para evaluar complejidades son los *patrones geométricos 2D* preponderantes en los rulesets. La combinación de dos valores exactos define un punto, la de valor exacto-wildcard/prefijo define una línea, la de dos prefijos define rectángulos, mientras que la de dos wildcards abarca todo el espacio de clasificación.

Las duplas IP, llamadas en ese trabajo *filtros*, se dividen en *parcialmente especificadas*, cuando ellas involucran la especificación wildcard (*) en alguno de los campos; y *completamente especificadas*, cuando su rango de valores se especifica para ambos campos. Por un lado, se observó que sólo un número reducido de filtros incluyen wildcards para el caso de los rulesets de grandes ISPs. Para los filtros completamente especificados, DstIP es una dirección determinada (de 32 bits en IPv4, que representa un *host*) mientras que SrcIP es un prefijo (de longitud menor a 32 bits en IPv4, representando un conjunto de hosts en un *dominio de direccionamiento IP*). Por otro lado, se caracterizaron los *solapamientos* entre filtros, los que determinan el numero de filtros coincidentes con un determinado paquete y en definitiva la complejidad de clasificación. observando que la mayor parte de ellos se deben a filtros parcialmente especificados. Ya que éstos son minoría, se espera que la cantidad y complejidad de solapamientos de los rulesets reales sea bastante menor a los límites teóricos.

Los tres rulesets de grandes ISPs demostraron preponderancia de filtros totalmente especificados ya que sus reglas responden a políticas administrativas entre dominios IP dentro de las redes de

estos ISPs. En el rulesets de la pequeña intranet, en cambio, este tipo de reglas ya ha sido implementada externamente por algun ISP por lo que los administradores se concentran en establecer políticas respecto a orígenes o destinos específicos (no ambos); por ello las reglas correspondientes son parcialmente especificadas en su mayoría.

En primer término se analiza la incidencia y estructura de reglas parcialmente especificadas. Típicamente, las ACLs son más pequeñas a medida que se alejan del núcleo de internet hacia las redes de los clientes. Las ACLs en las redes clientes suelen aplicar políticas para la conexión de su rango específico de hosts hacia afuera, involucrando por tanto wildcards en el campo DstIP. Las grandes ACLs cercanas al núcleo de intenet regulan el tráfico desde todos sus hosts a ciertos servidores de importancia, y por tanto involucran mayormente wildcards en el campo SrcIP. En cuanto a la longitud de los prefijos presentes, se observó que ésta se concentra principalmente en 8, 16, y 24 bits, correspondientes a las longitudes de clase A, B, y C utilizadas antes del surgimiento de CIDR. Un pequeño porcentaje, en tanto, se distribuye en las demás longitudes de prefijo. Esto significa que, geométricamente, la mayor parte de loa filtros parcialmente especificados son líneas y rectángulos.

En segundo término se analizan las reglas totalmente especificadas; estas pueden ser dominio-dominio (prefijos menores a 32 bits en ambos campos), dominio-host, host-dominio, o host-host (direcciones IP específicas en los dos campos). Para las ACLs de grandes ISPs, se observó que en la mayoría de los casos el origen o el destino es un host específico, correspondiente a servidores de importancia. En la pequeña ACL de la intranet, en cambio, la mayor parte de estos filtros son del tipo dominio-dominio. Las longitudes de prefijo, en tanto, se concentran en valores altos con gran incidencia de valores exactos (longitud 32 en IPv4); esto significa que geométricamente proponderan las líneas y los puntos 2D.

Considerados estos factores, se analizan en primer término los patrones de solapamiento 2D SrcIP/DstIP de los rulesets considerados. Según la cantidad de filtros SrcIP/DstIP que puedan coincidir con un paquete, será la cantidad de rangos a comparar en demás campos de aplicación. Para los rulesets analizados, se observó que un determinado paquete puede coincidir en promedio con 4 filtros; si bien esta cantidad no es muy grande, su búsqueda entre todos los filtros existentes puede significar importante procesamiento. A raíz de este problema, los autores plantean como alternativa a la comparación secuencial de los múltiples filtros IP coincidentes, el almacenamiento de *nuevos* filtros para cada solapamiento resultante. Según lo tratado en esta Tesis, esto significa almacenar un filtro por cada UR en 2D. Para prefijos IP, en el peor caso la cantidad de filtros generados es $O(n^2)$; sin embargo los resultados del análisis demuestran que en la práctica este número es varios órdenes de magnitud menor por lo que la propuesta es factible. Los autores validan su hipótesis comprobando que, en casos reales, los filtros 2D se agrupan formando grupos o *clusters*, de modo que los solapamientos 2D se producen dentro de sus respectivos clusters pero no con los demás. En particular, se pueden separar claramente los solapamientos producidos i) entre filtros parcialmente especificados, ii) entre filtros completamente especificados, y iii) entre filtros total y parcialmente especificados. En filtros parcialmente especificados (que especifican sólo uno de los campos y el restante es wildcard), los solapamientos se producen entre filtros que contienen wilcards en campos diferentes; por ejemplo no puede existir solapamiento parcial entre dos filtros SrcIP/Dst

IP que especifiquen sólo el campo SrcIP. En los rulesets analizados, la mayor fuente de solapamientos parciales 2D son los filtros parcialmente especificados, a la vez que estos filtros son un porcentaje pequeño del total de filtros presentes. Sobre la base de estas dos observaciones, se determina que el número total de solapamientos parciales es comunmente mucho menor que el máximo teórico $O(N^2)$.

En cuanto a los campos a nivel de transporte, es decir SrcPT/DstPT y protocolo, los UVs 3D definidos son muchos menos que los UVs 2D SrcIP/DstIP, por lo que gran cantidad de filtros IP comparten este reducido grupo de especificaciones de transporte. En el caso de los protocolos, estos en su mayoría son bien TCP o UDP. En el caso de los puertos, en general se especifica sólo un rango o valor de puerto de destino mientras que el puerto de origen es un wilcard; esto es debido a que las reglas se aplican a puertos bien conocidos de servidores que esperan por conexiones entrantes de clientes no conocidos a priori.

Con base en este conjunto de observaciones, los autores proponen definir dos etapas de clasificación claramente diferenciadas, una de red y otra de aplicación, y en base a sus características peculiares aplicar algoritmos y teconologías de clasificación específicas. En el caso de red las especificaciones se limitan a distintos tipos de prefijos, mientras que los solapamientos parciales 2D son muy limitados; por ello se propone realizar una búsqueda donde cada regla almacenada represente un solapamiento parcial (UR) 2D entre las reglas SrcIP/DstIP de clasificación originales. Ya que esta etapa involucra sólo prefijos, puede realizarse mediante software. De acuerdo al resultado de esta etapa, una segunda etapa realiza búsqueda en el sub-grupo correspondiente de transporte, de (k-2) dimensiones, el cual segun lo observado es mucho mas reducido que el de red. Por ello, esta segunda etapa puede implementarse mediante técnicas de búsqueda concurrente en hardware, tales como TCAMs.

Las observaciones de [103] son profundizadas y explotadas en [99], proponiendo una arquitectura para clasificación en dos etapas de red y aplicación respectivamente. En la etapa de red se utiliza el esquema *Most Specific Filter Match (MSFM)*, mientras que en la etapa de aplicación se explota el concepto de *Transport-Level Sharing (TLS)*. El algoritmo MSFM permite agregar resultados de lookup Best Match (BM) sobre prefijos IP, absorbiendo las incertidumbres que surgen según lo estudiado en la Sec. 4.3. MSFM optimiza el consumo de memoria de un esquema tipo cross-producting [96] identificando tres conjuntos de combinaciones 2D, a saber, *filtros completamente cubiertos (Fully Covered, FC), filtros parcialmente cubiertos (Partially Covered, PC), y filtros no-cubiertos (Not Covered, NC)*. Estos filtros 2D se almacenan en tablas de agregación independientes, reduciendo el tamaño de la tabla original. En particular, los filtros parcialmente cubiertos se separan en dos tablas, llamadas *tablas secundarias*, según sea el campo wildcard SrcIP o DstIP, las cuales se direccionan exclusivamente mediante el resultado BM de DstIP o SrcIP respectivamente. Es decir, la búsqueda en cada campo devuelve dos índices; uno para ingrasar a la tabla de productos principal y otro para ingresar a la tabla secundaria correspondiente. En la Fig. 7.16 se muestra un ejemplo de esta separación, donde se aprecian tres filtros originales A-E, así como los grupos derivados de éstos $FC_1 - FC_3,\ PC_1 - PC_3,\ NC_1 - NC_3$. En este caso, una tabla de productos cruzados requiere $5 \cdot 5 = 25$ posiciones de memoria para combinar las posibles combinaciones de resultados 1D; sin embargo como vemos existen distintos casos que se pueden separar. Las combinaciones $NC_1 - NC_3$ se pueden almacenar en una sola posición de memoria ya que todas ellas involucran el filtro D.

Las combinaciones $PC_1 - PC_3$, en tanto, generan un solo índice común para la tabla secundaria de reglas especificadas sólo en DstIP, correspondiente al filtro 2D A. Las combinaciones $FC_1 - FC - 3$, finalmente, son pseudo-reglas que deben ser almacenadas en l atbala principal resultando todas ellas en el filtro 2D B. Según los resultados reportados en [99], y teniendo en cuenta que un alto porcentaje de reglas involucran filtros parcialmente especificados, este esquema logra ahorros de memoria considerables dependiendo del ruleset particular, logrando reducción de 77 % para el caso de ACL3.

