RNPS No. 2143
ISSN 2072-6260
Versión Digital

CENATAV
Centro de Aplicaciones de Tecnologías de Avanzada
MINISTERIO DE LA INDUSTRIA BÁSICA

REPORTE TÉCNICO
Minería de Datos

SERIE GRIS

Algoritmos paralelos para la identificación de cadenas en flujos de datos usando hardware reconfigurable

José Manuel Bande Serrano y
José Hernández Palancar

RT_015 **octubre 2011**

7ma. No. 21812 e/218 y 222,
Rpto. Siboney, Playa;
La Habana.
Cuba. C.P. 12200
www.cenatav.co.cu

RNPS No. 2143
ISSN 2072-6260
Versión Digital

SERIE GRIS

REPORTE TÉCNICO
Minería de Datos

Algoritmos paralelos para la identificación de cadenas en flujos de datos usando hardware reconfigurable

José Manuel Bande Serrano y
José Hernández Palancar

RT_015 **octubre 2011**

CENATAV
Centro de Aplicaciones de Tecnologías de Avanzada
MINISTERIO DE LA INDUSTRIA BÁSICA

7ma. No. 21812 e/218 y 222,
Rpto. Siboney, Playa;
La Habana.
Cuba. C.P. 12200
www.cenatav.co.cu

Tabla de contenido

Lista de figuras

Lista de tablas

Algoritmos paralelos para la identificación de cadenas en flujos de datos usando hardware reconfigurable

José Manuel Bande Serrano[1] y José Hernández Palancar[2]

[1] Dpto. Redes y Seguridad Informática, Centro de Aplicaciones de Tecnologías de Avanzada (CENATAV),
Ciudad Habana, Cuba
jbande@cenatav.co.cu
[2] Dpto. Minería de Datos, Centro de Aplicaciones de Tecnologías de Avanzada (CENATAV),
Ciudad Habana, Cuba
jpalancar @cenatav.co.cu

RT_015, Serie Gris, CENATAV
Aceptado: 24 de septiembre de 2010

Resumen. El reconocimiento de cadenas es un problema antiguo y resuelto por un conjunto del algoritmos eficientes, (lineales, y sublineales con respecto al tamaño de los datos de entrada). Además constituye una de las operaciones más comunes en aplicaciones prácticas contemporáneas. Por otra parte, el poder de cálculo requerido para su solución aumenta continuamente en correspondencia con el incremento de las velocidades de los flujos de datos y, de los conjuntos de cadenas a detectar. El poder de cálculo requerido supera, en no pocas ocasiones, al alcanzado por implementaciones de los algoritmos más eficientes sobre *Procesadores de Propósito General (GPP)*. Esto de debe en gran medida, a la naturaleza secuencial los GPP. Alternativamente, la tecnología de *Arreglos de Compuertas Programable por Campo (FPGA, Field Programmable Gate Array)*, ofrece un amplio rango de posibilidades debido al grado de paralelización en los procesos que se puede alcanzar con estos dispositivos. En este Reporte Técnico, hacemos primeramente una revisión del estado del arte de los algoritmos eficientes desarrollados sobre GPP, luego enfocados en las facilidades de los FPGAs, hacemos un breve estudio a modo de introducción de su arquitectura interna y prestaciones. Continuamos con el estado del arte de los algoritmos implementados en FPGAs con el objetivo de evaluar las tendencias de diseño y estrategias de optimización mayormente utilizados por su eficiencia. Proponemos, un algoritmo de procesamiento paralelo que reconoce cadenas de un flujo de datos procesando dos caracteres por ciclo. Por último hacemos un análisis crítico y evaluamos de posibles áreas de investigación.

Palabras clave: reconocimiento de cadenas, NIDS, FPGA.

Abstract. String matching is one of the most common and high-execution needed problem, especially, if high-speed and a huge amount of string recognizing is required in a data flow as is the case in applications involved in network security such as Network Intrusion Detection Systems (INDS) applications. The sequential nature of the General Purpose Processors (GPP) makes the performance of the algorithms implemented on this processors lay down under the actual speed requirements. However, the Field Programmable Gate Array (FPGA) technology offers a wide range of possibilities due to its capability of reconfiguration and parallelism. In this Technical Report, we first, make an overview of the art's state of the efficient software implementations aimed to reach high throughput, we then introduce a brief study of the reconfigurable devices architectures. Later, we make a study of architectures and techniques implemented in hardware with the intention of understanding, and identify pattern designs and optimization strategies asserted by its efficiency, in order to find better solutions to the efficiency problem imposed by the string matching on high speed data flows. Then we propose an architecture capable of processing tow bytes per cycle. At the end we make a critical analisys and evaluate possibles researching fields.

Keywords: string matching, NIDS, FPGA.

1. Introducción

Encontrar en un flujo de datos a una elevada velocidad, secuencias de símbolos con algún interés particular, sin tener que realizar para este proceso ninguna regulación en la velocidad de dicho flujo de datos, es posiblemente el entorno más difícil que pueda enfrentar una aplicación que realiza reconocimiento de cadenas.

Aplicaciones de este tipo son principalmente necesarias en el entorno de las redes informáticas, donde la información se transmite mediante flujos de datos con velocidades elevadas y es necesario la detección de cierta información en dichos flujos sin afectar su velocidad, por ejemplo, un *router* necesita reconocer la dirección IP de destino de cada paquete de internet para poder enrrutarlo. Con este fin se debe realizar un reconocimiento de cadenas de dicho paquete contra su tabla de rutas identificando el próximo destino que le corresponde a dicho paquete, esta operación debe ser muy rápida para evitar sobrecargas en las colas que generen cuellos de botella, afectando la calidad del servicio.

El ámbito de la seguridad informática es quizás el más exigente en cuanto al desempeño exigido a los algoritmos que ejecuta el reconocimiento de cadenas. Los antivirus por ejemplo, inspeccionan los códigos de los programas en busca de cadenas identificadas como códigos malignos o parte de estos. Esta tarea de reconocimiento de cadenas requiere un elevado esfuerzo computacional, téngase en cuenta la gran cantidad de códigos maliciosos que pueden existir.

Aún más exigente que las aplicaciones de tipo antivirus, están aquellas cuyo objetivo es realizar el reconocimiento sobre un flujo de datos en tiempo real, precisamente lo que se ha denominado como, Sistemas de Detección de Intrusos, *(NIDS, Network Instrusion Detection Systems)*.

Los NIDS realizan la inspección de miles de cadenas sobre flujos de datos con velocidades del orden de los Giga bits por segundo $(Gbps)$. Por otra parte, la complejidad y el número de delitos informáticos obliga, si se desea una red segura, a realizar una inspección exhaustiva sobre todos los campos de los paquetes de red, detectando un número elevado de cadenas, y no sólo eso, detectando comportamientos anómalos mediante la detección de patrones de ocurrencias de varias cadenas. Estos patrones indican acciones mal intencionadas en contra de la calidad de servicio de la red, violaciones de las políticas de seguridad, o presencia de códigos malignos.

La eficiencia computacional requerida por los NIDS para realizar la detección eficaz de todas las posibles agresiones o violaciones a la seguridad de las redes informáticas no es un problema resuelto, dándose solamente soluciones parciales, o totales solamente en un período limitado de tiempo. Lo anterior se debe a las exigencias que deben superar este tipo de aplicaciones y a las limitaciones de la tecnología subyacente para lograrlo. Identificamos estas exigencias como:

1. Lograr capacidades de procesamiento que permitan desempeñar el reconocimiento de cadenas en los flujos de datos, soportando las velocidades actuales de transmisión en dichos flujos establecidas por el desarrollo tecnológico.
2. Detectar de forma concurrente el número total de cadenas que se requiera por la aplicación, este número puede ser elevado y creciente.
3. Lograr la actualización de dicho conjunto de cadenas de forma sencilla y rápida.

Ahora bien, desde el punto de vista tecnológico, el aumento gradual de la velocidad en la transmisión en flujo de datos, rebasa, en muchas ocasiones las capacidades de procesamiento de los algoritmos más eficientes implementados sobre Procesadores de Propósito General (GPP). Esto se debe a la forma secuencial en la que se ejecutan las instrucciones, que dotan a estos dispositivos de gran flexibilidad pero limitan la velocidad a la que pueden trabajar.

Este hecho ha inducido a la investigación y al desarrollo algoritmos implementados en arquitecturas de hardware, donde se puede definir circuitos independientes para aquellas funciones que se pueden ejecutar

en el mismo momento. Esto imprime una elevada concurrencia de procesos, a la vez que incrementa la velocidad y capacidad de procesamiento, a esta acción se le denomina Paralelismo o Concurrencia.

Los FPGAs, *(del inglés, Field Programmable Gate Array)*, son dispositivos de lógica reconfigurable, que ofrecen esta posibilidad, y no sólo eso, además se puede modificar completa o parcialmente cualquier función implementada sobre ellos, eso es reconfiguración. Esta cualidad permite la actualización del grupo de cadenas que se necesita reconocer, correspondiendo a nuevas formas de ataques a la red. Aún con estas ventajas los FPGAs presentan algunas limitaciones físicas que es importante destacar, estas son:

1. Los recursos del FPGA, o sea los elementos utilizados para implementar toda la lógica del hardware, son limitados y su número depende del tipo y la familia de cada dispositivo.
2. La frecuencia de operación de los dispositivos FPGA es relativamente baja, (Orden de los Mhz) comparada con los GPP.

La primera desventaja impone un límite al número de cadenas que pueden reconocer los algoritmos en dependencia la cantidad de recursos de hardware disponibles en el dispositivo. En los sistemas de detección de intrusos este conjunto de cadenas se encuentra en constante aumento, dándose soluciones capaces de detectar la totalidad de las cadenas maliciosas sólo en un período de tiempo no muy largo.

La segunda, por otra parte, implica que lograr las elevadas capacidades de procesamiento requeridas (Orden de los Gbps), depende en gran medida del nivel de paralelismo involucrado, entrando generalmente en contradicción con la limitante anterior, puesto que el aumento de paralelismo siempre conlleva a un aumento en el consumo de recursos.

Por estos motivos, la búsqueda de soluciones creativas que logren una posición favorable entre estas dos limitantes cumpliendo con los requerimientos de velocidad y capacidad de número de cadenas buscadas, actuales y futuras, es todo un reto investigativo e ingenieril, que ha sido abordado y superado temporalmente por múltiples investigadores en los últimos 30 años.

2. Problema del reconocimiento de cadenas

Comenzaremos por aclarar qué significa el término “cadena”: en nuestro contexto, una cadena no es más que una determinada secuencia finita de símbolos pertenecientes a un alfabeto, también finito. Pueden ser los símbolos que componen el alfabeto alfanumérico, las letras que representan las bases nitrogenadas del código genético, o en el caso computacional, el código ASCII compuesto por 256 símbolos o caracteres. Podemos ahora, ofrecer una definición más formal al problema del “Reconocimiento de Cadenas” (Navarro y Raffinot, Cap. 2, pp. 15 [33]):

- **Reconocimiento de cadenas:** Dado una cadena p=$p_1, p_2, \ldots, p_m$ y un texto largo $T = t_1, t_2, \ldots, t_n$, donde T y p son secuencia de caracteres de un grupo finito que denominaremos, alfabeto y denotaremos por Σ, encontrar todas las ocurrencia de p en T.

De igual manera se extiende este concepto para el caso del reconocimiento de múltiples de cadenas, el que sería:

- **Reconocimiento múltiple de cadenas:** Sea $P = \{p_1, p_2, \ldots, p_r\}$ un conjunto donde cada p_i es una cadena $p_i = p_{1i}, p_{2i}, \ldots, p_{mi}$ sobre un alfabeto Σ. Encontrar todas las cadenas de P que se encuentren en un texto $T = t_1, t_2 ... t_n$.

Comúnmente en la lengua inglesa, se utiliza el término *match* para afirmar la existencia de una cadena previamente conocida en un texto, nosotros utilizaremos los términos, cotejo, detección, u ocurrencia

indistintamente para denotar esta misma idea. Además de las definiciones anteriores es importante destacar algunos conceptos relacionados con el reconocimiento de cadenas (Crochemore y Rytter, Cap. 1, pp. 10-13 [19]):

- **Prefijo, sufijo, y factor:** Dado las cadenas x, y, z decimos que x es *prefijo* de xy, *sufijo* de yx, y *factor* de yxz.
- **Longitud del texto y de la cadena:** Sea un Texto $T = T[0 \ldots n-1]$, su cardinal o tamaño $|T| = n$, es la cantidad de caracteres que lo conforman contando las repeticiones y $T[i]$ el $i-esimo$ caracter del mismo. Similarmente la cadena $u = u[0 \ldots m-1]$, tiene tamaño $|u| = m$ y $u[j]$ su $j-esimo$ caracter.
- **Subcadena o subsecuencia:** Una cadena x es *subcadena* o *subsecuencia* de otra cadena, si x puede obtenerse a partir de esta, eliminando cero o más caracteres [2], *ej.*, la cadena $u = abaaba$, una *subcadena* de este sería $v = aab$.
- **Borde:** Un *borde* de x, denotado por $b(x)$, es toda subsecuencia que sea prefijo y sufijo a la vez, así por ejemplo: sea $u = abaaba$, los bordes de u serian, a, aba, y $abaaba$ siendo este último el borde trivial, o sea la cadena completa.
- **Ventana de búsqueda:** En un texto, la búsqueda o comparación de los elementos de la cadena con los elementos del texto se realizará dentro de los límites de la *ventana de búsqueda* (de longitud generalmente igual a la de la cadena buscada), comparando cada carácter de la cadena en la ventana, con el carácter en el texto alineado con este, y obteniendo para cada comparación un resultado booleano, verdadero o falso. Si todos los caracteres dentro de la ventana coinciden con los correspondientes caracteres del texto hemos obtenido una ocurrencia, o cotejo de la cadena, en otras palabras hemos hallado la cadena en el texto.

2.1. Tipos de procesamiento para el reconocimiento de cadenas

El reconocimiento de cadenas se caracteriza además por el entorno donde es empleado, lo que va a influir en la elección o desarrollo de los algoritmos. Los algoritmos responderán a estos entornos según el tipo de procesamiento que ejecutarán. Se denotan tres tipos fundamentales de procesamiento (Crochemore y Rytter, Cap. 1, pp. 8-9 [19]):

- **Fuera de línea:** Todos los datos de entrada, o sea todo el texto, puede ser almacenado en memoria antes de comenzar el procesamiento. En este caso nos interesará la eficiencia total del algoritmo para procesar todo el texto, o sea, la complejidad del algoritmo para el tiempo total de ejecución. El resultado que se espera es el resultado final, no nos interesan resultados intermedios o parciales.
- **En línea:** Los datos son procesados en porciones, ofreciendo un resultado parcial de cada procesamiento, la entrada puede ser tratada como una secuencia infinita de datos, nos interesa el resultado parcial de cada procesamiento, y el tiempo que toman en dar una respuesta para cada dato de entrada procesado, así, existirá una secuencia de entrada, y una de salida. Los algoritmos con este tipo de procesamiento deben de dar una respuesta antes de procesar la próxima porción de datos.
- **Tiempo real:** Son aquellos algoritmos con procesamiento En Línea óptimos, es decir, el tiempo en que procesan un dato de entrada y que dan una respuesta está acotado por una constante.

Es de especial interés aquellos algoritmos con tipo de procesamiento en línea y en tiempo real, puesto que no se conoce de antemano la longitud de la entrada, se asume que esta es infinita y además no perdurable, no podemos disponer de ella todo el tiempo, tal como sucede en un flujo de datos. Si dicho flujo posee una velocidad que supera el tiempo que toma el algoritmo en dar una respuesta, obviamente esta información desaparece quedando el algoritmo inservible, o asumiendo la pérdida de información como una penalización en cuanto a eficacia para resolver el problema.

2.2. Reconocimiento de cadenas para la seguridad de redes informáticas

Antes de abordar el problema desde el punto de vista de la satisfacción de las exigencias computacionales, analizaremos brevemente los costos, y daños debido a violaciones a la seguridad de la información que posicionan el problema de la seguridad informática como uno de los más apremiantes y de mayor actualidad.

Para ello nos referiremos al informe anual que ha venido realizando durante trece años el *Computer Security Institute* [1] que en encuesta realizada en el 2008 a 522 compañías en los Estado Unidos que practican políticas de seguridad, arrojó un promedio de pérdidas de 300,000 dólares por compañía a causa de violaciones a la seguridad utilizando como medio la red, el 44 % de las causas de pérdidas se deben a invasiones externas, o sea de ataques a la red ejecutado por intrusos.

De las aplicaciones utilizadas para la seguridad informática, los Antivirus conforman del 97 %, los *Firewall* el 94 %, y los Sistemas de Detección de Intrusos el 79 %. El hecho de que los NIDS estén por debajo de los Antivirus y de los *Firewalls*, aún cuando realizan funciones de ambos y otras más complejas, se debe a que el poder de computo necesario para que se desempeñen de de forma eficaz es mayor que el que pueden ofrecer la mayoría de los Procesadores de Propósito General.

En el ámbito de la seguridad de las redes informáticas se es particularmente exigente en cuanto a la eficiencia de los algoritmos que ejecutan reconocimiento de cadenas. Dos cotas altas impone este medio, primero, el volumen de cadenas a buscar en el flujo de datos es elevado y creciente, segundo, las velocidades de los flujos de datos son también elevadas y crecientes.

Los NIDS, realizan la inspección de paquetes mediante un conjunto de reglas, donde cada regla, representa la detección y respuesta a una violación a la seguridad de la red, e involucran chequeo de las cabeceras de los paquetes, puertos, protocolos, direcciones IP, y de los datos en la capa de aplicación, es en este último caso donde más esfuerzo computacional se invierte (31 % a 80 % del tiempo de ejecución según Sourdis y otros autores en [39]), dado que no se conoce de antemano la posición de la cadena o cadenas buscadas y es el campo de datos el más extenso en los paquetes de red.

Entre los NIDS más conocidos encuentran Snort, Linux L7-Filter, y Bro. De ellos el más difundido y de código abierto es Snort. Tomemos por ejemplo, la siguiente regla de snort [2]:

- alert tcp any any $->$ any 21 (flow:to_server, established; content: “root”; pcre:“/ user\s+root/i”;)

Esta regla generará una alerta si detecta un paquete con protocolo TCP, proveniente de cualquier dirección IP y cualquier puerto dirigido a cualquier dirección IP con puerto 21, que contenga entre otras opciones, en su carga de datos la cadena “*root*”, o si además contiene la cadena “*user*” separada entre uno o más espacios de la cadena “*root*”.

La palabra clave “*content*”, indica buscar la cadena entre comillas en cualquier posición dentro de la carga de dados, la palabra clave “*pcre*” indica que la búsqueda se hará mediante una expresión regular, esta es una forma de representar varias cadenas en una sola expresión, más adelante abordaremos el tema de las expresiones regulares de forma más detallada.

Para contabilizar el número de reglas, cadenas, y expresiones regulares, hemos extraído mediante un programa de lineas de comando, las reglas de Snort 2.8.5 del año 2009, en total obtuvimos, 5,968 reglas activadas (o sea que no estaban comentadas), de ellas 1,882 expresiones regulares diferentes definidas por la palabra clave *pcre*, y 6,757 cadenas estáticas definidas por las palabras clave “*content*” y “*uricontent*”.

Este elevado número de cadenas junto con la elevada velocidad del flujo de datos que soportan los puntos de acceso a las redes actuales, que van de $10Mbps$ a $10Gbps$ para los estándares de Ethernet sin mencionar las perspectivas de incremento para las redes de nueva generación son el reto principal que enfrenta el reconocimiento de cadenas en la inspección de paquetes de red.

Una aplicación NIDS que no logre procesar en un tiempo adecuado cada paquete puede provocar un cuello de botella, o sea que se llenen las colas en los dispositivos de interconexión, de manera que se pierdan paquetes, o que al verse desbordado no pueda brindar el servicio esperado a los usuarios, lo cual puede muy bien consistir en un ataque a la red si se hace con toda intención por alguna persona que conozca las limitaciones del mecanismo de inspección. Por otra parte, si para evitar el cuello de botella no se realiza una inspección exhaustiva al 100 % de los paquetes se corre el riesgo que se infiltren paquetes maliciosos o no deseados a la red interna.

Por lo tanto la selección del algoritmo adecuado juega un papel fundamental. A todas luces a de ser un algoritmo con procesamiento en línea o mejor en tiempo real, siendo de mayor valía aquellos que presentan mejor complejidad para el peor caso. Por lo dicho anteriormente, si conociendo el algoritmo y cual entrada lo hace trabajar más ineficientemente se puede vulnerar la seguridad, se hacen necesarios algoritmos cuyo desempeño no dependa de las características del flujo de entrada.

Aún con los algoritmos secuenciales más eficientes que se puedan implementar, no se satisfacen los requerimientos de número de cadenas y velocidad en muchas aplicaciones. El problema del reconocimiento de cadenas posee un paralelismo intrínseco elevado a partir del hecho de que ninguna cadena depende de otra en el conjunto, este paralelismo no puede ser explotado en toda su magnitud por los Procesadores de Propósito General. Este hecho a inducido a la búsqueda de nuevas plataformas tecnológicas que permitan desarrollar algoritmos con un alto nivel de paralelismo.

3. Algoritmos para el reconocimiento de cadenas

Luego de presentarse una ocurrencia, (todos los caracteres en la ventana coincidieron con el texto) o un fallo, (Ningún caracter coincidió, o coincidieron hasta cierta distancia), debemos desplazar la ventana para comenzar una nueva búsqueda, con lo cual se nos presentan dos situaciones:

1. Si ha habido una ocurrencia, desplazar la ventana a una distancia igual a la distancia de la cadena. Esto podría implicar que en el caso de cadenas en las que un sufijo es también prefijo de la misma podríamos perder la posibilidad de encontrarlos en el texto.
2. Si no ha habido ocurrencia, desplazar la ventana un caracter después del caracter fallido. Al igual que en el caso anterior si la parte leída que coincide con la cadena existe un sufijo que es también prefijo de esta perderíamos la posibilidad de encontrar una ocurrencia.

La forma natural que nos permitiría desplazar la ventana de una forma siempre segura, sería simplemente, haya o no una ocurrencia, desplazar la ventana una posición. Esto nos conduce a un algoritmo de fuerza bruta mediante el cual no se perdería ninguna posible ocurrencia, a costa de realizar varias comparaciones sobre los mismos caracteres del texto.

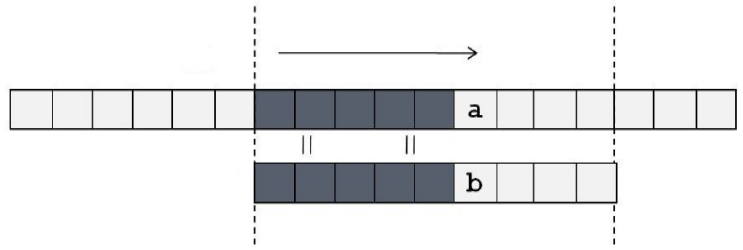
Para el peor caso, en el cual se comparan todos los caracteres de la ventana de búsqueda y luego esta se desplaza una posición, se realizan $m \times n$ comparaciones, tomando $O(n \times m)$ tiempo. Como caso promedio se esperan $2n$ comparaciones (Charras y Lecroq, 2004, [13]). Nótese la dependencia de este algoritmo en cuanto a su tiempo de procesamiento con la correlación entre la cadena y el texto, o sea que su tiempo de procesamiento depende en gran medida de las características del texto.

Los algoritmos desarrollados para la comparación de cadenas buscan elevar la velocidad de procesamiento total del texto evitando las comparaciones repetitivas y procesos innecesarios, sin perder ocurrencias, a un costo computacional admisible. Los algoritmos eficientes para el reconocimiento de cadenas son aquellos que presentan una complejidad de tiempo lineal o sublineal con respecto al tamaño del texto y de las cadenas, visto de otra forma, con orden de tiempo menor que $O(n \times m)$.

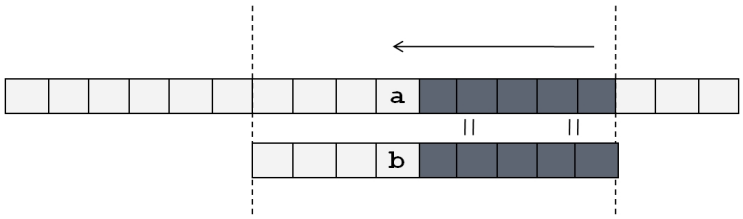
3.1. Métodos de búsqueda

Existen tres formas secuenciales mediante las cuales los algoritmos realizan la búsqueda, o comparación de caracteres dentro de la ventana (Navarro y Raffinot, Cap. 2, pp. 15-17 [33]). Asumimos para esta definición el sentido natural de lectura:

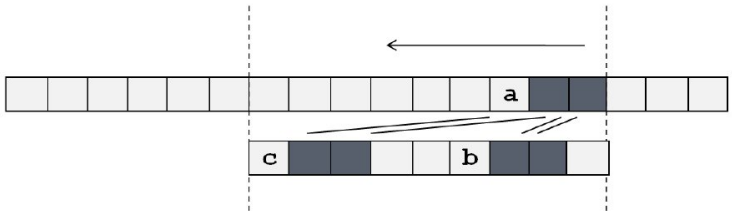
- **Búsqueda por prefijo:** El proceso de comparación de caracteres se realiza de izquierda a derecha dentro de la ventana buscando el prefijo más largo dentro del texto que sea a su vez prefijo de la cadena, figura 1a.
- **Búsqueda por sufijo:** El proceso de comparación de caracteres se realiza de derecha a izquierda en la ventana, buscando el mayor sufijo que es también sufijo de la cadena. Esta forma de búsqueda nos permite, en promedio, evitar leer algunos caracteres del texto, figura 1b.
- **Búsqueda por factor:** El proceso de comparación de caracteres se realiza de derecha a izquierda, pero en este caso se busca el sufijo leído en el texto que es también factor del de la cadena, figura 1c.



(a) prefijo



(b) sufijo



(c) factor

Fig. 1. Tipos de búsqueda, la parte sombreada indica los caracteres del texto que cotejan con un los de la cadena.

3.2. Algoritmos con búsqueda por prefijo

A continuación presentaremos el estado del arte de los algoritmos para el reconocimiento de cadenas, considerados Eficientes, estos presentan una complejidad acotada superiormente por un polinomio de

grado pequeño y algunos son sublineales como caso promedio, esto quiere decir que pueden evitar la lectura de algunos caracteres. La forma en que los hemos clasificados no es exclusiva, y se basa más bien en lo más distintivo de cada algoritmo y lo que más interés representa para nuestro estudio. En lo adelante tomaremos las variables m y n como los elementos de las cadenas, los elementos del texto respectivamente.

3.2.1. Morris-Pratt (MP)

Hagamos la siguiente observación, suponga que buscamos la cadena $u = abaaba$ en un texto, note que en la misma a y aba son sufijos y prefijos, o sea bordes, como ya hemos definido en la sección 2, $b(u) = a, aba$. Suponga ahora que hemos analizado un texto en el que sus caracteres coinciden hasta $x = abaab$ aquí x es el prefijo más largo leído del texto que coincide con u, si la comparación con el próximo carácter falla, entonces dado que $bx = ab$ es el borde más largo del texto leído x, se puede desplazar la ventana $|x| - |bx| = 3$ espacios, luego $x = b(x)$ o sea $x = ab$, y su borde $bx = \varepsilon$ o cadena vacía, dado que no posee ningún sufijo que sea a la vez prefijo de la misma.

Al repetir el proceso, si el próximo caracter del texto acierta con el próximo caracter de la cadena, que sería a, entonces $x = aba$ con $b(x) = a$, si no acierta, la ventana se desplaza $|x| - |bx| = 2$, quedando $x = b(x) = \varepsilon$ y recomenzando la búsqueda.

El algoritmo *Morris-Pratt* presentado por J. H. Morris y V. R. Pratt, en 1970 [31] precalcula los bordes más largos de todos los prefijos posibles de una cadena, esto constituye un preproceso, luego, utiliza búsqueda por prefijo reconociendo el prefijo más largo leído del texto que es también prefijo de la cadena, cuando falla en algún caracter, realiza el proceso descrito anteriormente, de esta manera evita ejecutar fuerza bruta aunque lee los caracteres al menos una vez.

La fase de preprocesamiento tiene complejidad $O(m)$ en espacio y tiempo. La fase de búsqueda toma $O(m+n)$ tiempo y realiza al menos $2n-1$ comparaciones. Es de interés destacar que este algoritmo fue el primero en tener una complejidad de tiempo de búsqueda lineal (Charras y Lecroq, 2004, [13]). Al ser $O(m+n)$ implica que el tiempo en que dará respuesta de todas las ocurrencias una cadena en un texto, es lineal con respecto a la suma del tamaño del texto y el tamaño de la cadena.

3.2.2. Knuth-Morris-Pratt (KMP)

En MP, al fallar la búsqueda en un caracter, se debe desplazar la ventana para realizar una nueva comparación con el caracter inmediato al nuevo prefijo asumido. Si este es igual al caracter de la cadena en el que se falló al comparar con el texto entonces se realizará una comparación innecesaria puesto que arrojará el mismo resultado.

Donal E. Knuth y otros autores, 1977 [25] mejoran el algoritmo MP, implementado una función que precalcula los bordes más largos de todos los prefijos posibles de una cadena seguidos de un caracter diferente, a este algoritmo se le conoce como Knuth-Morris-Pratt (KMP). Mediante dicha función se eliminan comparaciones que de antemano se sabe que el resultado sería fallido, elevando el desempeño del algoritmo al acotar el número máximo de comparaciones para un solo caracter por $\log_{\Phi}(m+1)$ donde $\Phi = \dfrac{1+\sqrt{5}}{2}$ denominado *golden ratio* (Charras y Lecroq, 2004, [13]).

Otros algoritmos que han seguido la filosofía de KMP son, *Colussi, Gail-Giancarlo* y *Apostolico-Crochemore*, todos con $O(m)$ espacio y $O(n)$ tiempo. Por último, KMP ve su extensión al reconocimiento de múltiples cadenas en el algoritmo *Aho-Corassick* [3] que presentaremos más adelante.

3.3. Algoritmos con búsqueda por sufijo

Una notable cualidad de los algoritmos que desarrollan la búsqueda mediante sufijos es que pueden ser sublineales, como ya hemos mencionado esto quiere decir que puede evitar procesar caracteres, al menos como caso promedio.

3.3.1. Booyer-More (BM)

El algoritmo Boyer-Moore, (Robert S. Booyer y J. Strother Moore, 1977 [9]) precalcula tres funciones de desplazamiento d_1, d_2 y d_3, para todas ellas habremos leído un sufijo v en la ventana de búsqueda que es sufijo de la cadena buscada u, y habremos fallado al leer el próximo caracter, σ. Note que se está utilizando la búsqueda por sufijo. Cada una de las funciones corresponden con las siguientes situaciones.

1. Desplazar la ventana de búsqueda hasta alinearse con la próxima ocurrencia v en u lo más a la derecha posible y que sea precedido por un caracter distinto al caracter donde se falló. La idea consiste en precalcular para cada sufijo de la cadena esta distancia.
2. Si el sufijo v no ocurre en ninguna otra posición como factor de u. Desplazar la ventana de búsqueda hasta el sufijo más largo de v que sea prefijo de u.
3. Si no se han cumplido ninguna de las dos condiciones anteriores, alinear el carácter del texto, con el caracter que coincida en u lo más a la derecha posible, si no existe en u tal caracter la ventana de búsqueda se desplaza una posición después del caracter donde se falló en el texto. Note que esta condición puede dar como resultado un caracter que coincida con el fallido pero que ya se haya procesado como parte de v, en este caso BM toma el mayor de los desplazamientos entre los otorgados por las tres condiciones.

BM consume $O(m+|\Sigma|)$ espacio y la parte de búsqueda toma $O(m \times n)$ tiempo para el peor caso, pero presenta comportamiento *sublineal* en promedio. El principal inconveniente que lo hace poco ágil es el procesamiento de sus tres funciones.

Los algoritmos descendientes de BM son, entre otros, *Turbo-MB* y *Apostolico-Giancarlo* $O(m+|\Sigma|)$ espacio y $O(n)$ tiempo, *Reverse-Colussi* $O(m \times |\Sigma|)$ espacio y $O(n)$ tiempo, *Hoorspool* $O(|\Sigma|)$ espacio y $O(m \times n)$ tiempo, este último ejecuta ente $\frac{1}{|\Sigma|}$ y $\frac{2}{|\Sigma|+1}$ comparaciones por caracter en promedio (Charras y Lecroq, 2004, [13]). Hoorspool es extendido para reconocimiento múltiple en *Wu-Mamber* [43] que presentamos a continuación.

3.3.2. Wu-Manber (WM)

Wun Su y Udi Manber presentan en [43] el algortimo Wu-Manber, el cual utiliza una función h_1, para "desmenuzar" *(hashing en inglés)* una cadena en bloques de caracteres, por ejemplo tomando bloques de dos caracteres, la palabra "cadena" generaría los bloques "ca, ad, de, en, na", de esta manera y para todas las cadenas del grupo crea una tabla SHIFT que relaciona cada bloque con los desplazamientos mínimos hasta el final de la cadena.

Una vez detectado un bloque de caracteres en el texto idéntico a algún bloque final de una cadena o grupo de cadenas, se utiliza este como índice para direccionar la posición en la lista de cadenas donde pueden estar las cadenas coincidentes con este bloque final, esto se hace mediante el uso de otra tabla denominada *HASH*, que relaciona los bloques finales de cada cadena del grupo con la dirección en que se encuentra dentro del grupo de cadenas. Se procede entonces, a comparar en la ventana, mediante búsqueda por sufijo, cada cadena especificada con el texto hasta encontrar una ocurrencia o no.

La principal desventaja que tiene este algoritmo es el hecho que el desplazamiento máximo alcanzable estará limitado por la cadena de menor longitud dentro del grupo, aunque en la práctica es uno de los

algoritmos que mejor desempeño presenta (Navarro y Raffinot, Cap. 4, pp. 74-76 [33]). WM es $O\left(B\frac{n}{m}\right)$ en la fase de búsqueda donde B es el tamaño de los bloques de caracteres.

3.4. Algoritmos basados en autómatas

Un autómata finito o máquina de estado finito es un modelo matemático de un sistema que recibe una cadena constituida por símbolos de un alfabeto y determina si esa cadena pertenece al lenguaje que el autómata reconoce. Al decir "lenguaje que el autómata reconoce", se refiere a que el autómata concede como resultado final la aseveración de que ha sucedido a su entrada una secuencia de símbolos que el autómata es capaz de reconocer. A continuación ofrecemos algunas definiciones formales en relación con los autómatas.

- **Lenguaje del autómata:** Sea Σ^*, el conjunto de todas las cadenas de cualquier tamaño conformadas con los caracteres del alfabeto Σ, $\zeta \subseteq \Sigma^*$. O sea ζ es el subconjunto de cadenas sobre Σ que el autómata detecta.
- **Autómata finito:** Se conoce por autómata finito a un conjunto de finito de estados Q de entre los cuales uno es inicial ($estadoI \in Q$) y algunos otros son finales o terminales, (grupo de estados $F \subseteq Q$). La transición entre estados está etiquetada o definida por elementos de Σ (el alfabeto), en nuestro contexto serán caracteres. Los que están formalmente definidos por una función de transición D, la cual asocia para cada estado ($q \in Q$) un grupo de estados $q_1, q_2 ... q_i$ de estados de Q por cada $\alpha \in \Sigma$ (caracter del alfabeto). Un autómata es entonces definido por la quíntupla $A = (Q, \Sigma, I, F, D)$.
- **Autómata finito no determinístico:** *(NFA, Non-deterministic Finite Automaton)*. Si la función de transición D es tal que existe un estado q asociado mediante un caracter α dado a más de un estado, digamos $D(q, \alpha) = q_1, q_2, \ldots, q_k$ $k > 1$, o hay alguna transición denotada por ε. Entonces D se denota por un arreglo de tripletes $\Delta = (q, \alpha, q'), q \in Q, \alpha \in \Sigma \cup \varepsilon, q' \in D(q, \alpha)$.
- **Autómata finito determinístico:** *(DFA, Deterministic Finite Automaton)*. Si la función de transición D es denotada por la función parcial $\delta : Q \times \Sigma \to Q$, tal que si $D(q, \alpha) = q'$, entonces $\delta(q, \alpha) = q'$

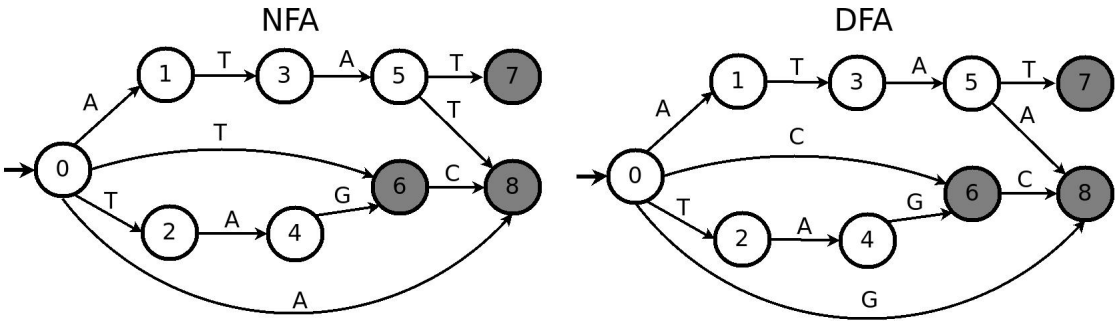


Fig. 2. Autómatas no determinístico NFA y determinístico DFA, el estado 0 es el estado inicial, los estados sombreados son los estados finales [33].