El concepto de TLS, en tanto, surge de separar conjuntos de especificaciones de aplicación que comparten una especificación común de red. Para explotar esta propiedad, se re-ordena la tabla de reglas durante su construcción según los grupos definidos. Luego, durante la clasificación, la etapa de MSFM entrega una lista ordenada de índices correspondientes a estos grupos. Las opciones de implementación de TLS son i) comprobación de rangos individuales (técnica referida como comprobación explícita de rangos o ERM en esta Tesis) y ii) TCAMs con expansión de rangos.

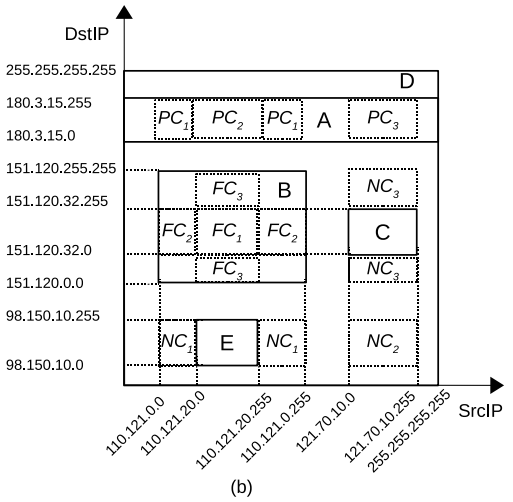
En términos generales, las complejidades de solapamiento de los filtros de red SrcIP/DstIP dependen del tipo de ruleset considerado. En los trabajos anteriores se han utilizado exclusivamente rulesets del tipo ACL. Los filtros de red presentes en rulesets ACL suelen presentar la menor complejidad de solapamiento ya que los patrones se agrupan fuertemente en clusters; mientras los rulesets del tipo IPC y FW presentan complejidades crecientes. En el trabajo [151] se analizan cuidadosamente sus características a fin de lograr agrupación (clustering) eficiente en el espacio de filtros de red. Esencialmente, se busca *reducir* en forma eficiente la cantidad de reglas a comparar durante la clasificación; para ello se buscan bits específicos de los campos en base a los cuales dividir la tabla de reglas. Se analiza en primer término la distribución de los bytes más significativos (MSBs) en los campos SscIP/DstIP y SrcPT/DstPT, concluyendo que la sola utilización de uno de estos campos no es capaz de reducir suficientemente el espacio de búsqueda. Se propone entonces utilizar una combinación del MSB de SrcIP y el MSB de DstIP, formando una dirección de 16 bits con la cual sectorizar el espacio de búsqueda; esto se realiza por direccionamiento directo de memoria. En cada cluster resultante, se deben en una segunda etapa comprobar (a) los $32 - 8 = 24$ bits restantes de SrcIP y DstIP, y (b) los restantes campos, es decir, SrcPT/DstPT (16 bits cada uno) y Protocolo (8 bits). Por supuesto, se espera que de las $2^16 = 64K$ posibles combinaciones de MSB_{DstIP}, MSB_{SrcIP}

Esta es claramente una técnica de sectorización directa, muy similar a la propuestas en [99] y cercana al concepto de *bits eficientes* utilizado en [150]. También podemos relacionarla con el concepto de *sets independientes* desarrollado en [154]; la diferencia es que en este caso la sectorización se realiza en base a los MSBs de SrcIP/DstIP en lugar de las URs utilizadas en aquél para definir grupos independientes donde realizar búsquedas lineales. A diferencia de aquéllos, el principal atractivo de esta técnica reside en su simplicidad, clara presentación y evaluación de múltiples alternativas de implementación en software y hardare, así como su comprobado desempeño con los rulesets considerados. Sin embargo, las estructuras de datos utilizadas pueden variar notablemente su eficacia para rulesets no contemplados.

En base a las consideraciones anteriores, se muestra en la Fig. 7.17 un resumen de los posibles patrones 2D, tanto desde el punto de vista teórico como en casos reales. En particular, nos interesa

Filtro	SrcIP	DstIP
A	*	180.3.15.*
B	110.121.*.*	151.120.*.*
C	121.70.10.*	151.120.32.*
D	*	*
E	168.121.20.*	151.120.30.*

(a)



(b)

Figura 7.16: Separación en grupos FC, PC y NC: (a) tabla de filtros de red, (b) interpretación geométrica

evaluar las cantidades de *Valores Únicos* $|UV|$ y *Regiones Únicas* $|UR|$ generados, así como el máximo solapamiento entre UVs $|UV|_{max}$ para cada caso, ya que ellos definen en gran medida la complejidad de clasificación. En base a estas métricas, podemos considerar como el peor caso aquél que obtiene *máximos* $|UR|$ *y* $|UV|_{max}$ a partir de un *mínimo* $|UV|$ (peor relación ($|UR|$, $|UV|_{max}$) / $|UV|$). A partir de este extremo, se encuentran otros casos desfavorables donde bien $|UR|$ ó $|UV_{max}|$ son máximos. Los casos de las Figs. 7.17(a) y 7.17(b) son implementables con prefijos (en realidad casos especiales de rangos), mientras que el caso de la Fig. 7.17(c) es implementable mediante rangos arbitrarios y fue extraído de los casos evaluados en la Fig. 7.14. En la Fig. 7.17(a) se considera el peor caso posible 2D de prefijos para $|UV_{AB}|_{max}$ y UR_{AB}; se debe observar sin embargo que este caso exige $|UV_A| = |UV_B| = |UV_{AB}|$ ya que no se re-utilizan valores únicos al pasar de 1D a 2D. Además, el caso está limitado para valores donde $n \leq m$; por ejemplo, para prefijos IP donde $M = 32$ podremos obtener este caso siempre que consideremos $n \leq 32$; mas allá de este valor se repetirán UVs y no se cumplen las relaciones citadas. En la Fig. 7.17(b) se muestra un caso intermedio de prefijos, donde se obtiene un caso pesimista considerando que se comparten todos los UVs 1D en los UVs 2D. Las Fig. 7.17(c), en tanto, considera la peor combinación de rangos arbitrarios ya presentada en la Fig. 7.15(a), donde se producen las peores relaciones $(|UR_{AB}|_{max}, |UV_{AB}|_{max})/|UV_{AB}|$ en 2D.

La Fig. 7.17(d) muestra un patrón típico de rulesets reales, donde se observa la existencia de grupos o *clusters* de reglas 2D alejandose notablemente de los peores casos presentados anteriormente. Estos patrones son intensivamente utilizados en [103], [99] para optimizar la clasificación en filtros de red SrcIP/DstIP y reducir el tamaño de la tabla de agregación correspondiente. En la Fig. 7.18 se muestra un caso más detallado; se comprueba aquí que esta agrupación en clusters puede no ser tan efectiva al proyectar en campos 1D debido a la reutilización de UVs 1D, sin embargo vemos que $|UV|$ y $|UR|$ son sensiblemente menores en 1D. De estas observaciones podemos concluir que la sectorización según clusters debería efectuarse en base a patrones 2D, mientras que la estratificación puede ser más efectiva en 1D.