En la figura 2 se muestran ambos autómatas, en el caso de los NFA un caracter puede activar a varios próximos estados, por tanto pueden existir varios estados activos al mismo tiempo, además existen transiciones donde el caracter es irrelevante. En los DFA solo puede haber un estado activo, puesto que un caracter conduce sólo a un próximo estado. La transicion mediante el caracter vacío ε, en lo NFA se utiliza

mayormente para simplificar el NFA aunque siempre existe un autómata equivalente que no requiere de la transición con el caracter ε para reconocer el mismo lenguaje.

En los DFA un caracter activa solo un próximo estado, mientras que en los NFA un mismo caracter puede activar varios. Debido a que se necesita un caracter para la próxima transición y que sólo un estado puede activarse, los DFA pueden llegar a ser $O(n)$, pero consumen un espacio en memoria $O(|\Sigma| \times 2^m)$,en el caso de los NFA por cada caracter deben activarse varios estados siendo para el peor caso $O(m \times n)$ y un espacio $O(m)$.

3.4.1. Aho-Corasick (AC)

Aho-Corasick presentado por Alfred V. Aho y Margaret J. Corassick en 1975[3] puede verse como la extensión de KMP a la búsqueda de múltiples cadenas, el algoritmo utiliza un autómata especial llamado Autómata Aho-Corasick, dicho autómata se construye a partir de una estructura arbórea creada a partir del grupo cadenas P. En este, los nodos constituyen los estados del autómata y los caracteres, la condición para la transición entre los estados, según Navarro y Raffinot [33] una estructura arbórea sobre conjunto de cadenas es denominada *trie*, nosotros adoptaremos ese término a lo largo del texto. Lo que hace especial a

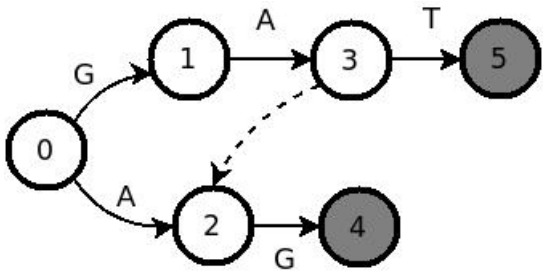


Fig. 3. Autómata AC que reconoce las cadenas "GAT" y "CG", una transición suplementaria se muestra con líneas discontinuas.

este autómata es la existencia de una función auxiliar o función suplementaria *Sac*, esta función se utiliza cuando en el estado actual no existe transición al próximo estado mediante el caracter leído del texto.

Formalmente, sea q_i un estado del *trie* construido sobre P, $L(q_i)$ es la trayectoria recorrida hasta el estado q_i que esta definida por la secuencias de caracteres que condicionan las transiciones desde q_0 (estado inicial) hasta q_i, por ejemplo en la figura 3 $L(q_3)$="GA". Entonces, la función *Sac* se define como, $Sac(q_i) = q_j$, tal que el sufijo más largo de $L(q_i)$ sea igual a $L(q_j)$ donde $L(q_j)$ es prefijo de alguna otra cadena $p^r \in P$. En la figura 3 por ejemplo, tenemos que $Sac(q_3) = q_2$, porque "GA" tiene como sufijo a la cadena "A", siendo esta prefijo de la cadena "AG".

El número de comparaciones es $O(n + nocc)$ si las transiciones se implementan utilizando tablas de acceso directo y $O(nlog|\Sigma| + nocc)$ si se hace a través de listas enlazadas, *nocc* es la cantidad de ocurrencias detectadas. AC se puede mejorar si se adicionan las transiciones de la función suplementaria a las transiciones de los estados. Esto es, $\delta(q_i, \sigma) = \delta(Sac(q_i), \sigma)$ donde $\delta(q_i, \sigma)$ es la función de transición de los estados del autómata.

Lo anterior implica que para el peor caso se obtendrán $|\Sigma|$ transiciones por cada estado, aumentando considerablemente el tamaño del autómata, con una complejidad de $O(|P| \times |\Sigma|)$ espacio. El elevado consumo de memoria es una de las principales desventajas de este autómata lo que lo hace poco viable cuando se tiene un elevado número de cadenas.

Como mayor ventaja se tiene que es un algoritmo capaz de reconocer múltiples cadenas de forma concurrente y en un orden de tiempo lineal. Esto último ha valido como para que varios investigadores hayan implementado AC en hardware con muy buenos resultados.

3.4.2. Expresiones regulares (RegExp)

Las expresiones regulares son cadenas que expresan de forma declarativa un conjunto de cadenas que aceptan, mediante la representación de las posibles relaciones que puedan existir entre los caracteres o expresiones regulares que componen dicho conjunto de cadenas, Hopcroft, 2001 [23].

Mediante esta forma, se pueden agrupar varias cadenas en una sola expresión regular, representando las relaciones entre sus caracteres mediante el uso de caracteres especiales llamados *metacaracteres* $(*, |, ?)$, los cuales poseen un significado exclusivo para establecer diferentes relaciones entre los caracteres normales o *literales*. Por ejemplo, la expresión regular $(ab|bc)*$ indica a seguido de b, ó b seguido de c cero, o más veces, en este caso la condición OR se representa mediante el metacaracter "|", "$*$" indica cero o más veces, y dos caracteres seguidos indica concatenación.

Nótese que una expresión regular que sólo posee caracteres literales y concatenación contiene una sola cadena, esto es una cadena estática o simplemente cadena, tal como se definió en la sección 2. La forma de reconocer cadenas mediante el uso de expresiones regulares se realiza construyendo un autómata que acepte su lenguaje, este puede ser DFA o NFA, en cuyo caso se nombran máquinas de texto directo y máquinas de expresiones regulares directa.

Las expresiones regulares son ampliamente utilizadas dado la capacidad de compactar varias cadenas en una misma expresión, lo que hace un uso más eficiente del medio sobre el que se implemente, al requerir menos recursos para la máquina de estados que las describa, ya sea software o hardware, esto hace que las expresiones regulares sean una solución viable cuando se necesita encontrar ocurrencias de un número elevado de cadenas con un consumo de recursos eficiente.

3.5. Algoritmos con búsqueda por factor

Teóricamente una sola expresión regular de longitud r puede ser expresada en un NFA con $O(r)$ estados, mientras que si el NFA es convertido a DFA puede generar $O(2^r)$ estados. Respecto al procesamiento, cada caracter a la entrada el DFA, ha de verificarse contra la tabla de transición de un único estado activo, para activar el próximo estado, tomando $O(n)$ tiempo si la implementación se realiza mediante tablas de acceso directo, y $O(|\Sigma|)$ si se implementa con listas enlazadas. El NFA por su parte toma $O(n \times r^2)$ en el

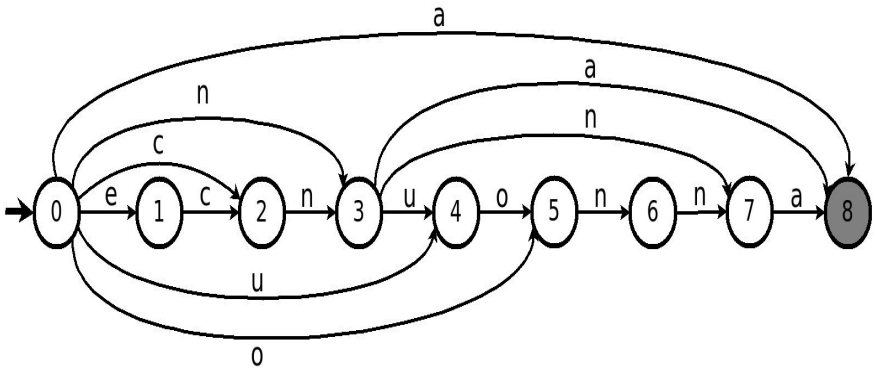


Fig. 4. Autómata Factor Oracle, la palabra queda descrita por el autómata de forma inversa [33].

peor caso, para un caracter de entrada se verifican las transiciones de todos los r posibles estados activos, hacia todos r posibles próximos estados. Téngase en cuenta que en un GPP este proceso de actualización ha de ser secuencial lo que aumenta el tiempo de ejecución. En resumen los DFA de expresiones regulares

consumen más memoria que los NFA, aunque son más rápidos, los NFA por el contrario consumen mucho menos memoria pero su implementación conlleva a un procesamiento secuencial impuesto por plataforma donde son implementados (GPP) que los hace mucho más lentos.

Los algoritmos vistos hasta ahora desarrollan la lectura del texto en forma de prefijo o sufijo existe una familia de no menos importantes algoritmos que realizan la búsqueda por factor. Suponga que se ha leído hacia atrás un factor u del texto que es también factor de la cadena p y al leer el próximo caracter α, este no coincide, entonces podemos decir que la cadena αu no es ningún factor posible de la cadena buscada, y podemos por lo tanto desplazar la ventana de búsqueda de manera segura después del caracter donde ha errado la búsqueda. La principal dificultad que se presenta bajo este paradigma es que se hace necesario reconocer el grupo de factores de la cadena.

3.5.1. Backward-Dawg-Matching (BDM)

Backward-Dawg-Matching (BDM) presentado por Crochemore y otros autores en [18] utiliza un autómata de sufijo, o sea un autómata que describe la cadena de manera inversa, para reconocer cuando una secuencia de caracteres dada es factor de la cadena. La forma en que se implementa este autómata nos conduce a dos variaciones del mismo algoritmo, *Backward-Nondeterministic-Dawg-Matching (BNDM)* y *Backward-Oracle-Matching (BOM)*[26]. Independientemente de cuál de los dos sea, el autómata para la búsqueda por factor debe tener las siguientes propiedades:

1. Determinar cuando una cadena u es factor de la cadena p en un tiempo $O(|u|)$. Una cadena u es factor de la cadena p sí y sólo sí existe $L(q_i) = u$.
2. Reconocer los sufijos de la cadena sobre la que se construyó el autómata dado que si se llega a un estado final, la etiqueta de la trayectoria es un sufijo de la cadena.
3. Sea posible construir sobre una cadena $p = p_1, p_2, \ldots, p_m$ en un tiempo $O(m)$, y en Tiempo Real, esto quiere decir que se pueden adicionar caracteres p_j, uno tras otro dentro de la estructura y actualizar a cada paso el autómata de sufijo del prefijo $p_1, p_2, \ldots, p_{j-1}$ hasta obtener $p_1, p_2, \ldots, p_j$.

BDNM, utiliza la misma heurística que BDM pero el autómata es implementado utilizando paralelismo de bits. La técnica de Paralelismo de Bits no es más que utilizar la concurrencia en las operaciones sobre una palabra de computadora, o registros del procesador (Navarro y Raffinot [33]). Lo anterior hace al algoritmo viable sólo cuando la longitud de la cadena es pequeña (lo suficiente como para ser representada en un sólo registro del procesador). Cuando la longitud de la cadena es grande no es posible almacenar la palabra en un sólo registro y por tanto no se pueden realizar las operaciones de manera concurrente.

En la figura 4, se muestra un ejemplo del autómata acíclico determinístico *Factor Oracle* para la palabra *announce*, nótese que la disposición de los estados en sentido inverso al de la palabra y que tomando cualquier camino desde el nodo inicial hasta el final se obtienen trayectorias etiquetadas con factores de la palabra, por ejemplo si seguimos la trayectoria nodos, $0, 3, 8$ habremos reconocido el factor *an*.

El algoritmo implementado de esta manera se conoce como Backward-Oracle-Matching (BOM). Con el mismo paradigma de búsqueda que BDM, en el que se lee hacia atrás en la ventana, mediante un autómata que describe de forma inversa la cadena. Si se falla en una letra después de haber leído u entonces se conoce que σu no es un factor de p y se puede desplazar de manera segura la ventana después de σ. Si el principio de la ventana es alcanzado, dado que Factor Oracle reconoce una única cadena de tamaño $|p|$ se dice que ha habido una ocurrencia. Para el peor caso BOM es $O(m \times n)$.

La extensión natural de este algoritmo al reconocimiento múltiple de cadenas se logra con el algoritmo *Set-Backward-Oracle-Matching (SBOM)* [4] donde el autómata se construye en forma de trie sobre P (P, conjunto de cadenas, ver sección 2), de manera inversa, y se agrega al igual que en AC una función

suplementaria que asocia a cada estado un estado suplementario, dicho estado suplementario se selecciona bajo el criterio de ser una trayectoria igual a algún factor de cualquier otra cadena en P o de ella misma, en este caso SBOM es $O(n \times |P|)$.

3.6. Conclusiones sobre los algoritmos

Navarro y Raffinot, Cap. 4, pp. 74-76 [33] muestran de forma experimental los algoritmos más eficientes en la práctica, los más aventajados son, WM, SBOM y AC. Es importante hacer notar la diferencia entre el procesamiento de un texto estático y el de un flujo de datos, como hemos dicho anteriormente en un flujo de datos no se conoce de antemano la longitud del texto y la información es sensible a la velocidad con que se procese, algoritmos como KMP, WM, SBOM que realizan más de un procesamiento sobre un caracter requieren de la detención del flujo de datos en tanto no se termine el procesamiento antes de admitir otro símbolo. Aún logrando garantizar un desplazamiento grande de la ventana de búsqueda este desplazamiento no es fijo, y está sujeto a las características de los datos de entradas pudiéndose reducir al máximo para el peor caso.

Al operar sobre un flujo a alta velocidad, algoritmos con un tiempo por encima de $O(n)$ aunque presenten características sublineales, se ven superados en la práctica, por las velocidades en los flujos de datos de las redes informáticas actuales, viéndose obligados a dejar de procesar información en pos de no atentar contra la eficacia y velocidad de la red. Incluso $O(n)$ no es suficiente, en realidad la tecnología subyacente actual, no ofrece la velocidad de operación suficiente como para superar las velocidades de las redes procesando un caracter por ciclo, en el estudio realizado por Fang Yu y otros autores en [49], L7-filter, una aplicación para la detección de intrusos, decrementa su capacidad de procesamiento a $10Mbps$ cuando se realiza el máximo de inspección, Snort con 500 cadenas sólo puede admitir $50Mbps$ en un Pentium 3 Dual a $1Ghz$ [15], es visible la reducida velocidad frente a la alcanzada por el flujo de *1Gb-Ethernet*, o *10Gb-Ethernet*, estándares actuales de red.

4. Potencialidades de los FPGAs

Los dispositivos lógicos programables (PLD), forman parte de los circuitos de alto nivel de integración de lógica programable que incluye, entre otros, los PLD simples (SPLD), los PLD complejos (CPLD) y los arreglos de compuertas programables por campo, *Field Programable Gate Array, FPGA*.

De los dispositivos mencionados anteriormente los FPGA son los que más capacidad presentan para diseñar sistemas grandes, estos están constituidos, por un arreglo bidimensional de bloques programables, CLB por sus siglas en inglés, en los que es posible implementar funciones lógicas, tanto combinacionales como secuenciales, con un número pequeño de entradas y salidas. Además, constan de una red de interconexión programable que posibilita la conexión entre todos los CLBs, de esta manera se conforman funciones mayores y más complejas. Existen, también, bloques de entrada y salida programables que brindan comunicación con el exterior y otros elementos embebidos destinados a optimizar el funcionamiento del dispositivo o a proveer funciones específicas, por ejemplo, memorias, multiplicadores, gestores de reloj y hasta microprocesadores en algunas familias de dispositivos (Virtex-5 FPGA User Guide, 2009 [47]).

La configuración del FPGA se lleva a cabo por el usuario final. En dependencia de la tecnología de interconexión, este podrá ser *reprogramable*, con el uso de SRAM, *o no reprogramable*, con *anti-fusibles*. Existen dos categorías básicas de FPGA en el mercado actual, la primera basada en tecnología SRAM y la segunda basada en anti-fusibles. Los productores líderes de la primera categorías son *Xilinx y Altera*. En la segunda se encuentran Actel, Quicklogic, Cypress, y Xilinx con algunos productos.

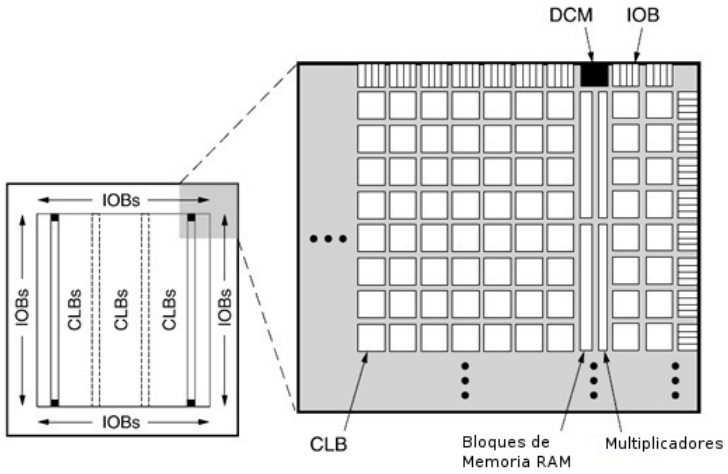


Fig. 5. Arquitectura general de un FPGA, se muestra el arreglo de los diferentes tipos de bloques lógicos sobre la oblea de silicio [45].