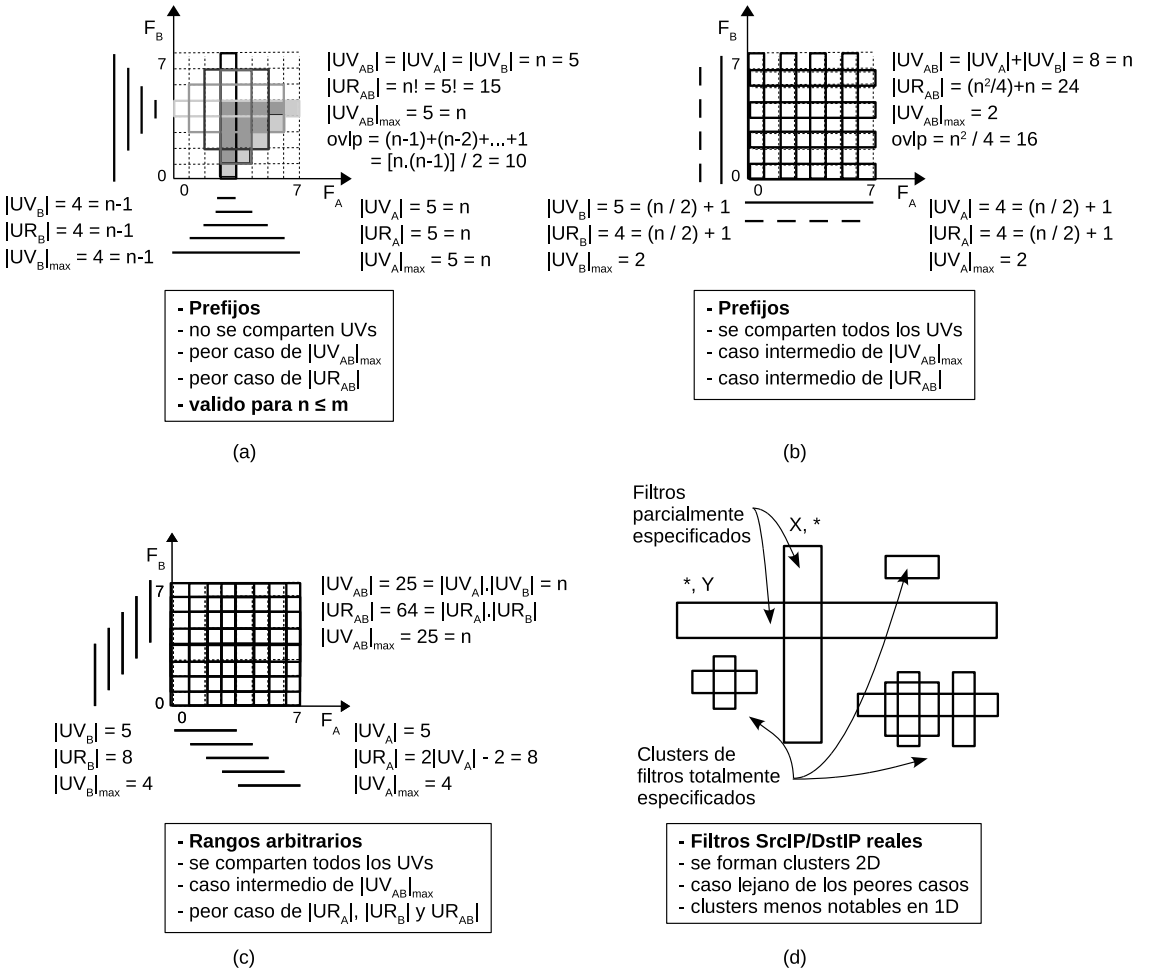


Figura 7.17: Patrones 2D: (a) peor caso en prefijos, (b) caso intermedio en prefijos, (c) peor caso en rangos arbitrarios, (d) caso típico de filtros SrcIP/DstIP reales

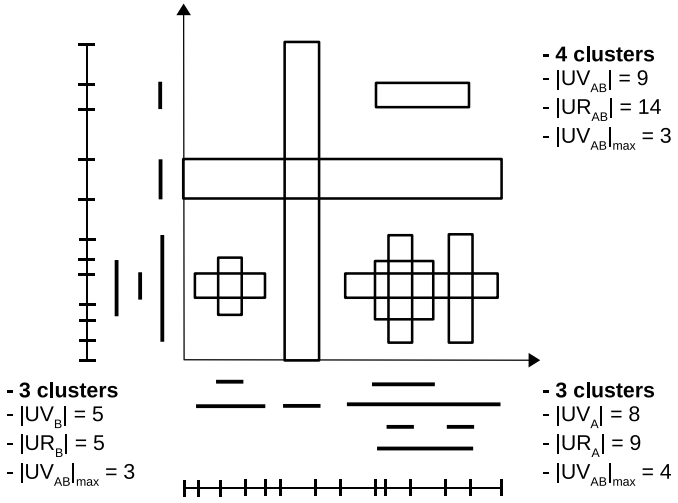


Figura 7.18: Patrón típico de filtros IP en rulesets reales

Otro trabajo reciente [158] propone analizar exhaustivamente los campos involucrados en rulesets reales a fin de atacar el problema de clasificación de paquetes y mejorar las optimizaciones de los algoritmos existentes. Sobre esta base, los autores proponen optimizaciones generales a algoritmos basados en hashing. En particular, se analizan algoritmos basados en *espacios de tuplas*, los cuales dividen el espacio de clasificación según los bits especificados en cada campo. En caso de prefijos como los campos SrcIP/DstIP, estos dependen de las longitudes de prefijos; para valores exactos como protocolo es sólo un booleano (valor especificado o no), mientras que para el caso de rangos arbitrarios se define como una de las capas sin solapamiento de la Fig. 7.4(c) (llamadas *nesting levels* en ese trabajo).

Los encabezados analizados consisten de un máximo de 8 campos conteniendo direcciones SrcIP/DstIP (32 bits cada una), puertos de origen y destino (16 bit cada uno), Type-of-Service (TOS, 8 bits), protocolo (8 bits) y flags de protocolo de transporte (8 bits), en un total del 120 bits. Los protocolos especificados son un limitado conjunto de valores TCP, UDP, ICMP, IGMP, (E)IGRP, GRE, IPINIP y wildcard (*). Los puertos de transporte incluyen variadas especificaciones de rango, tales como "gt 1023" o "20 – 24". Se observó que el tamaño actual de rulesets es menor a mil reglas; sin embargo esta cantidad puede aumentar notablemente a medida que los servicios y sus clasificacdores asociados se desplazan hacia el nucleo de la red. Se analizaron los 12 rulesets ACL, FW y IPC utilizados en [95] y [134], con tamaños variables entre 68 y 4557 reglas. El formato de cada regla en los rulesets definidos es *@[Source IP address in dot-decimal notation]/[Prefix length] [Destination IP address in dot-decimal notation]/[Prefix length] [Low source port]:[High source port] [Low destination port]:[High destination port] [Protocol value in hexadecimal/[Protocol mask in hexadecimal]]*, por ejemplo *@175.77.88.155/32 119.106.158.230/32 0:65535 6888:6888 0x06/0xFF*. Los rulesets ACL1, FW1 e IPC1 se obtienen de rulesets reales, mientras que los demás rulesets se generan mediante Classbench. Los autores realizaron análisis complementarios a los de [103], estudiando estadísticamente las especificaciones de cada campo en los orulesets considerados. En general, se observó que (a) los bits válidos en SrcIP/DstIP se encuentran mayormente entre los bits 0-4 del primer octeto y los bits 16-32 del trecer y cuarto octetos; (b) los puertos se especifican más para el origen que para el destino, (c) los scopes de puerto son en general grandes, y (d) los puertos de destino contienen mas valores exactos que sus contrapartes de origen.

El análisis de longitudes de prefijo demuestra que la mayor parte de las especificaciones SrcIP/DstIP se concentran en grandes longitudes (máxima longitud=valor exacto, un host) y en menor medida en longitudes nulas (wildcard). En cuanto a los puertos de origen y destino, se observó una distribución más equitativa entre valores rpesentes en el caso de los puertos de origen, ya que generalmente las reglas especifican rangos de puertos de origen y un valor particular de puerto de destino; por lo demás la distribucion de los valores es bastante equitativa en el total de reglas. En cuanto a los intervalos (scopes) cubiertos por las especificaciones de puertos, se observa que los puertos de origen son por lo general rangos muy amplios o wildcards, mientras que los puertos de destino son rangos bien muy amplios (wilcards en el extremo) o muy acotados (valores exactos en el extremo). Los autores argumentan que, teniendo en cuenta estas observaciones, se pueden optimizar los algoritmos basados en división por tuplas; sin embargo no se aportan propuestas concretas de optimización.

7.3.3. Resultados obtenidos

En los trabajos mencionados se nota que los formatos ACL, FW y IPC difieren bastante entre sí en cuanto a patrones de reglas. En [116], por ejemplo, la evaluación se efectúa exclusivamente mediante rulesets ACL, mientras que en [156], [150], [61] y [151] se enfatizan las mayores complejidades de IPC y más notablemente FW. Motivados en particular por las observaciones de estos trabajos, y con el objeto de complementar aquéllas para su mejor análisis, se realizaron nuevos estudios sobre la batería de rulesets disponibles. Los rulesets utilizados son los mencionados en [95] y [134], donde se detallan sus características. En primer término, interesa observar más claramente las propiedades mencionadas de las duplas IP. Para ello, en las Figs. 7.19(a), 7.19(b), y 7.19(c) se observan las distribuciones 2D obtenidas de las duplas $\{MSB_{ScrIP}, MSB_{DstIP}\}$ presentes en filtros *ACL1, IPC1 y FW1* de 10K reglas respectivamente. Las ocurrencias obtenidas son el resultado de sumar las contribuciones de todas las reglas en el campo considerado. Cabe destacar que la mínima sensibilidad en estos graficos es 1 MSByte de los cuatro bytes que contiene IPv4, por lo que cada unidad de los ejes significa un incremento de 2^{24} posiciones en el campo correspondiente. Por otro lado, ya que no es posible encontrar solapamientos entre prefijos de igual longitud, podemos afirmar que dos prefijos con MSBs distintos no se solaparán. La especificaciones *wildcard* se expresan mediante el prefijo 0.0.0.0/0, por lo que se sitúan en el cero de alguno de los dos ejes. En términos generales, observamos en la Fig. 7.19(a) predominancia de reglas que involucran longitudes de prefijo < 8 en el origen y < 8 en el destino, lo que significa reglas que aplican a desde un grupo grande de hosts hacia un grupo chico de hosts (incluyendo un host exacto, que aquí no diferenciamos). Estos grupos están bastante bien definidos en ambos ejes, lo que corrobora la baja complejidad de los rulesets ACL. En la Fig. 7.19(b), en tanto, vemos que el ruleset IPC involucra exclusivamente prefijos de longitud > 8 en ambos ejes, que se distribuyen en dos grandes sectores del plano IP. Ya que el numero de reglas es similar al de ACL, es de esperar que muchas reglas compartan el mismo MSByte, produciéndose mayores solapamientos que en el caso ACL. El ruleset FW, ilustrado en la Fig. 7.19(c), posee un patrón aún más complejo, con fuerte presencia de reglas involucrando wildcards en alguno de los ejes, las cuales generan gran cantidad de solapamientos.un porcentaje menor se ubica a lo largo de la diagonal $MSB_{SrcIP} = MSB_{DstIP}$, y en líneas similares a las de ACL pero de muy baja ocurrencia, indicando menor solapamiento ya que se distribuyen entre diferentes duplas MSB_{ScrIP}, MSB_{DstIP}.