4.1. Arquitectura general de los FPGAs

La forma en que están dispuestos espacialmente los bloques sobre la oblea de silicio queda representada figura 5. Un anillo de bloques de entrada salida, (I/O-CLBs), rodea el arreglo regular de CLBs. La cantidad de columnas de memoria RAM embebidas en el arreglo varía entre los distintos dispositivos de las familias, que van desde una hasta cuatro columnas en los dispositivos con más densidad de lógica, cada columna contiene varios bloques de $18Kbits$, y cada bloque está asociado a un multiplicador dedicado. Los gestores de reloj (DCM, por sus siglas en inglés), proporcionan señales de reloj, soportan auto-calibración, y ofrecen soluciones totalmente digitales para la distribución, retardo, multiplicación, división, y corrimiento de fase de las señales de reloj, estos se encuentran en cada extremo de la columna y distribuidos según su cantidad.

4.1.1. Arquitectura interna de los bloques programables

Cada bloque está dividido internamente en cuatro subestructuras o subbloques estrechamente interconectados entre sí denominados *Slices*. Dos de las grandes familias de dispositivos más difundidas en el mercado actual son Spartan y Virtex, ambas son productos de Xilinx, tomamos por referencia a la familia de dispositivos *Spartan-3* de Xilinx, la diferencia más remarcada en cuanto a arquitectura de los bloques lógicos de Virtex-5 con los de Spartan-3 consisten en que cada CLB de Virtex contiene 2 *Slices* y cada una 4 células o celdas lógicas de 6 entradas cada una.

Como se muestra en la figura 6, existen dos tipos de *Slices*, denominadas *Slices* de la parte izquierda, y *Slices* de la parte derecha. Las Slices de la izquierda, poseen capacidades adicionales, a diferencia de su contra parte, estas soportan funcionalidades de: registro de desplazamiento, y RAM distribuidas, dos aspectos a los que haremos referencia posteriormente cuando analicemos su estructura interna.

4.1.2. Arquitectura interna de las slices

Las *Slices* se componen de los siguientes elementos, dos generadores de funciones lógicas, dos elementos de almacenamiento que pueden funcionar como *flip-flop* o *latch*, multiplexores de funciones amplias, lógica de acarreo de bits (utilizado para funciones aritméticas) y compuertas aritméticas (Spartan-3 FPGA

Family Complete Data Sheet [45]), ver figura 7. Los principales elementos para implementar funciones

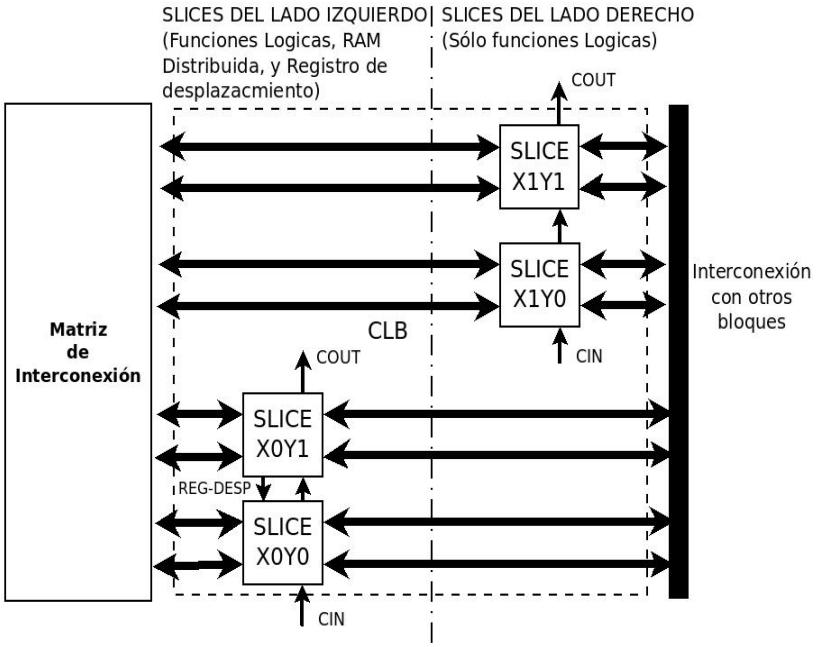


Fig. 6. Arquitectura interna de un CLB de la famila Spartan-3, cada Slice recibe dos entradas de 4 bits cada una, CIN y COUT son las entradas y salidas de la lógica dedicada al acarreo de bits [45].

lógicas son los generadores de funciones o *look-up tables, LUT*, por sus siglas en inglés, estos son memorias RAM de 16×1 bits, con un bus de direcciones de cuatro bits que consistirá en la entrada de la función lógica que se desee implementar. Esta implementación se lleva a cabo almacenando en la memoria la tabla de la verdad de dicha función, por lo que, en una LUT se puede implementar cualquier función lógica de 4 entradas y una salida.

En la figura 7, se pueden apreciar dos partes de la *Slice* muy similares, parte superior y parte inferior, estas constituyen las celdas lógicas, cuyos pares de LUT+*flip-flop* son G-LUT+FFX y F-LUT+FFY respectivamente. En cada celda se implementan funciones lógicas de cuatro entradas como máximo. Estas a su vez se pueden combinar a través de los multiplexores de funciones amplias F5MUX y FiMUX para lograr funciones de más de cuatro entradas.

Como se mencionó anteriormente las *Slices* de la parte izquierda permiten la implementación de registros de desplazamiento de hasta 16 bits, el tamaño de estos es programable. También poseen funcionalidades de RAM distribuida, estas capacidades se llevan a cabo en las LUTs.

Como registro de desplazamiento, en la figura 9, el bit de entrada será por DI, el que se desplazará una posición en cada pulso de entrada, la salida por D será en dependencia de a qué dirección se apunte en A[4:1] configurando así el tamaño deseado del registro, la salida MSC15 esta puesta al desplazamiento máximo que es de 16 bits, esta sirve para concatenar con otra LUT de manera que se puedan lograr registros de desplazamiento de más grandes.

Como RAM distribuida se utilizan las LUT para almacenar datos, teniendo un espacio de memoria de $16x1$ bits con un bus de 4 bits de dirección y asociándolas debidamente se pueden lograr diferentes capacidades de memoria RAM. La capacidad máxima que se pude alcanzar está limitada por la cantidad de LUT disponibles en cada dispositivo. En los manuales de usuario de cada dispositivo de la familia se ofrecen las capacidades máximas de memoria RAM distribuidas que es posible implementar en cada caso.

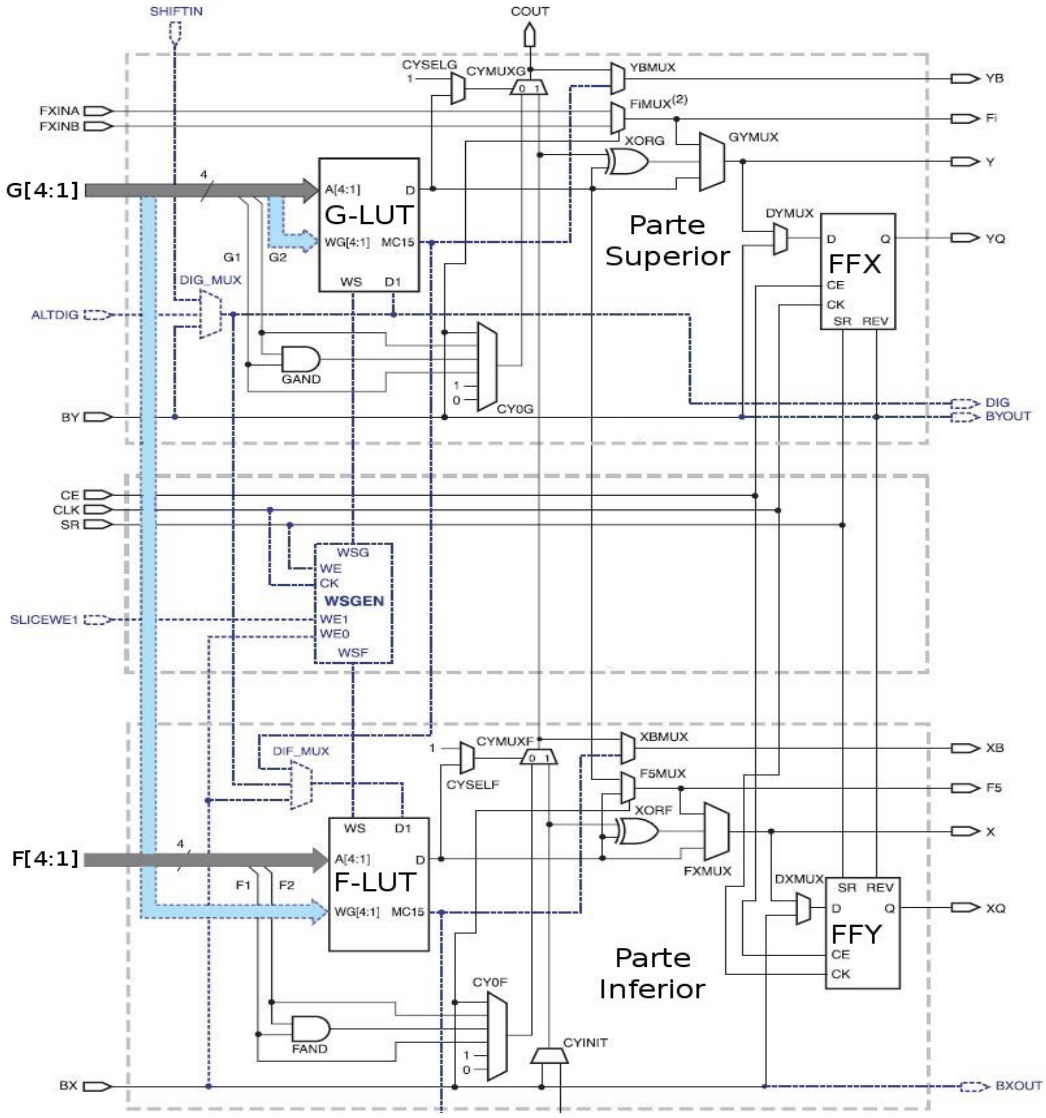


Fig. 7. Arquitectura interna de un Slice de Spartan-3, G y F, son las entradas de 4 bits a cada LUT correspondiente, la salida se obtiene de el puerto de salida de cada flip-flop o directamente [45].

4.2. Capacidad de lógica reconfigurable y prestaciones

Para tener una idea de los recursos disponibles en los dispositivos presentamos algunos datos las familias de dispositivos más utilizadas:

- **Spartan-3:** Posee con capacidad de lógica que van desde 50 mil hasta 5 millones de compuertas lógicas equivalentes, desde 192 hasta 8,320 bloques programables y entre 1,728 y 74,880 células lógicas equivalentes [45]. Diseñados para aplicaciones que requieran grandes volúmenes de lógica, contienen hasta 784 pines de entrada y salida, pueden soportar 622 Mbps de trasferencia de datos por circuito de entrada-salida. Presentan cinco elementos programables fundamentales, bloques lógicos configurables (CLB), bloques de entrada/salida (IOB), bloques de memoria RAM, multiplicadores dedicados, y controladores de reloj digital (DCM).

- **Virtex-5:** Ideados para altos niveles de ejecución y elevadas velocidades, con una frecuencia de trabajo máxima de 550 Mhz, gestores de reloj DCM de nueva generación, bloques de memoria RAM/FIFO embebidas en bloques de 36 Kbits, además algunos modelos traen hasta 2 microprocesadores *PowerPc 440 RISC*. Los dispositivos de Virtex-5 superan a Spartan-3 además en el volumen de lógica reconfigurable, de 4,800 CLBs hasta 51,840, donde cada CLBs esta compuesto por 2 Slices, las que a su vez contienen 4 células lógicas con LUTs de 6 entradas, un volumen de células lógicas (LUTs + latch) desde 12,480 hasta 207,360 [47].

Estas no constituyen la última generación de dispositivos porque en el momento de terminación de este reporte técnico ya se anunciaba la salida de la próxima generación de dispositivos, Virtex-6 y Spartan-4.

4.3. FPGA vs GPP

Existe una infinidad de algoritmos desarrollados en software para ejecutarse sobre procesadores de propósitos generales *(GPP, General Purpose Processors)*, estos procesadores siguen el paradigma del computador de Von-Neummann, ver figura 8. Bajo este paradigma cualquier algoritmo puede ser implementado en una estructura hardware fija, compuesta de tres partes fundamentales, memoria, donde se almacenan instrucciones y los datos, unidad de control, donde se decide la dirección de la próxima instrucción a ser ejecutada, y unidad aritmética y lógica donde la instrucción es ejecutada, las instrucciones son leídas y ejecutadas una tras otra (Bobda, 2007[8]).

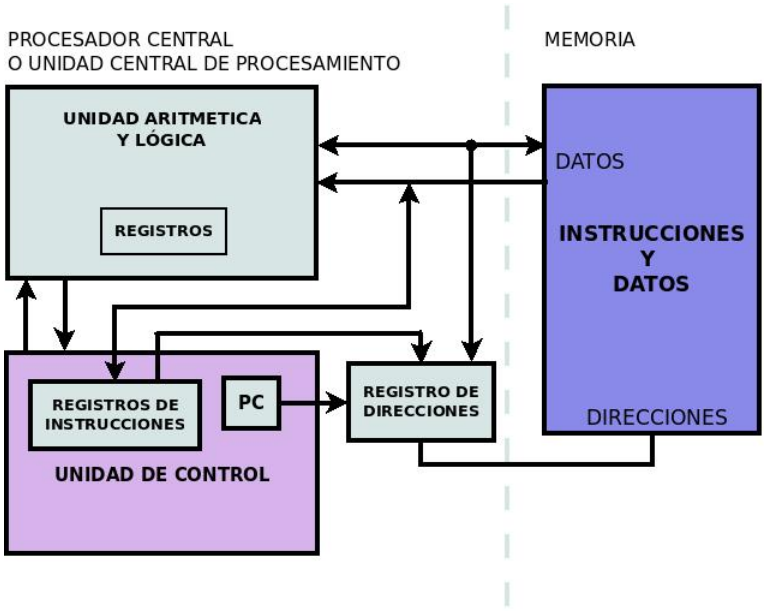


Fig. 8. Computador de Von-Neumman, se compone de tres partes fundamentales, Memoria, Unidad de Control, Unidad Aritmética Lógica.

Esta arquitectura brinda una gran flexibilidad dado que es posible implementar sobre ella casi cualquier algoritmo que siga las reglas del paradigma de Von-Neummann, en este sentido se dice que, el algoritmo debe adaptarse por sí mismo al hardware.

La secuencialidad entre la ejecución de instrucciones aún no siendo dependiente una del resultado de otra, es lo que hace que se vea disminuida la velocidad de procesamiento total. No es posible ejecutar varias instrucciones de manera bajo el paradigma de Von-Neummann.

Si bien es cierto que los procesadores modernos utilizan el concepto de *pipeline*, que no es más que la utilización varias líneas de procesos encadenados donde las instrucciones son procesadas de forma más o menos concurrente (la ejecución real de la instrucciones no ocurre al mismo tiempo), el número de tales *pipelines* es generalmente menor que la cantidad de instrucciones que pueden ejecutarse en paralelo. Visto de otra forma, no explotan al máximo el paralelismo intrínseco de la solución del problema.

Los requerimientos actuales de velocidad de procesamiento y volumen de cadenas en muchas aplicaciones encargadas de ejecutar reconocimiento de cadenas son superiores a los alcanzados por cualquier implementación de algoritmos software, implementado sobre GPP incluso para los más eficientes.

Otro tipo de procesadores son los Procesadores de Aplicaciones Especificas, (ASIP, por sus siglas en inglés), en estos las instrucciones son implementadas directamente en hardware, o sea, el hardware se adapta a la aplicación, de forma tal que, funciones que no son dependientes pueden ser ejecutadas concurrentemente alcanzando un paralelismo real, reduciendo los tiempo de ejecución. Los procesadores de aplicaciones específicas se pueden implementar en un solo chip y son generalmente nombrados ASIC, *(Application-Speciffic Integrated Circuits)*.

La desventaja de los ASIC, es que una vez creados solamente responden a una aplicación en específico, no se pueden reprogramar, ni crear nuevas funciones o actualizar su arquitectura, esto los hace inflexibles.

Los FPGA son reconfigurables ya que se puede modificar parcial o totalmente su arquitectura interna, por lo tanto decimos que son *adaptables a la aplicación*, ofreciendo grandes posibilidades para cumplir con los requerimientos de velocidad y flexibilidad, logrando las ventajas del paradigma de Von-Neummann, y el nivel de ejecución de los ASICs. Siempre que no exista dependencia entre la salida ó resultado de un proceso y la ejecución de otro, y en la medida que los recursos disponibles de hardware lo permitan, se puede disponer de estos recursos para ejecutar estos procesos concurrentemente disminuyendo el tiempo de ejecución total, esto se denomina paralelismo.

4.4. Capacidad de procesamiento

En el contexto del reconocimiento de cadenas en flujos de datos la Capacidad de Procesamiento es la cantidad de bits de información que es capaz procesar un algoritmo (implementado en Software o sobre una arquitectura de Hardware), por unidad de tiempo, de un flujo de datos de entrada.

Dicho flujo de datos representa un texto en el cual tendremos que hallar la ocurrencia de un conjunto de cadenas, por tanto, por procesar entenderemos las acciones que realiza determinada arquitectura sobre dicha información en un intervalo de tiempo fijo, con el objetivo de resolver el problema del reconocimiento de cadenas. Por ejemplo, si tenemos un autómata que en cada evento de reloj de duración fija lee un caracter y dada la función de transición se genera el próximo o los próximos estados, decimos que el autómata tiene una capacidad de procesamiento de 1 caracter (byte) por ciclo, si este ciclo dura la quinta parte de un segundo podemos afirmar que el autómata procesa 5 $bytes$ por segundo.

Hemos dicho que los flujos de datos en las redes informáticas actuales superan los $10Gbps$, los dispositivos FPGAs poseen frecuencias máximas de operación de hasta $500MHz$, dichas frecuencias de operación se ven afectadas a medida que la complejidad y el tamaño de los diseños crece. Típicamente los diseños para el reconocimiento de cadenas logran frecuencias de operación entre $45MHz$ y $250MHz$, con una frecuencia de operación de $100MHz$, procesando 1 caracter por cada ciclo de reloj, o sea $8bits$, se obtiene una capacidad de procesamiento de $0.8Gbps$, esta velocidad de procesamiento se haya por debajo de las velocidades alcanzadas por los flujos de datos actuales ($1Gbps - 10Gbps$).

La solución más acertada y aceptada a esta situación es procesar más de un caracter por ciclo de reloj. Lo anterior impone otro reto algorítmico puesto que en este sentido, se introduce otra variable al problema. Se trata del desalineamiento, con el objetivo de no perder ninguna ocurrencia se debe tomar en cuenta la

posición aleatoria en que se puede situar el comienzo de la cadena dentro del bloque de datos que se procese por cada ciclo de reloj. En muchos casos lidiar con este desalineamiento implica replicar el área tantas veces como caracteres se procesen de una vez.

Otras propuestas sacan ventaja de las características de la conmutación de paquetes, distribuyendo los paquetes en las colas de los dispositivos de enrrutamiento entre varias arquitecturas replicadas a modo de *pipelines*. De esta manera se logra procesar en paralelo múltiples caracteres por ciclo, pero con un costo de área elevado e ineficiente, ya que debido al tamaño variable de los paquetes existirán *pipelines* inactivos habiendo datos que procesar.

4.5. Utilización del área programable

En el contexto del reconocimiento de cadenas mediante lógica reconfigurable, dado que la entrada a nuestro problema consiste en cadenas de m caracteres, se puede dar el costo o consumo de una determinada arquitectura según la cantidad de CLB, slices, o células lógicas que consume por caracter, cadena, o grupo de cadenas buscadas.

Generalmente, dado que la estructura más pequeña que puede implementar una función lógica es la célula lógica, utilizamos *células lógicas por caracter*, aunque para tener una visión más global del costo en área configurable podemos utilizar las otras formas. Análogamente a la notación de complejidad de los algoritmos en software podemos decir que tal arquitectura toma, por ejemplo $O(m \times n)$ células lógicas o Slices.

Teniendo en cuenta que el número de cadenas es elevado y pronostica un continuo aumento, se impone un criterio de *economía de recursos programables*, dado que son agotables en mayor o menor medida según el dispositivo. Las investigaciones en este campo se proponen, encontrar soluciones que permitan reconocer la mayor cantidad de cadenas a una elevada velocidad, utilizando la menor cantidad de área programable, para esto, además del desarrollo de diseño de arquitecturas poco costosas, se recurren a una serie de optimizaciones que analizaremos más adelante.

Existe una relación inversa entre la frecuencia de operación y la complejidad de la arquitectura implementada. En la medida que existan más rutas de interconexión a través de los recursos de la matriz de enrrutamiento de los FPGA, de manera que una señal recorra trayectorias más largas entre la fuente y el destino, aumentará la *latencia*; esto es, el tiempo entre el surgimiento de un estimulo en la fuente y su establecimiento en el destino, téngase en cuenta que en un FPGA el intervalo entre pulsos de reloj para lograr el sincronismo de toda la arquitectura debe ser mayor, que la mayor demora o latencia que posea una trayectoria cualquiera en el circuito.

4.6. Prestaciones de las arquitecturas de los dispositivos

Existen ciertas prestaciones que nos ofrecen las arquitecturas de los dispositivos, en dependencia de la familia, o modelo, que son factibles de tener en cuenta a la hora de implementar un diseño, pues su uso consciente, conlleva a diseños relativamente económicos y que presentan elevadas capacidades de procesamiento.

En este punto el nivel de lógica involucrado juega un papel fundamental. Si por ejemplo, para implementar una función cualquiera utilizamos algunos o todos los recursos de un CLB, tendremos un primer nivel de lógica, (es importante destacar que dentro del mismo las conexiones son dedicadas y por lo tanto la latencia será mucho menor). Si en vez de esto, concatenamos dos de estos bloques tendremos entonces dos niveles de lógica, y así sucesivamente. Dado que la latencia en la red que interconecta los CLBs, es mayor

sus propias conexiones internas, esto provocará en última instancia, que se vea reducida la frecuencia de operación global del circuito.

- **LUT como registro de desplazamiento:** La capacidad de utilizar las LUTs de las Slices de la parte izquierda como registros de 16 bits, figura 9, y la posibilidad de concatenarlas para lograr rangos mayores, hace que el uso de esta forma de implementación e registros de desplazamientos redunde en un uso eficiente el área. Nótese que se pueden implementar desplazamientos de hasta 32 bits en un mismo nivel de lógica.

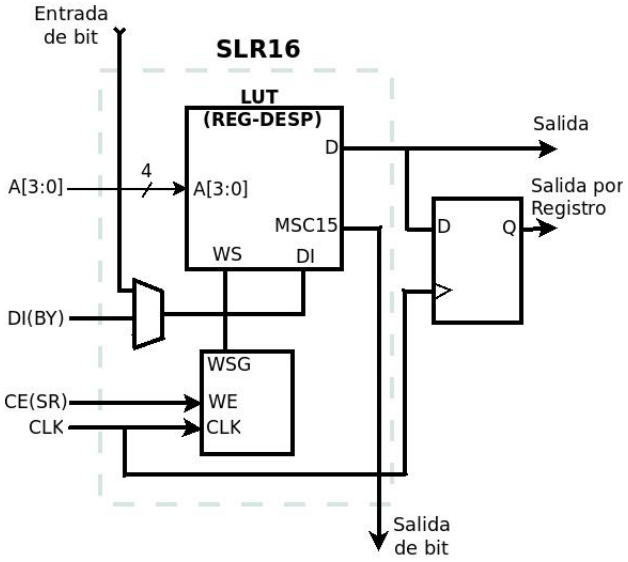


Fig. 9. LUT como registro de desplazamiento, el tamaño del registro se configura por la entrada estándar de las LUT, la salida es por D, MSC15 porporciona una salida para concatenar con otros registros [46].

Esta implementación se puede inferir por las herramientas de síntesis en dependencia de cómo esté descrito el diseño, o si hacemos uso de los módulos de propiedad intelectual de Xilinx, en el que el módulo de registro de desplazamiento. SRL16 explota esta capacidad, de otra manera siempre que implementemos registros sin entradas de *set* o *reset* sincrónicas o asincrónicas y no se hagan acceso a todos los bits al mismo tiempo se usará este tipo de registro de desplazamiento.

- **Uso de multiplexores dedicados:** Otra característica no menos interesante de las Sílices de estas familias de dispositivos es la existencia de multiplexores internos que permiten lograr funciones de más de cuatro entradas. De esta manera, la Generación Spartan-3 de Xilinx puede realizar multiplexores de 4:1 en un Slice, 8:1 en un CLB, 16:1 en dos CLBs. El uso de la sentencia CASE posibilita a las herramientas de síntesis infieran este recurso, las sentencias con IF THEN o ELSE sólo cuando sus condiciones son mutuamente excluyentes lo implementan, de otra manera se utiliza lógica extra para generar un árbol de prioridades.

5. Algoritmos paralelos para el reconocimiento de cadenas en FPGA

Debido a la capacidad de paralelismo y reconfiguración de los FPGAs y lo relativamente barato que resultan, mucho se ha investigado y desarrollado en busca de formas más eficientes de explotar sus recursos.

Dándose respuesta a los requerimientos (siempre crecientes) de velocidad de procesamiento y cantidad de cadenas que las aplicaciones deben reconocer en un flujo de datos.

Muchas de las propuestas hechas por los investigadores son implementaciones en hardware de algoritmos desarrollados para procesadores secuenciales que en la mayoría de los casos han obtenido considerables mejoras de eficiencia respecto a las implementaciones en software. Otras provienen de las propias características de los dispositivos explotando al máximo las particularidades de cada arquitectura.

Como ya se ha dicho, el entorno de la seguridad de las redes informáticas resulta particularmente exigente para el reconocimiento de cadenas en flujos de datos por lo que la mayoría de las investigaciones realizadas sobre el tema, apuntan a resolver los problemas de eficiencia computacional requerida por este tipo de aplicaciones y en especial los NIDS. Del estudio de estos trabajos, obtenemos un conjunto de soluciones que por el acertado uso de los recursos de hardware así como por lo que han representado en cuanto a aumento de la capacidad de procesamiento, son aplicados en muchos diseños actuales de elevada eficiencia constituyendo líneas o patrones de diseños sobre las que se realizan varias investigaciones.

Aún con lo mucho que se ha mejorado en la relación área-capacidad de procesamiento, que ha permitido alcanzar el orden de las decenas de Giga bits por segundo, en cuanto a velocidad y el reconocimiento de un volumen de miles de cadenas, esta sigue siendo desfavorable, puesto que siempre existe el caso de que, o el diseño no logra una velocidad de ejecución requerida por el flujo de datos, o que no soporta el volumen de cadenas requerido, conduciendo a elevar los costos al tener que replicar el hardware para superar estos requerimientos.

5.1. Paralelismo aportado por los FPGAs al reconocimiento de cadenas

El tipo de paralelismo requerido por el reconocimiento de cadenas es aquel que permita la ejecución de los procesos de comparación de cada cadena del conjunto, de forma independiente y concurrente, sobre un flujo de datos común. Puesto de otra manera, imaginemos que por cada cadena existe un módulo de hardware que la detecta, estos módulos de hardware pueden ser vistos como pequeños procesadores totalmente independientes, que reciben un flujo de datos común y se encargan de detectar la cadena que ellos reconocen. Desde el punto de vista topológico este es un esquema de, Múltiples Instrucciones, un Solo Dato, (*MISD, Múltiple Instruction, Single Data*).

Los elementos programables de hardware de los FPGAs permiten crear estos módulos de forma independiente, entonces la secuencialidad queda relegada a los diferentes módulos. Dado que en una cadena el orden de sus elementos es determinante, el paralelismo en estos módulos es alcanzable siempre y cuando se respete este orden. Por ejemplo, la cadena *abbbcd* se puede procesar leyendo *abb* y después *bcd*, o *ab,bb,cd*, en cualquier caso se introduce paralelismo al procesar varios caracteres a la vez, nótese que la cantidad de tiempo para leer la primera secuencia de tres caracteres a la vez, es menor que la segunda.

5.2. Autómatas en FPGA

Los autómatas son implementados en FPGA como Máquinas Estados Finitas *(FSM, Finite State Machine)*. Una FSM es la implementación de un circuito lógico, que acepta valores de entrada y en función de esos valores y del estado actual, se activa el próximo estado. Las FSM son de dos tipos: *Meely*, cuando la salida depende del estado actual y de la entrada, y *Moore* cuando solo depende del estado actual.

Existen dos esquemas básicos para implementar FSMs, y están estrechamente relacionados con el modo de codificar los estados, y el tipo de recurso utilizado para almacenarlos. El primer método utiliza

memorias para almacenar los estados que son codificados como números binarios, y una lógica elevada para generar los próximos estados a partir del dato de entrada y del estado actual ver figura 10.

El otro método, es codificando el estado con un solo bit que mediante compuertas se transmite entre flip-flops indicando el estado activo, de este modo existirá un flip-flop por cada estado, pero la lógica para la transición entre estados se reduce a una compuerta AND. Este esquema se le denomina *One-Hot-Encoding*, OHE, y son típicamente más rápidas que el esquema basado en memoria (Golson, 1994 [21]). Debido a que tanto las funciones de transición como los estado se implementan con elementos de hardware independientes (celdas lógicas), se logra un paralelismo en la activación de los estados, o sea $O(1)$. Para

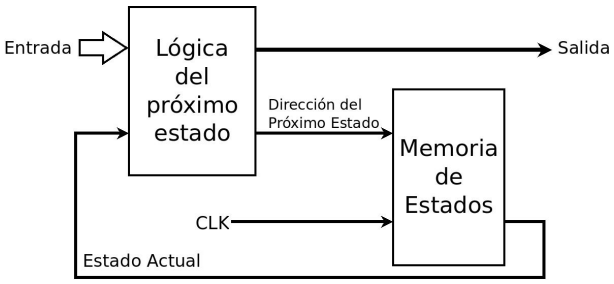


Fig. 10. Máquina Finita Estados, los estados son codificados como números binarios, la lógica del próximo estado se encarga de decodificar el estado actual y generar el próximo estado a partir de la entrada.

máquinas con un número elevado de estados, la naturaleza poco compacta de la codificación OHE a diferencia de las basadas en memoria, provoca que las distancias que deben recorrer las señales aumenten en correspondencia con el aumento del *fan-out* de los elementos lógicos. El término *fan-out* se utiliza para definir la cantidad de elementos lógicos se pueden conectar a la salida de otro elemento lógico. Esto tiene un fuerte impacto en el aumento de la latencia, haciendo que esta se incremente a medida que aumenta el *fan-out*. Por lo tanto las máquinas de estados que utilizan codificación OHE, tienden a introducir un aumento de la latencia a medida que el tamaño de estas se incrementa.

Por otra parte las FSM basadas en memoria, tienen a su favor que la actualización no es llevada a cabo mediante la reconfiguración de la lógica, sino mediante simples accesos a memorias siendo esta forma más rápida y flexible.

Los NFA alcanzan más de un estado por caracter de entrada, pero en máquinas secuenciales la activación de los próximos estados es, en mayor o menor medida, un proceso secuencial donde se activan uno a uno todos los próximos estados tomando un tiempo $O(m \times n)$. Implementado con lógica programable, esta activación puede ocurrir de manera paralela, o sea todos los próximos estados se activan en el mismo ciclo de reloj debido a que existen conexiones y elementos dedicados para cada transición, o sea tomando por caracter $O(1)$ tiempo. Por su parte los DFA generan estados $O(2^m)$ en teoría aunque desarrollos prácticos han demostrado que generan $O(m)$ con mayor frecuencia (Moscola y otros autores [32]).

Desde el punto de vista de diseño, la complejidad al implementar NFAs es mayor que con DFAs, dado que estos últimos son más simples ya que siempre el próximo estado estará relacionado con uno y solo un caracter, y no habiendo además transiciones debidas al caracter vacío ε. Ambas soluciones han sido implementadas ampliamente sobre lógica programable, a continuación proporcionamos un análisis de las propuestas más significativas.

5.2.1. Aho-Corassick

El autómata Aho-Corassick es un NFA que posee tiempo de procesamiento lineal con respecto al número de caracteres de entrada, esto implica que su tiempo de ejecución no es función de las características de los

datos en el flujo, cumpliendo con un requerimiento importante para los sistemas de detección de intrusos puesto que no es sensible a ataques que pretendan disminuir sus prestaciones.

Debido a esto se ha llevado a lógica reconfigurable de varias maneras [28][24] [29] [36], siendo el principal objetivo de los trabajos reducir el costo elevado del autómata Aho-Corassick. El esquema más utilizado para implementar FSM Aho-Corassick es el basado en memorias, esto se debe a que Aho-Corassick resuelve el problema que se presenta al fallar un caracter y tener que volver a comparar los caracteres ya procesados para no perder ninguna ocurrencia si estos son prefijos de alguna cadena del grupo (algo que en OHE se realiza de forma paralela), Aho-Corassick utiliza transiciones suplementarios para este caso.

En [24] se utiliza la técnica de *bit-spliting*, mediante la cual una máquina de estado Aho-Corassick es dividida en múltiples máquinas de estados que procesan solamente una porción específica de los bits de los caracteres de entrada. Los resultados parciales de las máquinas de estados resultantes son cotejados de manera que se obtenga el resultado final. Los autores agrupan las cadenas en módulos de detección donde subdividen el autómata Aho-Corassick en 4 máquinas de estados que procesan 2 bits de los caracteres de entrada cada una de forma independiente dando un resultado parcial sobre un vector en el que cada bit corresponde a una cadena reconocible por el módulo. Sobre estos vectores parciales se realiza una operación AND, obteniéndose un resultado con un valor lógico indicando la presencia de la cadena en el flujo.

Las máquinas de estado en este trabajo son implementadas utilizando los bloques de memoria embebidos en el FPGA, desafortunadamente los autores afirman que debido a que no es posible configurar estos bloques de manera que coincidan exactamente con el espacio de memoria generado por las máquinas de estados, se tiene un por ciento de inutilización de la memoria que va desde un 50 % hasta 82 % para módulos de 32 y 16 cadenas respectivamente. Aún con estos inconvenientes, con 47 módulos de 28 cadenas cada uno logran una capacidad de procesamiento de $1.6Gbps$ procesando un byte por ciclo a $200MHz$.

Lunteren en [29] define reglas para las transiciones entre los estados donde cada regla es una $n-tupla$ que contiene, el estado actual, el caracter de entrada, el próximo estado y la prioridad de la regla. La prioridad de cada regla es mayor para aquellas transiciones con el caracter correcto ver figura 3. Las reglas son luego almacenadas en memoria donde en cada entrada se agrupan las reglas. Para referenciar cada subgrupo de reglas se toma el índice generado por una función hash realizada sobre algunos bits específicos de los caracteres de entrada, luego se evalúan todas las posibles reglas en paralelo y se obtiene el próximo estado. En este trabajo se implementa el reconocimiento de 2000 cadenas extraídas de las reglas de Snort en sólo $128KB$ con una velocidad de $2Gbps$.

Wei Lin y Bin Liu [28] presentan una implementación en hardware del algoritmo Aho-Corassick basado en pipeline, al que nombran P2-AC. Este algoritmo agrupa los estados del autómata AC en niveles bajo el criterio de que en cada nivel estarán los estados que se encuentren a la misma distancia del estado inicial. Los niveles son almacenados en bloques de memoria independientes, y los caracteres de entrada son procesados por todos los niveles de forma paralela. De esta forma eliminan la necesidad de utilizar memoria para almacenar las transiciones suplementarias del AC transformándolo en un simple *trie* sin que este deje de ser $O(n)$.

Esta implementación de AC muestra magníficos resultados en cuanto a la reducción del costo de la arquitectura. Con un consumo de memoria menor del 47 % comparado con otras implementaciones de AC, es capaz de asumir la detección de hasta 80,000 caracteres. Aunque no se ofrece por parte de los autores datos en cuanto a la capacidad de procesamiento suponemos que al ser una arquitectura que procesa solo un caracter por ciclo y que hace uso exhaustivo de las memorias embebidas de los FPGAs, esta no deba ser muy elevada.

Benfano Soewito y otros autores proponen en [36], un método mediante el cual se obtienen todos los próximos estados posibles a partir de la propia codificación del estado actual, a este método lo denomi-

naron SAM-FSM *(del inglés, Self-Addresable Memory FSM)*. Basándose en la observación de que para cada estado es necesario almacenar 256 próximas transiciones en memoria, aún cuando algunas no se usan, en la arquitectura se modifica la lógica del direccionamiento de memoria al mas bajo nivel, de forma tal que al decodificar el estado actual se habilitan solamente aquellos espacios de memoria que pueden ser direccionados por los caracteres correspondientes a los próximos estados. Los resultados muestran que para 500 cadenas solo se consumen $34{,}36KB$, siendo este valor 63 veces menos de lo que consumiría un autómata tradicional. Creando pipeline con la misma arquitectura sobre el cual distribuyen los paquetes de internet logran una capacidad de procesamiento total de $4Gbps$.