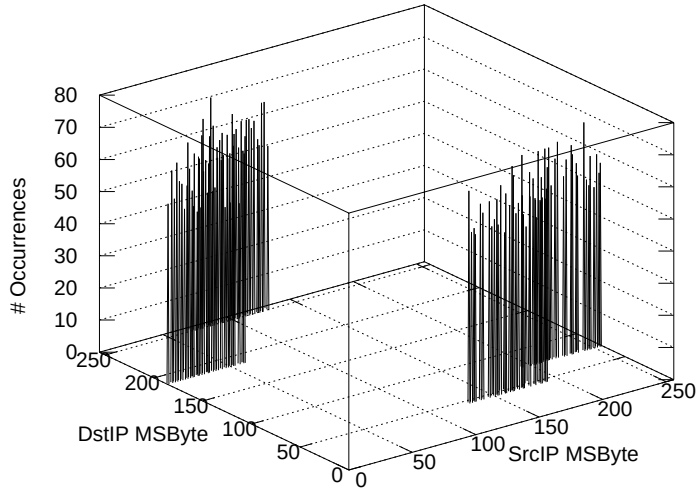
En segundo lugar, interesa observar las magnitudes de $|UV|$, $|UR|$ y $|UV|_{max}$ para los rulesets considerados. En el Cuadro 7.1 se muestran valores del parámetro $|UV|$ para los campos individuales y sus resultados de agregación 2D, tanto para distintos rulesets (I) como para rulesets del mismo tipo a distinta escala (II). En el Cuadro 7.3, en tanto, se muestran los valores $|UR|$ de campos y sus etapas de agregación en los distintos rulesets, para el esquema de agregación más conveniente según los estudios de [106] y [134]. En este cuadro, $T1, T2$ y $T3$ de la etapa 1 corresponden a los resultados de agregar *SrcIP[31:16]/SrcIP[15:0]*, *DstIP[31:16]/DstIP[15:0]* y *Prot[7:0]/SrcPT[15:0]/DstPT[15:0]* respectivamente. *T1* de la etapa 2, en tanto, es la agregación de *SrcIP[31:0]/DstIP[31:0]*; mientras que *T1* de la etapa 3 es el resultado final de agregación, es decir, $|UR|_{1...k}$. El Cuadro 7.4, en tanto, muestra las cantidades de valores únicos y la cantidad de capas (layers) resultantes de aplicar estratificación en cada campo para rulesets de distintos tamaños. Estos valores se obtuvieron mediante algoritmos de estratificación 1D, luego de realizar ordenamiento en base a puntos finales.

Cuadro 7.1: UVs en rulesets reales (I)

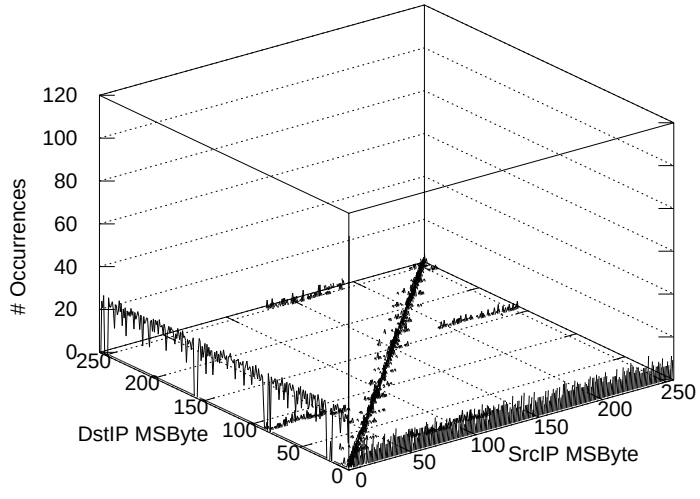
Set	Rules	Network (IP)			Application (Ports)		
		srcIP	dstIP	2D UVs	srcPRT	dstPRT	2D UVs
ACL1	752	96	205	**425**	1	140	**140**
ACL2	557	124	293	**462**	1	25	**25**
ACL3	2302	377	551	**1426**	3	164	**167**
ACL4	2781	228	838	**1634**	3	203	**205**
ACL5	3289	289	942	**1664**	1	39	**39**
FW1	269	56	66	**128**	13	43	**55**
FW2	62	36	15	**44**	9	1	**9**
FW3	140	28	15	**41**	8	34	**42**
FW4	224	7	43	**63**	19	42	**108**
FW5	127	18	17	**34**	9	22	**35**
IPC1	1550	152	128	**941**	34	54	**85**
IPC2	557	20	46	**73**	3	3	**9**

Cuadro 7.2: UVs en rulesets reales (II)

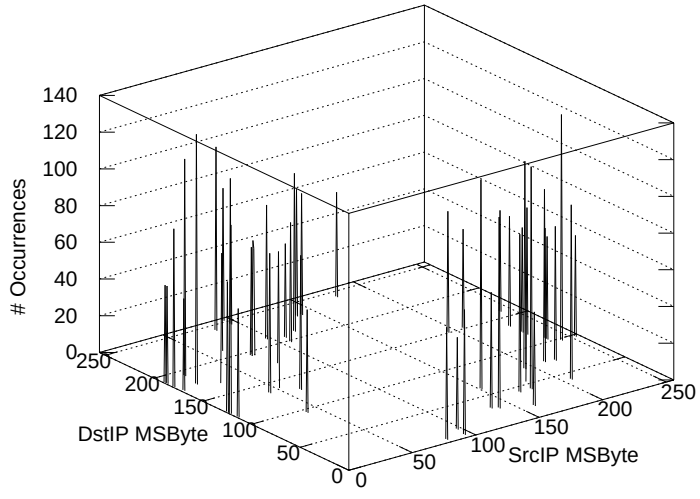
Set	Rules	Network (IP)			Application (Ports)		
		srcIP	dstIP	2D UVs	srcPRT	dstPRT	2D UVs
ACL1	752	96	205	**425**	1	140	**140**
ACL1_1K	916	103	297	**493**	1	99	**99**
ACL1_5K	4415	805	640	**2118**	1	108	**108**
ACL1_10K	9603	4784	733	**6826**	1	108	**108**
FW1	269	56	66	**128**	13	43	**55**
FW1_1K	791	123	175	**317**	13	42	**95**
FW1_5K	4653	1745	2714	**3970**	13	43	**162**
FW1_10K	9311	3637	6952	**9237**	13	43	**196**
IPC1	1550	152	128	**941**	34	54	**85**
IPC1_1K	938	336	436	**843**	28	44	**76**
IPC1_5K	4460	584	1251	**3269**	34	53	**101**
IPC1_10K	9037	1515	2726	**6964**	34	54	**131**



(a)



(b)



(c)

Figura 7.19: Distribución del MSByte en filtros IP: (a) ruleset ACL1 10K, (b) ruleset FW1 10K, (c) ruleset IPC1 10K

Cuadro 7.3: Valores de $|UR|$ en distintas etapas de agregación

Set	#Reglas	Etapa 0							Etapa 1			Etapa 2	Etapa 3
		SrcIP [15:0]	SrcIP [31:16]	DstIP [15:0]	DstIP [31:16]	Prot [7:0]	SrcPT [15:0]	DstPT [15:0]	T1	T2	T3	T1	T1
ACL1	752	96	5	177	78	5	2	141	98	206	146	430	758
ACL2	557	93	49	225	108	6	2	26	125	294	45	1455	12126
ACL3	2302	293	80	541	43	6	4	166	378	552	432	9291	13929
ACL4	2781	186	58	807	119	8	4	204	229	839	529	7036	20026
ACL5	3289	214	44	941	120	5	2	40	290	943	70	1735	3339
FW1	269	55	14	67	9	6	14	44	58	67	153	2335	5348
FW2	62	30	13	16	9	6	10	2	38	16	15	159	272
FW3	140	26	15	16	9	5	9	35	30	16	177	238	1383
FW4	224	9	4	43	6	8	20	43	9	44	168	310	1644
FW5	148	38	17	39	12	5	11	29	45	41	158	1082	1675
IPC1	1579	282	40	549	97	8	32	51	329	641	2328	6162	NA
IPC2	144	21	5	47	13	5	4	4	21	47	11	385	514