Los algoritmos implementados en hardware del autómata Aho-Corassick, han estado encaminados especialmente a reducir el costo, sin embargo, la capacidad de procesamiento de las máquinas de estados basadas en memoria esta limitada por dos factores, primero el uso de memorias conduce a diseños mas lentos debido a la velocidad de acceso de estas, segundo su poca escalabilidad al procesamiento de múltiples caracteres por ciclo, debido al drástico aumento del consumo de memoria.

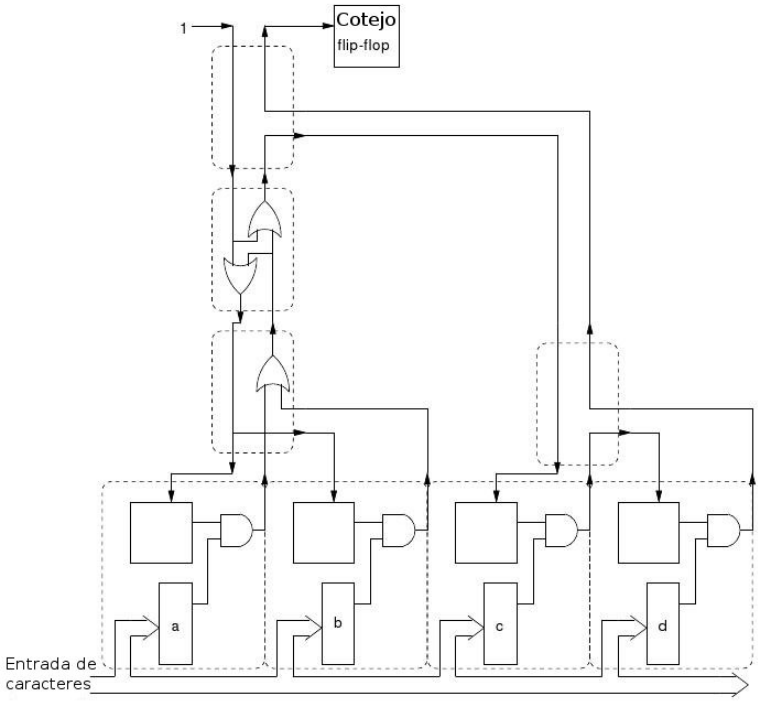


Fig. 11. Máquina de reconocimiento de la expresión regular, $((a|b)*)(cd)$ [35].

Por último es importante destacar que una máquina de estado con arquitectura arbórea o tipo *trie*, usando OHE, tiene el mismo procesamiento ($O(n)$) que una FSM Aho-Corassick basada en memoria, esto se debe al paralelismo que introduce el hecho de poseer hardware dedicado para cada estado. La principal diferencia consiste en que con esquema OHE el aumento de los recursos de enrrutamiento, la latencia y el fan-out son los factores críticos, mientras que con memorias son el elevado consumo y la velocidad de acceso de estas.

5.2.2. Expresiones regulares en FPGA

Las máquinas de estados de expresiones regulares clasifican entre los que mayor aceptación han tenido al ser implementados sobre FPGA [35] [11] [32] [10] [38] [27] [30] [48], la capacidad de compactar varias

cadenas en una sola expresión regular se traduce en elevada reducción del consumo de recursos para las máquinas de estados. La principal desventaja estriba, en que debido a que la transición entre estados se realiza procesando un caracter tras otro, lograr procesar k caracteres por ciclo conduce a soluciones $O(k \times m)$ en cuanto a consumo de recursos.

Uno de los trabajos más importantes en esta dirección es el realizado por Sidhu y Prasanna [35], según los autores, este trabajo es el primer uso práctico de las máquinas de estados no deterministas sobre FPGA. Con la intención de describir eficientemente en lógica reconfigurable expresiones regulares, se define un diseño de bloques básicos elementales a partir de los cuales se puede describir cualquier expresión regular en hardware. Sea r_1 expresión regular 1 y r_2 expresión regular 2, estos son correspondientes con los metacaracteres: $r_1|r_2$ (r_1 ó r_2), r_1* (cero o más r_1), r_1r_2 (r_1 seguido de r_2).

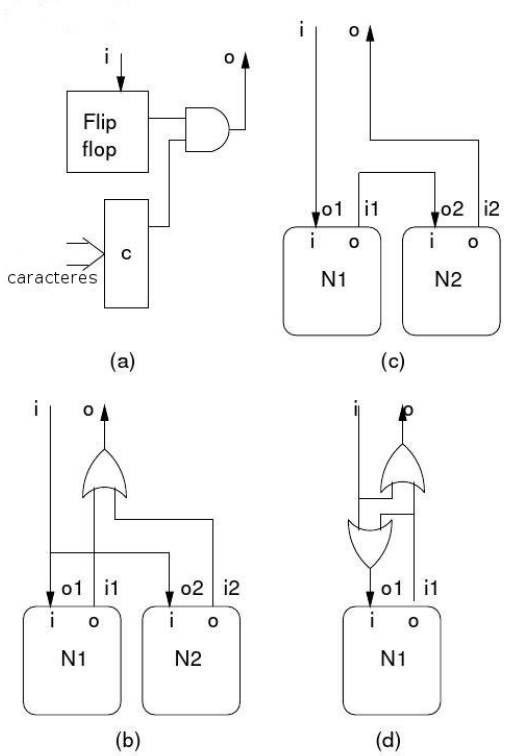


Fig. 12. Bloques básicos: (a) un solo caracter, (b) $r_1|r_2$, (c) r_1r2, (d) r_1* [35].

El esquema de implementación seguido es OHE, figura 12 (a), donde el bloque para procesar un caracter lo constituyen, un flip-flop para almacenar el estado activo, un comparador para detectar la presencia del caracter en el flujo de entrada, y a la salida de ambos una función lógica AND para implementar la función de transición, dichos bloque se muestran en la figura 11. Una *Máquina de Reconocimiento de Expresiones Regulares*, MRE, llamemos le así a la FSM de una expresión regular, que se construye conectando debidamente estos bloques.

Basados en estos bloques básicos en la propuesta de R. Franklin *et al.*[11] adicionan otro para el metacaracter "?" (cero o una vez), Sourdis y otros autores [38] incorporan $r\{N\}, r\{N,\}, r\{N,M\}$ (r repetido N veces, más de N veces, y entre N y M veces), además implementan algunas optimizaciones como compartir recursos, ver sección 6.1. En este trabajo alcanzan una capacidad de procesamiento de 2,9$Gbps$ consumiendo 1,28 celdas por caracteres.

Yi-Hua E. Yang y Viktor Prasanna [48] automatizan el proceso de transformación expresiones regulares a autómatas no determinísticos y luego a código VHDL, e introducen el uso de bloques de memoria para implementar las operaciones de tipo "clase de caracteres", por ejemplo $r_1[ab]$, o sea r_1 seguido de a o b. Implementan además una arquitectura capaz de procesar múltiples caracteres por ciclo, ver figura 19(b), obteniendo una capacidad de $10Gbps$ procesando 8 caracteres por ciclo con 267 expresiones regulares.

5.3. Shift-and-compare

La técnica de Desplazamiento y Comparación *(shift-and-compare)*, [5] [37], decodifica e introduce cada caracter en un pipeline correspondiente, de longitud igual a la posición que este ocupa en la cadena de izquierda a derecha, luego sobre salida de cada pipeline se realiza, una función AND, indicando la ocurrencia o no, de la cadena en el flujo de datos figura 13.

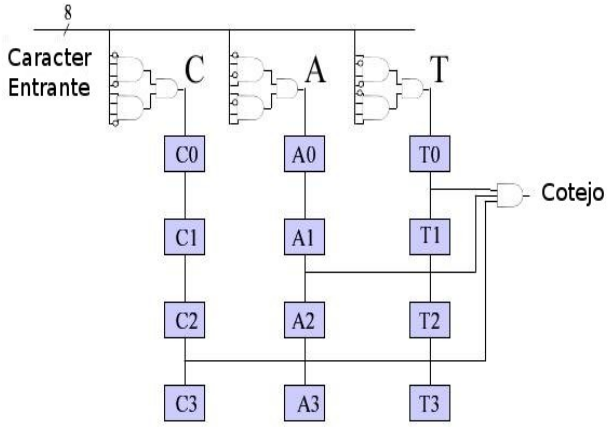


Fig. 13. Arquitectura shift-and-compare: detecta la cadena CAT, a cada comparador se le asigna un pipeline con un tamaño de acuerdo a la posición del caracter en la palabra [5].

Nótese que si el pipeline está compuesto por *flip-flops*, se puede detectar varias cadenas que compartan los mismos caracteres. Seleccionando las longitudes correspondientes de un pipeline según la distancia que ocupe ese caracter en la cadena en cuestión, y realizando una función AND para cada cadena del grupo se logra lo anterior, esto posibilita compartir el hardware entre varias cadenas. Otra ventaja de esta arquitectura es la fácil escalabilidad para procesar múltiples caracteres por ciclo, si tomamos en cuenta cada desplazamiento posible que pueda tener la cadena en el conjunto de caracteres que aceptara, el costo de procesar k caracteres por ciclo, es replicar el hardware total k veces.

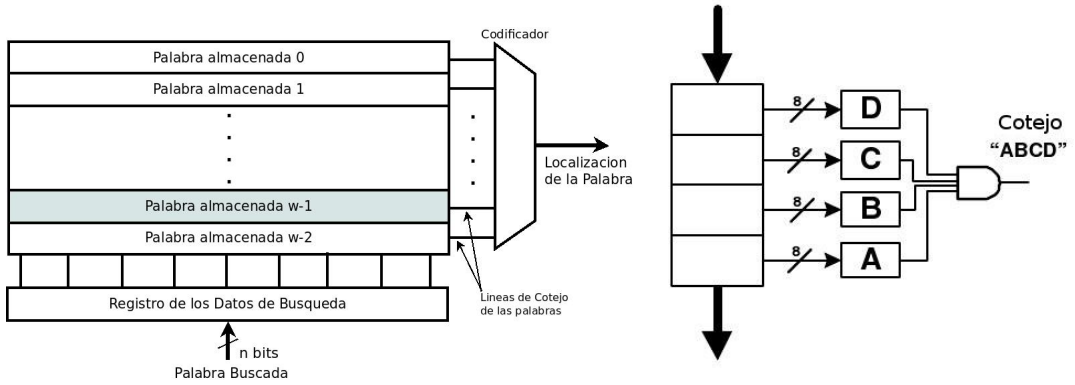
5.4. CAM y comparadores discretos

Las memorias de contenido direccionable CAMs, del inglés *Content-Addressable Memory*, es un tipo de memoria de computadora empleada en determinadas aplicaciones que requieren velocidades de búsqueda muy elevadas.

Al contrario de las memorias estándar, memorias de acceso aleatorio, *(RAM, Random Access Memory)* en las que el usuario introduce una dirección de memoria y la RAM devuelve los datos almacenados en esa dirección, una CAM está diseñada de manera que el usuario proporciona los datos y la CAM busca en toda la memoria para ver si esos datos están almacenados en alguna posición. Si los datos son encontrados, la

CAM devuelve una lista de una o varias direcciones en las que la información fue encontrada (en algunas arquitecturas, también devuelve los propios datos buscados, u otros tipos de información), en la figura 14a se muestra un esquema de este proceso. Esta propiedad de las CAM ha sido ampliamente explotada en el reconocimiento de cadenas [50] [37] [15].

Existen dos formas básicas de implementar las propiedades de las CAMs en FPGA en dependencia del tipo de recurso que se utilice. La primera forma es mediante el uso de las memorias embebidas en los dispositivos, que en la mayoría de los casos son de ancho de bus de dirección y datos programable. La otra forma de implementar CAMs es mediante comparadores implementados con lógica reconfigurable, donde cada comparador reconoce un caracter de la cadena y el resultado final de todas las comparaciones individuales es pasado a una función AND dando el resultado a su salida, esta forma es conocida como *comparadores discretos*, ver figura 14b.



(a) Memoria CAM **(b)** Comparador Discreto

Fig. 14. (a) Memoria CAM, las palabras a la entrada direccionan una posición de memoria la cual contendrá un valor lógico que indique la presencia de esta en la CAM.(b) Comparador discreto para la cadena *abcd*, cada comparador consume solo un slice.

Generalmente las implementaciones de CAMs utilizando comparadores discretos, alcanzan frecuencias de operación más elevadas que las basadas en memoria. Sin embargo ambos esquemas conllevan generalmente un consumo elevado de recursos. Con el objetivo de reducir este costo se han realizado varias investigaciones.

Con el objetivo de reducir el costo de recursos de hardware, Serif Yusuf y Wayne Luk [50], aplican una serie de transformaciones y optimizaciones de forma tal que el conjunto de cadenas queda representado como un diagrama de decisión binaria reducido. Este método es una forma compacta de representación de expresiones booleanas, a partir del cual se genera un árbol donde cada nodo representa una condición de verdadero o falso, (en el caso de las cadena sería si un determinado caracter del texto coteja o no con un caracter de la cadena), desde cada nodo se accede solamente a dos próximos nodos, en dependencia del resultado de la expresión. De esta forma se hace posible una implementación de CAMs utilizando comparadores discretos en forma arbórea, al que los autores denominan BCAM. En su implementación lograron una reducción del 30% en el costo, comparado con otras implementaciones basadas en CAMs, alcanzando $2.5Gbps$.

Observando la figura 14b podemos apreciar que en la medida en que crece el tamaño de la palabra se incrementan las entradas de la compuerta AND, en hardware esto implica la interconexión de varias *Slices* para lograr una función de N entradas lo que aumenta considerablemente la latencia disminuyendo la frecuencia de operación máxima de la arquitectura. Con el objetivo de reducir esta latencia, Sourdis [37] aplica *pipelines* donde cada paso se compone de una compuerta AND de 4 entradas, (implementable en

una LUT) y un registro que procesan y almacenan el resultado de la comparación del paso anterior. De esta forma la comparación se va realizando de forma gradual, obteniéndose el resultado en un tiempo fijo, reduciéndose en gran medida la latencia. Adicionalmente se implementan varias réplicas de los módulos de comparación para procesar múltiples caracteres por ciclo, alcanzando hasta $11Gbps$ pero con un alto consumo de recursos, no siendo posible reconocer un elevado número de cadenas.

5.5. Hashing para reconocimiento de cadenas

Las arquitecturas basadas en *hashing* son aquellas que procesan los datos de entrada mediante una función, o funciones *hash*. Estas aplican un conjunto de operaciones utilizando lógica combinacional y proveen un resultado garantizando que, siempre que se tengan los mismos datos en la entrada se obtendrá el mismo resultado. Esta condición no se cumple en sentido inverso, puesto que cualquier otro grupo de datos procesados por la función pueden generar el mismo resultado. En el caso de que dos o más cadenas que pertenezcan al grupo de cadenas buscadas den el mismo resultado se tiene una *colisión*, y si no pertenecen al grupo de cadenas dicha colisión constituirá un *falso positivo*.

El reto principal para esta técnica es precisamente la eliminación de falsos positivos y las colisiones. Los falsos positivos se eliminan generalmente con una segunda comparación contra la cadena que se supone que sea la cadena a la entrada y las colisiones con funciones de *hashing* más complejas o mediante la propia arquitectura. Los índices generados por las funciones de hash son generalmente utilizados con dos propósitos fundamentales, el primero es utilizarlos en un preproceso para adaptar la arquitectura de forma que en un segundo paso se verifique la presencia de la cadena en cuestión, el segundo es tomar el propio índice como resultado de la comparación y reducir la existencia de falsos positivos haciendo las funciones hash más complejas.

5.5.1. Arquitectura basada en técnicas de hashing

En este tipo de arquitectura la funciones de hashing pueden constituir un preproceso con el objetivo de seleccionar un conjunto de posibles cadenas para luego comparar contra la cadena en el texto [42], o pueden ser propiamente el método de detección, mediante la inserción, extracción y comparación de las cadenas en las tablas hash [44].

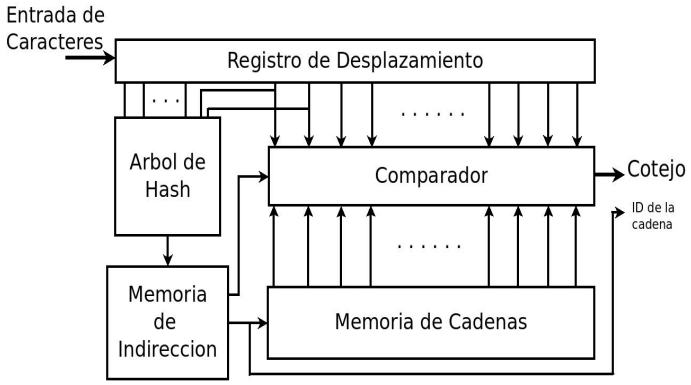


Fig. 15. Arquitectura basada en Hashing, el bloque de Hash, junto con la memoria de indirección, buscan una cadena candidata, luego esta se compara mediante fuerza bruta para dar un resultado del cotejo.

Desde el punto de vista físico el *hashing* tiene como ventaja que al poder procesar cualquier cantidad de caracteres en un único proceso se tienen tiempos de respuesta más cortos para el cotejo de las cadena.

Por otra parte se logra una representación compacta y reducida de las cadenas de forma tal que se consumen menos recursos para su posterior procesamiento. Las desventajas principales son la existencia de falsos positivos y de colisiones, a estas desventajas se han enfrentado los investigadores de varias maneras. Sourdis y otros autores en [42], figura 15, eliminan la colisiones utilizando un método de *perfect hashing*, que garantiza que no habrá colisiones en un grupo de cadenas. Un módulo de funciones hash genera la dirección en memoria donde está almacenada la posible cadena y la longitud de la misma, luego es comparada con la cadena en el texto, utilizando un tamaño apropiado para el comparador.

La ventaja más visible es que un mismo comparador es utilizado para el cotejo de todo el conjunto de cadenas compartiendo los recursos utilizados para implementarlo. Las cadenas son almacenadas utilizando los bloques de memoria embebidos en el dispositivo, por lo tanto el número de cadenas está limitado por la memoria disponible. La actualización de las cadenas se realiza mediante accesos a memoria, lo que constituye una ventaja cuando se quiera modificar el conjunto de cadenas reconocibles por la arquitectura. Otra ventaja es que el proceso de comparar toda la cadena y dar una respuesta ocurre en un solo ciclo de reloj lo que le permite lograr elevadas frecuencias de operación. Procesando 16 bits por ciclo (2 caracteres), en este trabajo se alcanzan $5.7Gbps$ con una capacidad de 20,911 caracteres a un costo de hardware de 0.57 celdas lógicas por caracter.