Cuadro 7.4: Resultados de estratificación en los rulesets analizados

Ruleset	Layer	SrcIP	DstIP	ScrPRT	DstPRT
ACL1_1K	L1	82	269	1	88
	L2	18	23	0	5
	L3	2	4	0	4
	L4	1	1	0	1
	L5	0	0	0	1
ACL1_5K	L1	699	606	1	97
	L2	89	21	0	5
	L3	16	6	0	4
	L4	1	4	0	1
	L5	0	0	0	1
ACL1_10K	L1	4276	692	1	97
	L2	381	26	0	5
	L3	126	10	0	4
	L4	1	2	0	1
	L5	0	0	0	1
FW1_1K	L1	96	155	11	40
	L2	18	13	1	1
	L3	9	6	1	1
	L4	1	1	0	0
	L5	0	0	0	0
FW1_5K	L1	1454	2542	11	41
	L2	259	149	1	1
	L3	32	22	1	1
	L4	1	1	0	0
	L5	0	0	0	0
FW1_10K	L1	3611	6950	11	41
	L2	26	1	1	1
	L3	1	0	1	1
	L4	0	1	0	0
	L5	0	0	0	0
IPC1_1K	L1	244	268	24	38
	L2	61	92	2	2
	L3	30	50	1	2
	L4	1	25	1	1
	L5	0	0	0	0
IPC1_5K	L1	466	748	30	46
	L2	73	239	2	4
	L3	45	175	1	2
	L4	1	87	1	1
	L5	0	0	0	0
IPC1_10K	L1	1024	1684	30	46
	L2	191	502	2	4
	L3	119	359	1	2
	L4	1	177	1	1
	L5	0	0	0	1

Bibliografía

[1] C. Hermsmeyer, H. Song, R. Schlenk, R. Gemelli, and S. Bunse, "Towards 100G packet processing: Challenges and technologies, " Bell Labs Technical Journal, vol. 14, no. 2, 2009.

[2] Approaching the Zettabyte Era, Cisco Systems White Paper, 2008.

[3] Leigh, K., Ranganathan, P., Sughlok, J., "Fabric Convergence Implications on Systems Architecture, " IEEE 14th International Symposium on High Performance Computer Architecture HPCA 2008, pp.15-26. IEEE Press, New York, 2008.

[4] T. Pan, X. Guo, C. Zhang, W. Meng, and B. Liu, "ALFE: A replacement policy to cache elephant flows in the presence of mice flooding, " 2012 IEEE International Conference on Communications (ICC), pp. 2961-2965, 2012, doi: 10.1109/ICC.2012.6364403.

[5] Virtual Networking Technologies at the Server-Network Edge, HP white paper, 2011.

[6] S. GadelRab, "10-Gigabit Ethernet Connectivity for Computer Servers, " IEEE Micro, vol. 27, no. 3, pp. 94-105, 2007.

[7] M. Petracca, R. Birke, and A. Bianco, "HERO: High-Speed Enhanced Routing Operation in Ethernet NICs for Software Routers, " Computer Networks, no. 53, vol. 2, pp. 168-179, 2009.

[8] G. Xie, P. He, H. Guan, Z. Li, G. Xie, L. Luo, J. Zhang, Y. Wang, K. Salamatian, "PEARL: A Programmable Virtual Router Platform, " IEEE Communications Magazine, vol. 49, no. 7, pp. 71-77, 2011.

[9] S. Haria, T. Ganegedara, and V. K. Prasanna, "Power-Efficient and Scalable Virtual Router Architecture on FPGA, " Proc. IEEE ReConFig Conference, pp. 1-7, 2012.

[10] N.M.M.K. Chowdhury, and R. Boutaba, "Network Virtualization: State of the Art and Research Challenges, " IEEE Communications Magazine, vol. 47, no. 7, pp. 20-26, 2009.

[11] R. Jain and S. Paul, "Network Virtualization and Software Defined Networking for Cloud Computing: A Survey, " IEEE Communications Magazine, vol. 51, no. 11, pp. 24-31, 2013.

[12] B.A.A. Nunes, M. Mendonca, X. Nguyen, K. Obraczka, and T. Turletti, "A Survey of Software-Defined Networking: Past, Present, and Future of Programmable Networks, " IEEE Communications Surveys & Tutorials, vol.16, no.3, pp.1617-1634, Third Quarter 2014.

[13] N. McKeown, T. Anderson, H. Balakrishnan, G. Parulkar, L. Peterson, J. Rexford, S. Shenker, and J. Turner, "OpenFlow: Enabling Innovation in Campus Networks, " SIGCOMM Comput. Commun. Rev., vol. 38, no. 2, pp. 69-74, 2008.

[14] Software-Defined Networking: The New Norm for Networks, ONF White Paper, April 13, 2012.

[15] E. Kohler, R. Morris, B. Chen, J. Jannotti, and M. F. Kaashoek, "The Click! Modular Router, " ACM Transactions on Computer Systems vol. 18, no. 3, pp. 263-297, August 2000.

[16] Z. Bozakov, and P. Papadimitriou, "OpenVRoute: An open architecture for high-performance programmable virtual routers, " 2013 IEEE 14th International Conference on High Performance Switching and Routing (HPSR), pp. 191-196, 2013, doi: 10.1109/HPSR.2013.6602311.

[17] J. W. Lockwood, "NetFPGA-An Open Platform for Gigabit-Rate Network Switching and Routing, " IEEE International Conference on Microelectronic Systems Education MSE '07, pp. 160-161, 2007.

[18] J. Naous, D. Erickson, G. Adam Covington, G. Appenzeller, and N. McKeown, "Implementing an OpenFlow Switch on the NetFPGA Platform, " Proc. 4th ACM/IEEE Symposium on Architectures for Networking and Communications Systems (ANCS '08), pp. 1-9, 2008.

[19] D. Unnikrishnan, R. Vadlamani, Y. Liao, J. Crenne, L. Gao, and R. Tessier, "Reconfigurable Data Planes for Scalable Network Virtualization, " IEEE Transactions on Computers, vol. 62, no. 12, pp. 2476-2488, Dec. 2013.

[20] S. H. Min, Y. C. Choi, N. Kim, W. Kim, O. C. Kwon, B. C. Kim, J. Y. Lee, D. Y. Kim, J. Kim, and H. Song, "Implementation of a Programmable Service Composition Network using NetFPGA-based OpenFlow Switches, " Proc. 1st Asia NetFPGA Developers Workshop, 2010.

[21] M. Berman, J. S. Chase, L. Landweber, A. Nakao, M. Ott, D. Raychaudhuri, R. Ricci, and I. Seskar, "GENI: A Federated Testbed for Innovative Network Experiments, " Computer Networks, vol. 61, pp. 5-23, 2014.

[22] M. Forconesi, G. Sutter, S. Lopez-Buedo, and J. Aracil, "Accurate and Flexible Flow-based Monitoring for High-Speed Networks, " 23rd International Conference on Field Programmable Logic and Applications (FPL 2013), pp. 1-4, 2013.

[23] N. Vaish, T. Kooburat, L. De Carli, K. Sankaralingam, and C. Estan, "Experiences in Co-designing a Packet Classification Algorithm and a Flexible Hardware Platform, " 7° ACM/IEEE Symposium on Architectures for Networking and Communications Systems (ANCS), pp. 189-199, 2011.

[24] G. Guichal, "Chaltén: placa con micro ARM9 y FPGA, " Simposio Argentino de Sistemas Embebidos SASE 2011, 2011.

[25] S. E. Tropea, D. J. Brengi, and J. P. D. Borgna, "FPGALibre: Herramientas de Software Libre para Diseño con FPGAs, " Proc. I Southern Conference on Programmable Logic (SPL), 2006.

[26] A. M. Quinteros, L. A. Guanuco, S. D. Olmedo, "Plataforma de Hardware Reconfigurable para el Diseño de Sistemas Digitales, " Congreso de Microelectrónica Aplicada (uEA), 2014.

[27] M. Handley, O. Hodson, and E. Kohler, "XORP: an open platform for network research, " SIGCOMM Comput. Commun. Rev. vol. 33, no. 1, pp. 53-57, Jan. 2003.

[28] A. Bianco, J. M. Finochietto, G. Galante, M. Mellia, and F. Neri, "Open-Source PC-Based software routers: a viable approach to high-performance packet switching, " In Proc. Third international Conference on Quality of Service in Multiservice IP Networks (QoS-IP'05), pp. 353-366, 2005.

[29] Q. Ye; and M.H. MacGregor, "Hardware bottleneck evaluation and analysis of a software PC-based router, " International Symposium on Performance Evaluation of Computer and Telecommunication Systems (SPECTS 2008), pp.480-487, 2008.

[30] J. C. Mogul and K. K. Ramakrishnan, "Eliminating receive livelock in an interrupt-driven kernel, " ACM Trans. Comput. Syst. vol. 15, no.3, pp. 217-252, 1997.

[31] M. Dobrescu, N. Egi, K. Argyraki, B. Chun, K. Fall, G. Iannaccone, A. Knies, M. Manesh, and S. Ratnasamy, "RouteBricks: exploiting parallelism to scale software routers, " Proc. ACM SIGOPS 22nd symposium on Operating systems principles (SOSP '09), pp. 15-28, 2009.

[32] B. Pfaff, J. Pettit, T. Koponen, K. Amidon, M. Casado, and S. Shenker, "Extending networking into the virtualization layer, " 8th ACM Workshop on Hot Topics inNetworks (HotNets-VIII), 2009.

[33] Z. Bozakov, and P. Papadimitriou, "OpenVRoute: An open architecture for high-performance programmable virtual routers, " IEEE 14th International Conference on High Performance Switching and Routing (HPSR 2013), pp.191-196, 2013.

[34] R. Giladi, "Network Processors: Architecture, Programming, and Implementation, " Morgan Kaufmann, Burlington (2008).

[35] G. Stark, S. Sezer, "NFP-6xxx – A 22nm High-Performance Network Flow Processor for 200Gb/s Software Defined Networking, " 25th Symposium on High Performance Chips (HotChips), 2013.

[36] K. Pocek, R. Tessier, and A. DeHon, "Birth and Adolescence of Reconfigurable Computing: A Survey of the First 20 Years of Field-Programmable Custom Computing Machines, " Highlights of the First Twenty Years of the IEEE International Symposium on Field-Programmable Custom Computing Machines, Apr. 2013.

[37] D. Bacon, R. Rabbah, and S. Shukla, "FPGA Programming for the Masses," ACM Queue, vol. 11, no. 2, pp. 40-53, Feb. 2013.

[38] J. Lockwood, N. Naufel, J. Turner, and D. Taylor, "Reprogrammable Network Packet Processing on the Field Programmable Port Extender (FPX), " Proc. ACM International Symposium on Field Programmable Gate Arrays (FPGA'2001), pp. 87-93, 2001.

[39] D. Taylor, J. Turner, J. Lockwood, E. Horta, "Dynamic Hardware Plugins: Exploiting Reconfigurable Hardware for High-Performance Programmable Routers, " Computer Networks, vol. 38, pp. 295-310, 2002.

[40] G. Brebner, "Single-chip gigabit mixed-version IP router on Virtex-II Pro, " Proc. 10th Annual IEEE Symposium on Field-Programmable Custom Computing Machines, pp. 35- 44, 2002.

[41] J. Shafer and S. Rixner, "RiceNIC: a reconfigurable network interface for experimental research and education, " Proc. Workshop on Experimental Computer Science, Part of ACM FCRC, 2007.

[42] Dual-Port 10GbE Enterprise Server Adapter Datasheet, Solar Flare Corp., http://www.solarflare.com/content/userfiles/documents/solarflare_sfn5122f_10gbe_adapter_brief.pdf.

[43] Intel 82599 10 GbE Controller Datasheet, Intel Corp., http://www.intel.la/content/www/xl/es/ethernet-controllers/82599-10-gbe-controller-datasheet.html.

[44] G. Gibb, J. Lockwood, J. Naous, P. Hartke, and N. McKeown, "NetFPGA - An Open Platform for Teaching How to Build Gigabit-rate Network Switches and Routers, " IEEE Transactions on Education, vol. 51, no. 3, pp. 364-369, Aug. 2008.

[45] DE4 NetFPGA website, http://keb302.ecs.umass.edu/de4web/DE4_NetFPGA/.

[46] O. Hagsand, R. Olsson, and B. Gorden, "Towards 10Gb/s open source routing, " Proc. of the Linux Symposium, 2008.

[47] V. Tanyingyong, M. Hidell, and P. Sjodin, "Improving performance in a combined router/server, " Proc. IEEE 13th International Conference on High Performance Switching and Routing (HPSR), pp.52-58, 2012.

[48] V. Tanyingyong, M. Hidell, and P. Sjodin, "Using hardware classification to improve PC-based OpenFlow switching, " Proc. IEEE 12th International Conference on High Performance Switching and Routing (HPSR), pp.215-221, 2011.

[49] The Multilayer Open Virtual Switch, http://www.openvswitch.org.

[50] M. Casado, T. Koponen, D. Moon, and S. Schenker, "Rethinking Packet Forwarding Hardware, " ACM Special Interest Group on Data Communications Workshop on Hot Topics in Networks (HotNets- VII), pp. 1-6, 2008.

[51] M. Bilal Anwer, M. Motiwala, M. bin Tariq, N. Feamster, "SwitchBlade: A Platform for Rapid Deployment of Network Protocols on Programmable Hardware, " ACM Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications SIGCOMM 2010, pp. 183-194, 2010.

[52] C. Zerbini and J.M. Finochietto, "Reconfigurable Network Processing: The FPGA case, " Argentine Symposium on Technology, 40° Jornadas Argentinas de Informática (JAIIO), pp. 260-271, 2011.

[53] J.M. Finochietto, S. Paz, and C. Zerbini, "Hardware primitives for packet flow processing architectures, " VII Southern Conference on Programmable Logic (SPL), pp.37-43 2011.

[54] M. Arpaci, and J.A. Copeland, "Buffer management for shared-memory ATM switches, " IEEE Communications Surveys & Tutorials, vol. 3, no. 1, pp. 2-10, First Quarter 2000.

[55] J. Chao, B. Liu: High Performance Switches and Routers. Wiley & Sons, New Jersey (2007).

[56] Integrating 100-GbE Switching Solutions on 28-nm FPGAs, Altera White Paper, 2010.

[57] L. Mhamdi, K. Goossens, and IV Senin, "Buffered Crossbar Fabrics Based on Networks on Chip, " Eighth Annual Communication Networks and Services Research Conference (CNSR), pp.74-79, 2010.

[58] T. Karadeniz, L. Mhamdi, K. Goossens, and J.J. Garcia-Luna-Aceves, "Hardware design and implementation of a Network-on-Chip based load balancing switch fabric, " Proc. ReConfig 2012, pp. 1-7, 2012.

[59] M. Weber, "Arbiters: design ideas and coding styles, " Proc. Synopsys Users Group (SNUG) 2001, 2001.

[60] E. Shin, V. J. Mooney III, and G.F. Riley, "Round-Robin Arbiter Design and Generation, " 15th International Symposium on System Synthesis (ISSS '02), pp. 243-248, 2002.

[61] B. Yang, J. Fong, X. Wang, Y. Qi, J Li, W. Jiang, "ParaSplit: A Scalable Architecture on FPGA for Terabit Packet Classification, " Proc. IEEE 20th Annual Symposium on High-Performance Interconnects (HOTI), pp. 1-8, 2012.

[62] P. Gupta and N. McKeown, "Algorithms for packet classification, " Netwrk. Mag. of Global Internetwkg, vol. 15, no. 2, pp. 24-32, March 2001.

[63] SNORT network intrusion detection system, www.snort.org.

[64] F. Yu, R.H. Katz, R.H.; and T.V. Lakshman, .[Ef]ficient multimatch packet classification and lookup with TCAM, " IEEE Micro, vol. 25, no. 1, pp. 50-59, Jan.-Feb. 2005.

[65] F. Yu; T.V. Lakshman, M.A. Motoyama, and R.H. Katz, "Efficient Multimatch Packet Classification for Network Security Applications, " IEEE Journal on Selected Areas in Communications, vol. 24, no. 10, pp. 1805-1816, Oct. 2006.

[66] M. P. Fernandez, "Comparing OpenFlow Controller Paradigms Scalability: Reactive and Proactive, " Proc. 27th IEEE International Conference on Advanced Information Networking and Applications (AINA), pp.1009-1016, March 2013.

[67] F. Yu, R.H. Katz, and T.V. Lakshman, "Efficient multimatch packet classification and lookup with TCAM," IEEE Micro, vol. 25, no. 1, pp. 50-59, Jan.-Feb. 2005.

[68] M. Faezipour and M. Nourani, "Wire-Speed TCAM-Based Architectures for Multimatch Packet Classification, " IEEE Trans. Comput. vol. 58, no. 1, pp. 5-17, Jan. 2009.

[69] A. Rasmussen, A. Kragelund, M. Berger, H. Wessing, and S. Ruepp, "TCAM-based high speed Longest prefix matching with fast incremental table updates, " proc. IEEE 14th International Conference on High Performance Switching and Routing (HPSR), pp. 43-48, 2013.

[70] B. Agrawal and T. Sherwood, "Ternary cam power and delay model: Extensions and uses, " IEEE Transactions on Very Large Scale Integration (VLSI) Systems, vol. 16, no.5, pp. 554 –564, may 2008.

[71] K. Pagiamtzis and A. Sheikholeslami, "Content-addressable memory (CAM) circuits and architectures: A tutorial and survey, "IEEE Journal of Solid-state Circuits, vol. 41, no. 3, pp. 712-727, 2006.

[72] E. Spitznagel, D. Taylor, and J. Turner, "Packet classification using extended TCAMs, " Proc. 11th IEEE International Conference on Network Protocols, pp. 120-131, 2003.

[73] A. Bremler-Barr, and D. Hendler, "Space-Efficient TCAM-Based Classification Using Gray Coding, " Proc. 26th IEEE International Conference on Computer Communications INFOCOM 2007, pp.1388-1396, 2007.

[74] C. R. Meiners, A. Liu, and E. Torng, "Related Work,", Hardware Based Packet Classification for High Speed Internet Routers, New York, Springer Science+Business Media, 2010, ch. 3, pp. 15-24.

[75] C. R. Meiners, A. X. Liu, E. Torng, and J. Patel, "Split: optimizing space, power, and throughput for TCAM-based classification," Proc. 7th ACM/IEEE Symposium on Architectures for Networking and Communications Systems (ANCS), 2011, pp. 200-210.

[76] O. Rottenstreich, R. Cohen, D. Raz, and I. Keslassy, "Exact Worst Case TCAM Rule Expansion, " IEEE Transactions on Computers, vol. 62, no. 6, pp. 1127-1140, June 2013.