Tran y Kittitornkun en [44], las colisiones se eliminan implementando el algoritmo Cukoo-Hashing [34] en hardware, el algoritmo básicamente utiliza dos funciones hash, h_1 y h_2, con sus respectivas Tablas de Hash, T_1 y T_2, donde $T_1[h_1(x)]$ es un espacio de memoria direccionado por la función hash correspondiente donde se almacena una cadena x. Si al insertar el índice de una cadena en un espacio de memoria este ya se encuentra ocupado por una cadena y, entonces hay una colisión, para eliminarla, se sustituye por x en la posición de memoria actual y se preprocesa la cadena y con la otra función hash, direccionando y almacenando su índice en otra tabla donde se puede encontrar con la misma situación, siendo este un proceso recursivo hasta que se encuentra un espacio vacío. Para evitar caer en un ciclo infinito, limitan las iteraciones por una constante, cumplida esta constante, se cambian las funciones hash y se reorganizan de nuevo todas las tablas.

El proceso descrito anteriormente se implementa en FPGA utilizando bloques de memoria para las tablas de *hash* los que son direccionados por las dos funciones de *hashing*, otro bloque de memoria un comparador se encarga de implementar la función de inserción recursiva del algoritmo. La mayor ventaja de este esquema es la fácil y rápida actualización del grupo de cadenas. En este trabajo, se utilizan extensivamente los bloques de memoria RAM embebidos en los FPGAs, logrando una razón de 0.033 celdas lógicas por caracter, y una velocidad de $2.28Gbps$.

5.5.2. Filtros Bloom

Los Filtros Bloom se implementan de la siguiente manera: dada una cadena de entrada X se realizan sobre esta r funciones hash, dando como resultado r valores que van desde 1 hasta l, estos valores se almacenan como un valor lógico en una dirección de un registro de l largo. Luego, se realiza este procedimiento para varias cadenas, a esta fase se le nombra fase de programación del registro.

En la fase de búsqueda al llegar una cadena Y, se realizan sobre ella las mismas funciones hash y, si el resultado coincide con todos los valores correspondientes a esa cadena en el vector, existe una gran posibilidad de que esa cadena pertenezca al grupo.

No se puede afirmar definitivamente que esa cadena se encuentra en el grupo debido a la existencia de falsos positivos que propicia este esquema, puesto que pueden existir cadenas de entrada que no sean del grupo, que al ser procesadas por las funciones hash generen resultados válidos. Por ello, se debe realizar una segunda comparación contra la cadena que indica ser la cadena de entrada, para dar un resultado final.

La probabilidad de falsos positivos viene dada por la ecuación:

$$f = (1 - e^{-pk/l})^r \tag{1}$$

Donde l es el largo del vector, r la cantidad de funciones hash y p es el número de cadenas programadas dentro del filtro. Las cadenas programadas en un filtro deben tener la misma longitud, por lo que se debe realizar un filtro para cada grupo de cadenas con el mismo tamaño. La longitud fija de las cadenas para cada filtro y la probabilidad de falsos positivos constituyen la mayor desventaja del uso de los filtros Bloom.

Mediante simple inspección notamos que el aumento de l, y r, disminuye la probabilidad de falsos positivos, pero implica un aumento de recursos, puesto que aumentan los registros del vector y aumenta la lógica utilizada para implementar las k funciones. Por otra parte, si se quiere implementar un mecanismo para eliminar los falsos positivos, es necesario adicionar más lógica para el método de comprobación final que asegure que la cadena pertenece al grupo. La capacidad de agrupar cadenas es a todas vistas elevada. Dharmpurikar y otros autores en [20], implementan filtros Bloom con una probabilidad de falsos positivos de 0.00097 con 1,419 cadenas por filtro, alcanzando 12,771 cadenas en total. La capacidad de procesamiento en cambio, no presenta un crecimiento considerable, solo 518$Mbps$, esto se debe quizás al uso intensivo de los bloques de memoria embebidos.

6. Estrategias para la reducción de hardware

Independientemente de que la disponibilidad de los recursos programables sea mayor o menor en dependencia del dispositivo, y existan arquitecturas más o menos costosas, existen varias optimizaciones y estrategias de diseño con el objetivo de reducir el costo de área programable de la arquitectura. El propósito es claro, mientras menos recursos se utilicen para detectar las cadenas, más cadenas podrán se detectadas.

La reducción del hardware tiene otras implicaciones favorables, en la medida que una arquitectura sea menos costosa será más viable replicarla con el objetivo de ganar en capacidad de procesamiento. Además desde el punto de vista de las característica físicas del dispositivo, este consumirá menos potencia de corriente, y en la medida que la arquitectura sea más compacta espacialmente tendrá una menor latencia y por tanto alcanzara una mayor frecuencia de trabajo.

6.1. Recursos compartidos

Una forma eficiente de ahorrar recursos es eliminando redundancias, o sea no implementar, varias veces el mismo circuito, sino compartir el resultado de la función que este realiza entre los demás componentes de la arquitectura. Una de estas optimizaciones saca ventaja de la existencia de prefijos comunes entre las cadenas.

- **Prefijo común compartido:** Sean cadenas estáticas o expresiones regulares, estas poseen prefijos en común. Compartir prefijos no es más que compartir el resultado que proporciona el circuito lógico que reconoce un prefijo entre el resto de los circuitos que reconocen el resto de las cadenas que comparten dicho prefijo ver figura 16 [7].

Esta estrategia se ha utilizado en [7] [11] con expresiones regulares. Cheng-Hung Lin y otros autores en [27] hacen un uso intensivo dela uso compartido de los de recursos de hardware al compartir el resultado de los circuitos que detectan factores comunes entre las cadenas. Bajo determinadas restricciones, alcanzan

una reducción en el área de un 46 % frente a un 21 % típico que se produce al compartir prefijos. Baker y Prasana en [6] comparte prefijos en una arquitectura *shif-and-compare* y Yusuf y Luk en [50] comparten prefijos mediante una estructura arbórea implementada con CAMs.

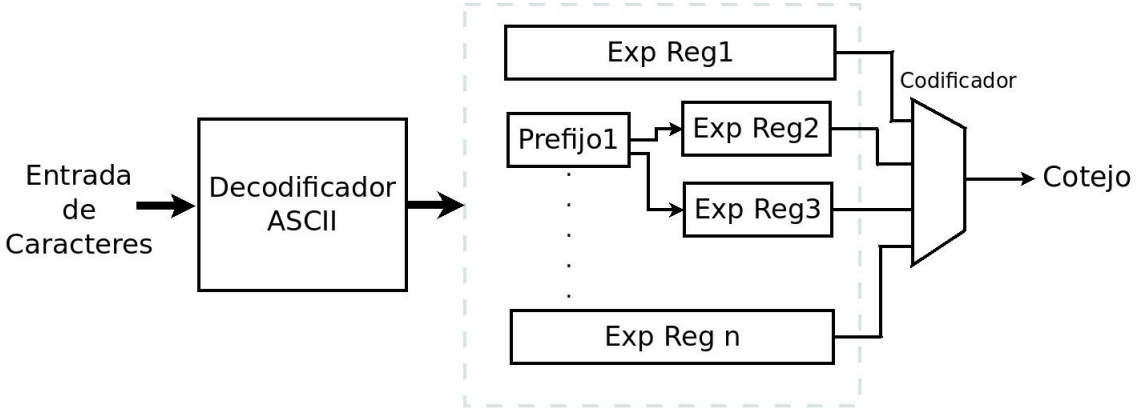


Fig. 16. Arquitectura donde se comparten los prefijos de las expresiones regulares, el prefijo 1, es común para la expresion regular 2 y 3 [7].

La desventaja de compartir hardware entre otros módulos estriba en el aumento del *fan-out* de los circuitos, reduciendo los tiempos de establecimiento de la señal a la entrada de estos, o sea un aumento de la latencia que penaliza en cuanto a frecuencia de trabajo, puesto que el intervalo de tiempo máximo entre pulsos de reloj, siempre va a ser mayor que el retardo máximo de una señal cualquiera dentro del circuito entre el origen y el destino de la misma, esto provocó que en [32] por ejemplo, se desechara la idea de compartir prefijos.

6.2. Prefiltrado

Las estrategias de prefiltrado ejecutan dos pasos fundamentales, el primer paso, realiza una reconocimiento parcial de la cadena o de la regla, esta búsqueda parcial, sirve para reducir el grupo de cadenas o reglas a comparar en el próximo paso, luego en el segundo paso se realiza una búsqueda exhaustiva. El hecho de reducir temporalmente el conjunto de cadenas o reglas a comparar, implica que se pueden utilizar menos recursos toda vez que la arquitectura que realice el paso subsiguiente pueda configurarse para reconocer solamente el subgrupo en cuestión cargando la configuración desde memoria.

En la sección 2.2 se muestra un ejemplo de regla de Snort, dicha regla se puede dividir en dos partes, cabecera y cuerpo. La cabecera se constituye de aquellos campos que no involucran la carga de datos de los paquetes, o sea, protocolos, (TCP, UDP, ICMP, entre otros), direcciones IPs origen y destino, y puertos origen y destino. El resto de la regla, (cuerpo), involucra el reconocimiento de cadenas en la carga de datos de los paquetes, pudiendo requerir de la ocurrencia de varias cadenas, o de varias expresiones regulares, para generar la acción definida para la regla.

Sourdis y otros autores en [39] implementan prefiltrado reconociendo la cabecera y parte del cuerpo de la regla para cada paquete. Según los autores, al reconocer la cabecera y hasta 8 caracteres del prefijo de la primera cadena en la carga de datos, un paquete entrante reduce la comparación a 1.8 reglas como promedio. Teniendo 32 reglas como máximo para evaluar por cada paquete, el prefiltrado previene el reconocimiento de casi el 99 % de las reglas de Snort por paquete.

6.3. Detección y corrección del desalineamiento

En la sección 4.4 mencionamos que cuando se procesan múltiples caracteres a la vez, dentro del conjunto de bytes que se procesan no se conoce donde comienza la cadena. Este problema implica que se tenga que replicar el hardware utilizado para poder reconocer la cadena sea cual sea la posición donde esta comience, a este problema le denominamos problema del desalineamiento.

Si se lograra conocer de antemano, como resultado de algún preproceso donde comienza la cadena, entonces se puede adaptar en una segunda etapa, el resto de la arquitectura de forma que se alineara con la cadena y fuese capaz de reconocerla sin la necesidad de replicar el diseño varias veces.

En un trabajo realizado por Yeim y otros autores en [12], a partir de la detención del prefijo de la cadena se reconoce el alineamiento de esta, y luego se configura la entrada de un autómata utilizando multiplexores de forma tal que se alinea la entrada del mismo con el comienzo de la cadena. De esta manera el autómata es capaz de reconocer la cadena procesando múltiples caracteres por ciclo sin que este posea hardware replicado, en este trabajo se muestran magníficos resultados que alcanzan 7.27$Gbps$, procesando 4 bytes por ciclo y con una capacidad de 16,000 caracteres.

6.4. Uso de las prestaciones de los dispositivos FPGA

Otra estrategia útil para la realización de diseños eficientes en cuanto a área, consiste en la explotación de las facilidades o prestaciones que presentan los dispositivos FPGA sobre los cuales se desarrolla el diseño.

Las familias de dispositivos tienen características propias en cuanto a la arquitectura interna de sus bloques lógicos y bloques especiales embebidos que al ser utilizadas redundan en un ahorro de recursos. Dos ejemplos ponemos a continuación, el primero ya fue mostrado en la sección sobre la arquitectura interna de los FPGAs, se trata de el uso de las LUT como registros de desplazamientos, SLR16, figura 9. El segundo es el uso de la lógica del *carry* interna de las slices para lograr que comparadores de 8 bits consuman sólo una *Slice*, figura 17, propuesto por Caver y otros autores en [11], esta optimización reduce el costo de hardware debido a los comparadores considerablemente.

El uso de las LUT como registros de desplazamiento no sólo ofrece una reducción del área sino que además, al ser una memoria, no hace uso de la red exterior de enrutamiento del FPGA lo que le da mayor velocidad. Esta estrategia se ha usado en [11] para bloques básicos de expresiones regulares que se adicionan en este trabajo expuestos anteriormente. Sourdis en [37], implementa *shift-and-compare*

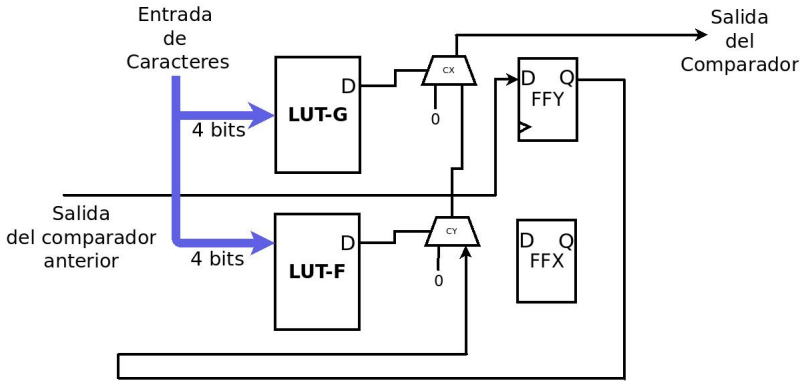


Fig. 17. Uso de la lógica de acarreo para crear un comparador de 8 bit en un slice, la salida se de celda lógica se vuelve a reinsertar en el slice por medio de la entrada de la lógica de acarreo, de esta manera, se logra una función de 8 entradas como comparador en solo un slice.

usando SLR16 como registros de desplazamiento, el autor nombra su diseño *Decode CAM*, DCAM ver figura 18. El autor clasifica su diseño como una optimización que sigue esquema de memorias CAMs, en nuestro criterio se asemeja más a una arquitectura de tipo *shift-and-compare* dada la presencia de pipelines dedicadas a los caracteres, véase la similitud entre las arquitecturas de las figuras 13 y 18.

La desventaja de esta estrategia es que el tamaño del registro es fijo, o sea no se puede acceder como en [5], a cualquier distancia del registro, por lo tanto para poderlo compartir deben ser sólo por caracteres iguales y que además tengan la misma posición dentro de la cadena.

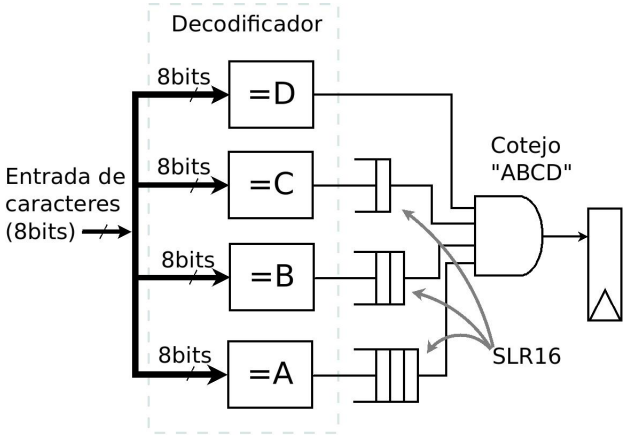


Fig. 18. Arquitectura Pre-decode CAM, DCAM, que utiliza SLR16, bajo el mismo esquema de shift-and-compare, a la salida de los comparadores se introducen demoras de acuerdo con el posición del caracter en en la palabra [37].

7. Estrategias para el aumento de la capacidad de procesamiento

Ya hemos mencionado, que los dispositivos lógicos programables actuales logran frecuencias de operación entre 50 y 300 MHz, y que además, el aumento de la lógica involucrada en una arquitectura va en detrimento de esta frecuencia máxima. Con la elevada velocidad en los flujos de datos de las redes actuales, es difícil competir procesando solamente un caracter a la vez, se hace necesario entonces arquitecturas *multicaracter* que procesen más de un caracter por ciclo de reloj.

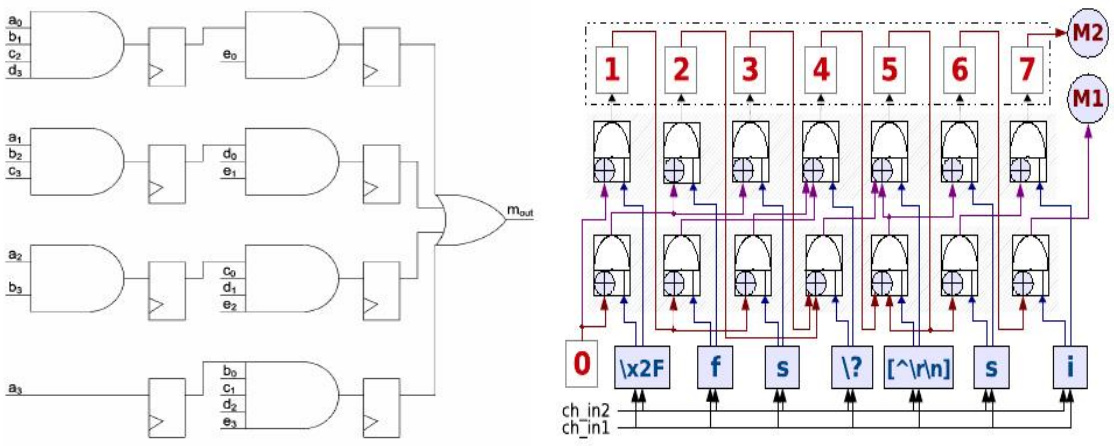
Otra manera de afrontar el problema de la capacidad de procesamiento es mediante la reducción de la latencia que como ya hemos mencionado implica poca penalización debido a la propia complejidad del diseño de la frecuencia de operación. La reducción de la latencia, ya sea mediante optimizaciones, o con diseños que generan con poca latencia constituye el otro frente de lucha por lograr elevadas capacidades de procesamiento. En este grupo podemos citar, Predecodificación, Particionamiento, y *Fan-out Tree*

7.1. Arquitecturas multibyte

Las arquitecturas como CAM, DCAM, Comparadores Discretos y *Shif-and-Compare*, son fácilmente escalables para procesar múltiples caracteres por ciclo, a costa de aumentar de forma exponencial el consumo de recursos. Esto se debe a que al tomar varios caracteres de un flujo no se puede saber donde comienza la cadena en realidad, necesitándose una replica del diseño por cada posible desplazamiento del inicio de la cadena en el conjunto de caracteres. En [5] y [37] se alcanzan 6.4 $Gbps$ y 10 $Gpbs$ respectivamente,

al procesar k bytes por ciclo, se generándose un aumento de k comparadores, compuertas, y k nuevos *pipelines*.