[77] F. Heile, A. Leaver, K. Veenstra, "Programmable Memory Blocks Supporting Content-Addressable Memory," in Proc. Eighth International ACM Symposium on Field-Programmable Gate Arrays (FPGA'00), pp. 13-21, 2000.

[78] Altera Corporation. (2001, July) Implementing high-speed search applications with Altera TCAM [Online].

[79] J. Brelet. (1999, September 23) An overview of multiple TCAM designs in Virtex family devices [Online].

[80] J. Ditmar, K. Torkelsson, and A. Jantsch, "A Dynamically Reconfigurable FPGA-Based Content Addressable Memory for Internet Protocol Characterization, " Proc. 10th International Conference on Field Programmable Logic and Applications FPL2000, pp. 19-28, 2000.

[81] T. K. Lee, S. Yosuf, W. Sloman, E. L, and N. Dulay, "Irregular reconfigurable TCAM structures for firewall applications, " Proc. 13th International Conference on Field Programmable Logic and Applications FPL 2003, pp. 890-899, 2003.

[82] J. Naous, D. Erickson, A. Covington, G. Appenzeller, and N. McKeown, "Implementing an OpenFlow Switch on the NetFPGA platform, " Proc. ACM/IEEE Symposium on Architectures for Networking and Communications Systems ANCS'08, pp. 1-9, 2008.

[83] G. S. Jedhe, A. Ramamoorthy, and K. Varghese, "A scalable high throughput firewall in FPGA, " Proc. 16th International Symposium on Field-Programmable Custom Computing Machines FCCM '08 , pp. 43-52, 2008.

[84] Advanced Synthesis Cookbook, Altera Corporation, pp. 61-64, 2011.

[85] J. Brelet and L. Gopalakrishnan. (2002, February 27) Using Virtex-II Block RAM for High Performance Read/Write TCAM [Online].

[86] J. Vuillamy. (2002, May 26) Implementing TCAMs in Stratix [Online].

[87] G. Varghese, *Network Algorithmics,* Morgan Kaufmann, 2005.

[88] M. A. Ruiz-Sanchez, E. W. Biersack, and W. Dabbous, "Survey and taxonomy of IP address lookup algorithms, " IEEE Network, vol. 15, no. 2, pp. 8-23, Mar/Apr 2001.

[89] A.V. Levitin, *Introduction to the Design and Analysis of Algorithms,* Addison Wesley, 2002.

[90] P. Gupta, and N. McKeown, "Classifying packets with hierarchical intelligent cuttings, " IEEE Micro, vol. 20, no. 1, pp. 34-41, Jan/Feb 2000.

[91] S. Singh, F. Baboescu, G. Varghese, and J. Wang, "Packet Classification Using Multidimensional Cutting, " Proc. ACM SIGCOMM, pp. 213-224, 2003.

[92] Y. Qi, L. Xu, B. Yang, Y. Xue, J. Li, "Packet Classification Algorithms: From Theory to Practice, " IEEE INFOCOM 2009, pp. 648-656, 2009.

[93] B. Vamanan, G. Voskuilen, and T. N. Vijaykumar, "EffiCuts: optimizing packet classification for memory and throughput, " SIGCOMM Comput. Commun. Rev., vol. 40, no. 4, pp. 207-218, Aug. 2010.

[94] W. Jiang and V.K. Prasanna, "Scalable Packet Classification on FPGA, " IEEE Trans. Very Large Scale Integration (VLSI) Systems, vol.20, no.9, pp. 1668-1680, Sept. 2012, doi:10.1109/TVLSI.2011.2162112.

[95] D. E. Taylor and J. S. Turner, "ClassBench: a packet classification benchmark, " IEEE/ACM Trans. Netw., vol. 15, no. 3, pp. 499-511, Jun. 2007.

[96] V. Srinivasan, G. Varghese, S. Suri, and M. Waldvogel, "Fast and scalable layer four switching, " Proc. ACM Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication SIGCOMM '98, pp. 191-202, 1998, doi:10.1145/285237.285282.

[97] D. E. Taylor and J. S. Turner, "Scalable packet classification using distributed crossproducting of field labels, " Proc. IEEE 24th Annual Joint Conference of the IEEE Computer and Communications Societies INFOCOM 2005, vol. 1, pp. 269-280, 2005, doi:10.1109/INFCOM.2005.1497898.

[98] W. Jiang and V.K. Prasanna, "Scalable Packet Classification: Cutting or Merging?, " Proc. 18th International Conference on Computer Communications and Networks ICCCN 2009, vol., no., pp.1-6, Aug. 2009.

[99] M. E. Kounavis, A. Kumar, R. Yavatkar, and H. Vin, "Two Stage Packet Classification Using Most Specific Filter Matching and Transport Level Sharing, " Comput. Netw., vol. 51, no. 18, pp. 4951-4978, Dec. 2007.

[100] D. E. Taylor, "Survey and taxonomy of packet classification techniques" ACM Comput. Surv., vol. 37, no. 3, pp. 238-275, Sept. 2005.

[101] S.O. Al-Mamory, W.S. Bhaya and A.M. Hadi, "Taxonomy of Packet Classification Algorithms, " Journal of Babylon University, vol. 21, no. 7, pp. 2296-2307, 2013.

[102] M. H. Overmars and A. F. van der Stappen, "Range searching and point location among fat objects, " J. Algorithms, vol. 21, no. 3, pp. 629-656, Nov. 1996.

[103] M. E. Kounavis, A. Kumar, H. Vin, R. Yavatkar, and A. T. Campbell, "Directions in Packet Classification for Network Processors, " Proc. of Network Processor Workshop in conjunction with Ninth International Symposium on High Performance Computer Architecture (HPCA-9), pp. 10-22, 2003.

[104] S. Dharmapurikar, H. Song, J. Turner, and J. Lockwood, "Fast packet classification using bloom filters, " In Proc. of the 2006 ACM/IEEE symposium on Architecture for networking and communications systems (ANCS '06), pp. 61-70, 2006.

[105] T. V. Lakshman and D. Stiliadis, "High-speed policy-based packet forwarding using efficient multi-dimensional range matching, " Proc. ACM Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication SIGCOMM '98, pp. 203-214, 1998.

[106] P. Gupta and N. McKeown, "Packet classification on multiple fields", In Proc. of the Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication (SIGCOMM '99), pp. 147-160, 1999.

[107] B. Xu; D. Jiang; J. Li, "HSM: a fast packet classification algorithm, " 19th International Conference on Advanced Information Networking and Applications AINA 2005, pp. 987-992, 2005.

[108] X. Gong, W. Wang, and S. Cheng, "ERFC: an enhanced recursive flow classification algorithm, " J. Comput. Sci. Technol., vol. 25, no. 5, pp. 958-969, Sept. 2010.

[109] F. Baboescu and G. Varghese, "Scalable packet classification, " IEEE/ACM Transactions on Networking, vol. 13, no. 1, pp. 2-14, 2005.

[110] P. Wang, H. Chang, C. Chan and S. Hu, "Scalable Packet Classification Using Condensate Bit Vector, " IEICE Trans. Commun., Vol. E88–B, No. 4, Special Section on Internet Technology V, pp. 1440-1447, 2005.

[111] Y. Chen, P. Wang, C. Lee, "Performance Improvement of Hardware-Based Packet Classification Algorithm, " Proc. 4th International conference on networking Part II, Lecture Notes in Computer Science, vol. 3421, pp. 728-736, 2005.

[112] C. Hsu, C. Chen, C. Lin, "Fast packet classification using bit compression, " Proc. IEEE Global Telecommunications Conference GLOBECOM 2005, vol. 2, pp. 739-743, 2005.

[113] P. Jiang, J. Liu, and Z. Qin, "On hashing techniques in networking systems, " Proc. 2010 International Conference on Information Networking and Automation (ICINA), vol. 2, pp. 444-44, 2010.

[114] A. Broder and M. Mitzenmacher, "Network Applications of Bloom Filters: A Survey, " Internet Mathematics, pp. 636-646, 2002.

[115] A. Nikitakis, and L. Papaefstathiou, "A Memory-Efficient FPGA-based Classification Engine, " Proc. 16th International Symposium on Field-Programmable Custom Computing Machines, FCCM '08, pp.53-62, 2008.

[116] W. Jiang and V. K. Prasanna, "A FPGA-based parallel architecture for scalable high-speed packet classification" Proc. International Conference on Application-Specific Systems, Architectures and Processors ASAP '09, 2009.

[117] W. Jiang, "Scalable Ternary Content Addressable Memory implementation using FPGAs, " Proc. 2013 ACM/IEEE Symposium on Architectures for Networking and Communications Systems (ANCS), pp. 71-82, 2013.

[118] T. Ganegedara, W. Jiang, and V. K. Prasanna, "A Scalable and Modular Architecture for High-Performance Packet Classification, " IEEE Transactions on Parallel and Distributed Systems, preprint, 10 Oct. 2013.

[119] Y. R. Qu; S. Zhou; and V. K. Prasanna, "High-performance architecture for dynamically updatable packet classification on FPGA, " Proc. 2013 ACM/IEEE Symposium on Architectures for Networking and Communications Systems (ANCS), pp. 125-136, 2013.

[120] L. Sun, H. Le, and V.K. Prasanna, "Optimizing Decomposition-Based Packet Classification Implementation on FPGAs, " Proc. 2011 International Conference on Reconfigurable Computing and FPGAs (ReConFig 2011), pp. 170-175, 2011.

[121] J. Van Lunteren and T. Engbersen, "Fast and Scalable Packet Classification, " IEEE Journal on Selected Areas in Communications, vol.21, no.4, pp.560-571, May 2003.

[122] H. T. Kung, "Why systolic architectures?, " Computer, vol. 15, no. 1, pp. 37-46, Jan. 1982.

[123] M. J. Foster and H. T. Kung, "The Design of Special-Purpose VLSI Chips, " Computer, vol. 13, no. 1, pp. 26-40, Jan. 1980.

[124] E. Boemo, S. López-Buedo, N. Acosta and E. Todorovich, "Local vs. Global Interconnections in Pipelined Arrays: an Example of Interaction between Architecture and Technology, " Proc. XIV Conference on Design of Circuits and Integrated Systems (DCIS'99), Nov. 1999.

[125] F. Zane, G. Narlikar, and A. Basu, "Coolcams: power-efficient TCAMs for forwarding engines, " Twenty-Second Annual Joint Conference of the IEEE Computer and Communications (INFOCOM 2003), vol. 1, pp. 42-52, 2003.

[126] S.K. Maurya and L.T. Clark, "A Dynamic Longest Prefix Matching Content Addressable Memory for IP Routing, " IEEE Transactions on Very Large Scale Integration (VLSI) Systems, vol. 19, no. 6, pp. 963-972, June 2011.

[127] M. Bando, L. Yi-Li, and H. J. Chao, "FlashTrie: Beyond 100-Gb/s IP Route Lookup Using Hash-Based Prefix-Compressed Trie, " IEEE/ACM Transactions on Networking, vol.20, no.4, pp. 1262-1275, Aug. 2012.

[128] Y.-H.E. Yang, Y. Qu; S. Haria, V.K. Prasanna, "Architecture and performance models for scalable IP lookup engines on FPGA, " IEEE 14th International Conference on High Performance Switching and Routing (HPSR 2013), pp. 156-163, 2013.

[129] A. Rasmussen, A. Kragelund, M. Berger, H. Wessing, and S. Ruepp, "TCAM-based high speed Longest prefix matching with fast incremental table updates, " IEEE 14th International Conference on High Performance Switching and Routing (HPSR 2013), pp. 43-48, 2013.

[130] F. Pong and N. Tzeng, "Concise Lookup Tables for IPv4 and IPv6 Longest Prefix Matching in Scalable Routers, " IEEE/ACM Transactions on Networking, vol. 20, no. 3, pp. 729-741, June 2012.

[131] M. Waldvogel, G. Varghese, J. Turner, and B. Plattner, "Scalable high-speed prefix matching, " ACM Trans. Comput. Syst. vol. 19, no. 4, pp. 440-482, Nov. 2001.

[132] P. Warkhede, S. Suri, and G. Varghese, "Multiway range trees: scalable IP lookup with fast updates, " Comput. Netw. no. 44, vol. 3, pp. 289-303, Feb. 2004.

[133] Y. Qu and V. K. Prasanna, "High-Performance Pipelined Architecture for Tree-Based IP Lookup Engine on FPGA, " Proc. IEEE International Symposium on Parallel & Distributed Processing, pp. 114-123, 2013.

[134] H. Song and J. S. Turner, "Toward Advocacy-Free Evaluation of Packet Classification Algorithms, " IEEE Transactions on Computers, vol. 60, no. 5, pp. 723-733, May 2011.

[135] C. A. Zerbini and J. M. Finochietto, "Performance evaluation of packet classification on FPGA-based TCAM emulation architectures, " Proc. IEEE Global Communications Conference (GLOBECOM), pp. 2766-2771, 2012.

[136] H. Song and J. W. Lockwood, "Efficient packet classification for network intrusion detection using FPGA, " Proc. ACM/SIGDA 13th international symposium on Field-programmable gate arrays (FPGA '05), pp. 238-245, 2005.

[137] D. Taylor, J. Turner, J. Lockwood, T. Sproull, and D. Parlour, "Scalable IP Lookup for Internet Routers, " IEEE Journal on Selected Areas in Communications, no. 21, pp. 522-534, May 2003.

[138] Xilinx. Contend-Addressable Memory v4.0. Xilinx Product Specification DS253 (v1.0), March 2003.

[139] W. Jiang and V. K. Prasanna, "Field-split parallel architecture for high performance multi-match packet classification using FPGAs, " Proc. 21st annual symposium on Parallelism in algorithms and architectures (SPAA '09), pp. 188-196, 2009.

[140] O. Ahmed, S. Areibi, K. Chattha, and B. Kelly, "PCIU: Hardware Implementations of an Efficient Packet Classification Algorithm with an Incremental Update Capability, " International Journal of Reconfigurable Computing, vol. 2011, Article ID 648483, 21 pages, 2011.

[141] T. Ganegedara and V.K. Prasanna, "StrideBV: Single chip 400G+ packet classification, " Proc. IEEE 13th International Conference on High Performance Switching and Routing (HPSR), pp. 1-6, 2012.

[142] A. Sanny, T. Ganegedara, and V.K. Prasanna, "A Comparison of Ruleset Feature Independent Packet Classification Engines on FPGA, " Proc. IEEE 27th International Parallel and Distributed Processing Symposium Workshops & PhD Forum (IPDPSW), pp. 124-133, 2013.

[143] S. Choi, R. Scrofano, V. K. Prasanna, and J. Jang, "Energy-efficient signal processing using FPGAs, " Proc. ACM/SIGDA eleventh international symposium on Field programmable gate arrays (FPGA '03), pp.225-234, 2003, doi: 10.1145/611817.611850.

[144] T. Ganegedara, V. Prasanna, and G. Brebner, "Optimizing packet lookup in time and space on FPGA, " Proc. 22nd International Conference on Field Programmable Logic and Applications (FPL), pp.270-276, 2012.

[145] R. Tessier, V. Betz, D. Neto, and T. Gopalsamy, "Power-aware RAM mapping for FPGA embedded memory blocks, " Proc. ACM/SIGDA 14th international symposium on Field programmable gate arrays (FPGA '06), pp.189-198, 2006.

[146] "Embedded memory blocks in Stratix V devices, " Altera Corporation, http://www.altera.com/literature/hb/stratix-v/stx5_51003.pdf, May 2013.

[147] Y. Qi; J. Fong, W. Jiang, B. Xu, J. Li, and V. Prasanna, "Multi-dimensional packet classification on FPGA: 100 Gbps and beyond, " Proc. 2010 International Conference on Field-Programmable Technology (FPT), pp.241-248, 2010, doi: 10.1109/FPT.2010.5681492.

[148] K. Kogan, S. I. Nikolenko, W. Culhane, P. Eugster and E. Ruan, "Towards efficient implementation of packet classifiers in SDN/OpenFlow, " Proc. HotSDN 2013, pp. 153-154, 2013.

[149] B. Yang, X. Wang, Y. Xue, and J. Li, "DBS: A Bit-level Heuristic Packet Classification Algorithm for High Speed Network, " Proc. 15th International Conference on Parallel and Distributed Systems (ICPADS), pp. 260-267, 2009.

[150] B. Yang, J. Fong, W. Jiang, Y. Xue, and J. Li, "Practical Multituple Packet Classification Using Dynamic Discrete Bit Selection, " IEEE Transactions on Computers, vol. 63, no. 2, pp. 424-434, Feb. 2014.

[151] O. Ahmed, S. Areibi, and G. Grewal, "Hardware Accelerators Targeting a Novel Group Based Packet Classification Algorithm, " International Journal of Reconfigurable Computing, vol. 2013, Article ID 681894, 33 pages, 2013.

[152] F. Baboescu, P. Warkhede, S. Suri, and G. Varghese, "Fast packet classification for two-dimensional conflict-free filters, " Computer Networks, vol. 50, no. 11, pp. 1831-1842, Aug. 2006.

[153] C. J. Huang, C. Chen, C. S. Chou, and S. T. Kao, "Fast Packet Classification Using Multi-Dimensional Encoding, " Proc. Workshop on High Performance Switching and Routing HPSR '07, pp. 1-6, 2007.

[154] X. Sun; S.K. Sahni, S.K.; Zhao, Y.Q., "Packet classification consuming small amount of memory, " IEEE/ACM Transactions on Networking, vol. 13, no. 5, pp. 1135-1145, Oct. 2005.

[155] P. Wang, "Scalable packet classification with controlled cross-producting, " Computer Networks, vol. 53, no. 6, pp. 821-834, Apr. 2009.

[156] Kai Zheng; Zhiyong Liang; Yi Ge, "Parallel packet classification via policy table pre-partitioning,"Proc. IEEE Global Telecommunications Conference GLOBECOM '05, 2005.

[157] J. Kleinberg and E. Tardos, *Algorithm Design,* Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2005.

[158] M. Ahmadi, S. A. Ostadzadeh, and S. Wong, "An Analysis of Rule-set Databases in Packet Classification, " Proc. 18th Annual Workshop on Circuits, Systems and Signal Processing, ProRisc 2007, pp. 24–30, 2007.