En el caso de autómatas para reconocer cadenas estáticas, en los que la secuencialidad de los caracteres es fundamental para alcanzar un estado final, se logra procesar múltiples caracteres generando k máquinas de estados en paralelo, donde cada transición está relacionada con un conjunto o bloque de caracteres, y cada máquina se diseña para un desplazamiento específico figura. 19 (a) [17], la salida de las k máquinas implican la ocurrencia de la misma cadena. En este trabajo se alcanzan 99 *Gbps* procesando 64 caracteres por ciclo, pero con un consumo de recursos tal que sólo le permite contener 250 caracteres. Esta es una muestra de la relación inversa entre capacidad de procesamiento y costo de área programable. Con las



(a) (b)

Fig. 19. Arquitecturas multibyes: (a) NFA multibyte, (b) REM multibyte, ambas arquitecturas procesan mas de un caracter por ciclo.

expresiones regulares el asunto se torna un tanto más difícil debido a las complejas relaciones que se establecen entre los caracteres, sin embargo en [48] Yang y Prasanna proponen una arquitectura, figura. 19 (b), donde los caracteres son pasados por un pipeline. El primer bloque de la expresión regular que detecte una ocurrencia impone el nivel del pipeline del cual los subsiguientes bloques aceptarán los caracteres. El valor máximo alcanzado es de 10*Gbps*, procesando 8 bytes por ciclo de reloj con 264 expresiones regulares.

Además, existe siempre una forma de procesar múltiples bytes por ciclo que es independiente de la arquitectura, esta toma ventaja de las características del flujo de datos de red que consiste en paquetes con una carga de datos limitada. Esta estrategia se basa en demultiplexar los paquetes de red entre varias arquitecturas idénticas, obteniéndose una elevada velocidad de procesamiento total, pero con dos deficiencias fundamentales. La primer deficiencias es que se pierde la relación entre paquetes que intervengan en la activación de una misma regla, y la segunda, el hardware puede ser subutilizado al existir la posibilidad de que módulos estén procesando un paquete mientras que otros estén en estado de espera.

7.2. Predecodificación

La Predecodificación consiste en decodificar centralizadamente cada caracter entrante del flujo de datos, mediante una serie de comparadores de caracteres obteniendo un vector de 256 líneas de bits (si se decodifica el código ASCII completamente), donde cada carácter quedará representado por una línea.

La razón de esta optimización se debe a que distribuir 8 líneas por cada caracter a través de toda la arquitectura, implica un uso elevado de los recursos de interconexión y la introducción de trayectorias de elevada latencia. Además, cada módulo de comparación tiene que desempeñar sus funciones sobre 8 líneas de bits, lo que conlleva a un aumento de la lógica necesaria. Todo lo anterior hace que se eleve la latencia y el costo del hardware, disminuyendo la frecuencia de operación máxima que puede alcanzar el dispositivo.

Obviamente, se necesitan mucho menos recursos para enrrutar una línea de un bit y desarrollar operaciones lógicas de un solo bit. Esta optimización de probada eficiencia es propuesta por Clark y Schimmel en [16] luego utilizada en [5][7][48][37].

7.3. Particionamiento

Baker y Prasanna proponen en [5] dividir el conjunto de cadenas en subgrupos bajo el criterio de: maximizar el conjunto de cadenas con caracteres comunes en subgrupos, y minimizar la similitud de cadenas entre subgrupos diferentes. Aplicando teoría de grafos, particionan el conjunto total de cadenas en varios

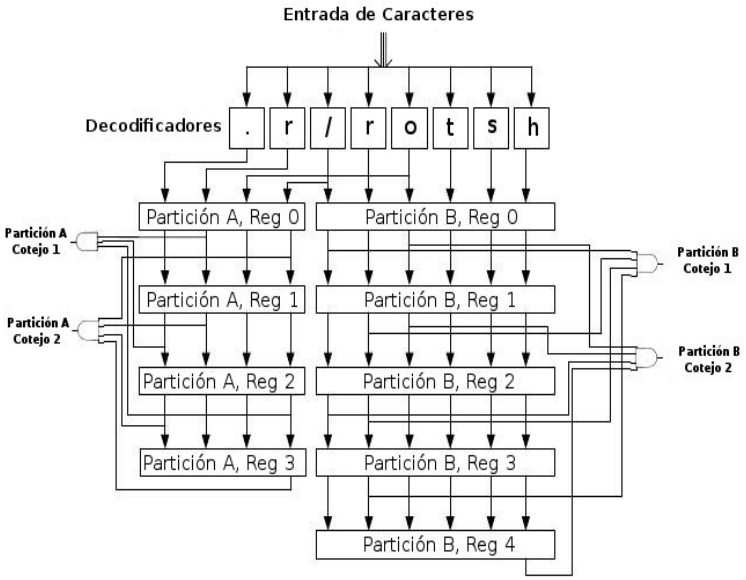


Fig. 20. Arquitectura Shift-and-Compare con Particionamiento, en cada partición las cadenas comparten la mayor cantidad de caracteres, pudiendose compartir el hardware utilizado para su detección [5].

módulos que comparten pocos caracteres en común entre ellos, mientras que dentro de cada módulo se maximiza la cantidad de caracteres en común.

Esto implicará que al ser implementado en hardware, se maximice la cantidad de recursos compartidos en un mismo grupo, a la vez que se puedan compartir entre grupos los recursos dedicados a la detección

de caracteres que son menos comunes. Por un lado se comparten más recursos entre módulos, y por otro se reducen las líneas de bits a distribuir entre módulos diferentes dando como resultado una arquitectura mucho más ágil y eficiente. La figura. 20, muestra esta idea, en ella se muestran dos particiones, A y B, las cadenas dentro de cada una comparten más caracteres mientras que, entre particiones, las líneas de bits compartidas son menores.

Otra forma de hacer un uso más eficiente de los recursos es la centralización en módulos de la funciones que realicen procedimientos semejantes. En la figura 16 [7] se puede apreciar como se separan en módulos las funciones dedicadas al reconocimiento de las subcadenas estáticas de las expresiones regulares, de las que realizan la función de reconocer las correspondientes a clases de caracteres, y luego se comparte el resultado entre las demás partes de la arquitectura.

7.4. Fan-out Tree

Como definimos en la sección 5.2, el *fan-out* de un elemento lógico se conoce como el número máximo de elementos lógicos que este puede alimentar. A mayor cantidad de dispositivos conectados a la salida de un elemento lógico, el tiempo en que la señal se estabilice en la entrada de estos será mayor, y el tiempo de reloj estará acotado por el mayor de los tiempos de establecimientos de la señal entre todas las líneas.

Para reducir esta latencia Sourdis [37] propone realizar una estructura de arbórea que divida en cada nodo la carga de datos aliviando el *fan-out* a cada paso e introduciendo por tanto un conjunto de tiempos de reloj fijo, desde que los datos están a la entrada hasta que son procesados por los módulos de reconocimiento. La implementación es sencilla, cada nodo es un *buffer* de 8 o más bits, y cada nodo final es un registro de desplazamiento que provee los caracteres a las máquinas de comparación de cadenas. La desventaja de esta propuesta es que introduce un costo en hardware al tener que disponer de recursos para la implementación del árbol.

8. Compilación de diseño en FPGA

La automatización del proceso de reconfiguración es de vital importancia, puesto que las reglas se actualizan constantemente. Reconfigurar cualquier arquitectura no puede ser un proceso engorroso que tome mucho tiempo o que requiera de la competencia de un programador de hardware.

La forma de compilar expresiones regulares en FPGA se puede decir que es propia para cada propuesta de arquitectura presentada por los investigadores, esto se debe, primeramente a que cada diseñador encamina su compilador a optimizar las características particulares de su arquitectura y por otra parte, no existe, hasta donde hemos podido indagar, una herramienta genérica para la generación de forma programática de cualquier arquitectura de grandes dimensiones.

Caver y otros autores en [11] desarrollan un compilador de expresiones regulares en hardware mediante el uso de un conjunto de librerías de Java, JHDL. Cada circuito en JHDL es representado como un objeto y las interconexiones y los circuitos son creados en la llamada del constructor, luego JHDL genera los ficheros de configuración del dispositivo FPGA para implementar el diseño. Moscola y otros autores en [32] usan un analizador léxico, Jlex, que genera las tablas de estados de los NFA correspondientes y luego usando la herramienta GWAK los traducen a código VHDL. Mitra y otros autores en [30] mediante las librerías de PCRE generan de forma automática el código VHDL. Estas librerías constan de dos partes, un compilador de expresiones regulares a NFA, y un motor de búsqueda de expresiones regulares que transforma los códigos de operación generado por dicho compilador mediante un programa a código VHDL.

Otras propuestas [42] [36], consideran no reconfigurar la lógica de los dispositivos, sino mantener las cadenas en memoria, ya sea embebida o externa, a fin de hacer más rápida y fácil la actualización, además de lograr una interacción software-hardware más extendida que permita lograr mayor flexibilidad y complejidad en las inspecciones a los flujos de datos.

9. Taxonomía y algunas consideraciones

Haciendo un resumen de las Algoritmos desarrolladas en FPGA, encontramos que predominan 5 líneas fundamentales, ver figura 21. La primera es el desarrollo de Máquinas Finitas de Estados a partir de autómatas deterministas. Estás se implementan ya sea utilizando memorias para la codificación de los estados, en cuyo caso pueden implementar tanto el autómata de Aho-Corassick, como expresiones regulares [10], y las que utilizan codificación OHE. En estas últimas se utiliza la lógica configurable del dispositivo para describir expresiones regulares, (Incluimos las cadenas estáticas como un caso particular de expresión regular formada solo por caracteres y concatenación). La próxima vertiente es *Shif-and-compare*,

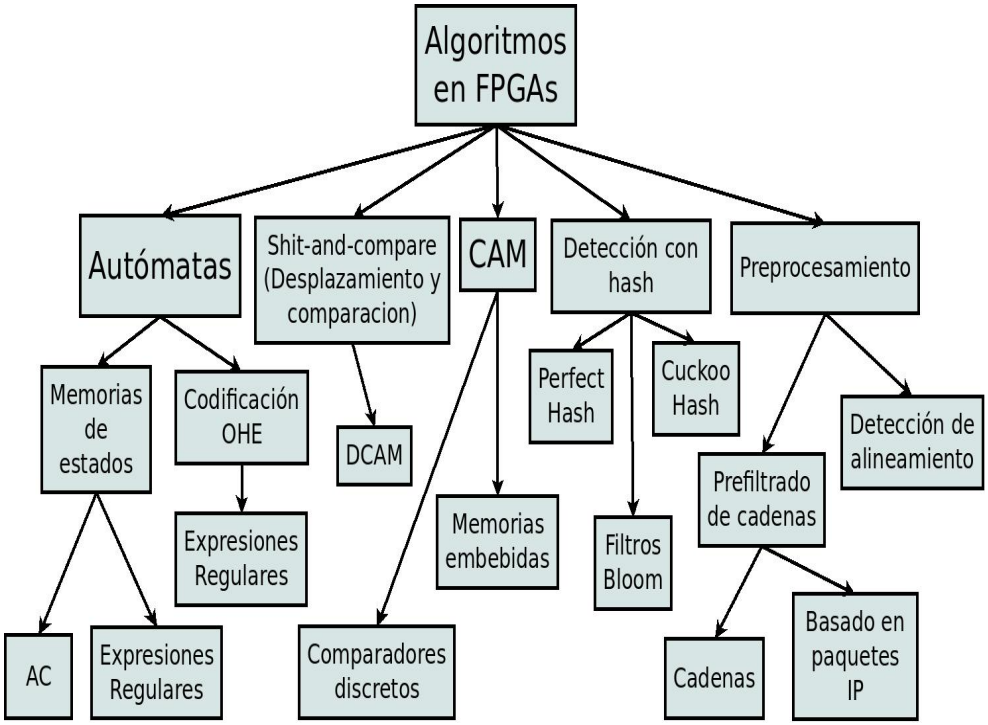


Fig. 21. Taxonomía de los diferentes algoritmos para al reconocimiento de cadenas en hardware.

donde los caracteres son introducidos en sus *pipelines* correspondientes lo que tienen un tamaño igual a la posición que ocupe el caracter en la cadena. Optimizaciones como *Decode-CAM*, DCAM utilizan las LUT como registros de desplazamiento para implementar estos *pipelines* logrando una arquitectura más rápida aunque un poco más costosa, debido a que es mas difícil compartir los recursos cuando se utilizan SLR16 como *pipelines*.

Le siguen aquellas arquitecturas que desarrollan el esquema de las memorias CAM, que inspeccionan mediante fuerza bruta la cadena completa o parte de ella en un flujo de datos verificando si existe

en la memoria. Hemos visto que en FPGA las formas de realizar este método son diferentes, ya sea usando memorias embebidas o comparadores discretos. La mayoría de las optimizaciones en esta línea van encaminadas a reducir el costo en hardware.

Continúan aquellas que reconocen las cadenas utilizando funciones de *hash*. Estas arquitecturas presentan dos problemas fundamentales, la existencia de falsos positivos y de las colisiones. Generalmente los falsos positivos son resueltos en una segunda etapa de detección donde se compara la cadena potencial con la cadena real del conjunto, en este podemos encontrar a los Filtros Bloom. Las colisiones por su parte son resueltas mediante métodos de *prefec-hashing* [42] o mediante implementaciones hardware del algoritmo Cuckoo-Hashing [34].

Tabla 1. Comparación entre distintas propuestas para reconocimiento de cadenas.

Descripción	Bits/ciclo	Dispositivo	Procesamiento (Gbps)	Celdas Lógicas	Celdas Lógicas /carácter	Memoria (KB)	Caracteres	Cadenas o Reg. Exp
Cho et al. [14] Hashing, Pre-filtering	8	Virtex4LX15	2.08	8480	0.26	276	32,384	2,044
Sourdis et al. [42] Hashing	16	Virtex2-1500	5.734	12,106	0.64	612	20,911	N/A
	8	Virtex2-1000	2.108	6,272	0.44	288		
Dharmapurikar et al. [20] Hashing, F.Bloom	8	Virtex XCV2000E	0.518	N/A	N/A	N/A	408,000	12,771 (Cad.)
Mitra et al. [30] FSM	128	(Virtex4)*2	12.9	N/A	N/A	768	N/A	200 (RE)
Yang et al. [48] FSM	64	Virtex4	10.0	27,000	N/A	N/A	N/A	267 (RE)
Clarck et al. [17] FSM	512	Virtex2-8000	100	N/A	N/A	N/A	250	N/A
	64	Virtex2-8000	10	N/A	N/A	N/A	15,000	N/A
	32	Virtex2-8000	2.5	N/A	N/A	N/A	40,000	N/A
	8	Virtex2-8000	1	N/A	N/A	N/A	50,004	N/A
Soewito et al. [36] FSM	8	Virtex 4	4.0	N/A	N/A	34.36	7,637	500 (Cad.)
Bispo et al. [7] FSM	8	Virtex4	2.9	25,074	1.28	0	19,000	509 (RE)
Jung et al. [24] FSM-AC	8	Virtex 4	2.2	6723	0.209	184	32,168	1,326 (Cad.)
Lunteren et al. [29] FSM-AC	8	Virtex 4	2.0	N/A	N/A	128	31,600	2,000 (Cad.)
Moscola et al. [32] FSM	32	Virtex XCV2000E	1.184	8,134	19.4	768	N/A	315 (RE)
Sourdis et al. [41] DCAM	32	Virtex2-6000	9.708	64,268	3.56	0	18,036	1,466 (Cad.)
	8	Virtex2-3000	3.080	23,228	1.58	0		
Sourdis et al. [40] Disc.Comp	32	Virtex2-6000	8.06	47,686	19.40	0	2,310	210 (Cad.)
Cho et al. [15] Disc. Comp	32	Altera EP20K	2.88	17,000	10.55	0	1,611	105 (Cad.)
Yusuf et al. [50] Disc. Comp	8	Vietex XCV8000	2.5	11,780	0.6	N/A	19,000	N/A
Baker et al. [5] Shift-and-compare	32	Virtex2-VP100	4.48	15,010	N/A	N/A	N/A	1,000 (Cad.)
Yiem-Kuan et al. [12] Deteccion de alineamiento	32	Virtex5-LX85T	7.27	7,635	2.10	N/A	16,028	N/A

Las arquitecturas que realizan un preprocesamiento en tiempo real sobre el flujo de datos, lo hacen con dos objetivos fundamentales, el primero es detectar cuales son las cadenas que potencialmente se encuentren en el flujo, permitiendo configurar una segunda etapa de comparación de manera tal, que solo se detecte ese conjunto reducido de cadenas potenciales. Este es un esquema de prefiltrado, el cual se lleva a cabo detectando las cadenas parcialmente, o en el caso de flujos de datos de internet, detectando previamente cuales de los campos de las cabeceras de los paquetes cumplen determinadas reglas de los NIDS, y cuales cadenas están relacionadas con esas reglas.

El segundo objetivo del preprocesamiento es conocer donde se encuentra el comienzo de la cadena en el flujo de datos. De esta manera se alinea la segunda etapa de comparación con el inicio de la cadena, de forma que no se necesite hardware replicado varias veces para detectar la cadena en todos sus posibles desalineamientos.

En la tabla 1, se muestra una comparación entre diferentes arquitecturas desarrolladas sobre FPGA. La columna Bits/ciclo, denota la cantidad de bits de entrada que son procesados en un sólo ciclo de reloj, y la columna procesamiento, la capacidad de procesamiento en *Gbps* que ha alcanzado la arquitectura. No hemos tomado en algunos casos la capacidad de procesamiento máxima alcanzada, puesto que esta corresponde generalmente con una cantidad pobre de cadenas o expresiones regulares, en cambio tomamos

aquellas velocidades alcanzadas en experimentos donde se involucran el mayor número de cadenas. La relación celdas lógicas por carácter, y uso de memoria, da una idea del costo de dicha implementación.

En la figura 22 se presenta un gráfico que muestra la relación entre el número de caracteres y la capacidad de procesamiento alcanzados por las diferentes arquitecturas. Se puede apreciar como para grandes capacidades de procesamiento el volumen de caracteres está por debajo de 30000, según Sourdis en [37] el promedio de caracteres por cadenas en las reglas de Snort es de 17 caracteres, basándonos en ese calculo y en el conteo del numero de cadenas que hemos realizado vemos que la capacidad de caracteres necesaria es de más de 108000 caracteres. Las mayores capacidades de procesamiento son

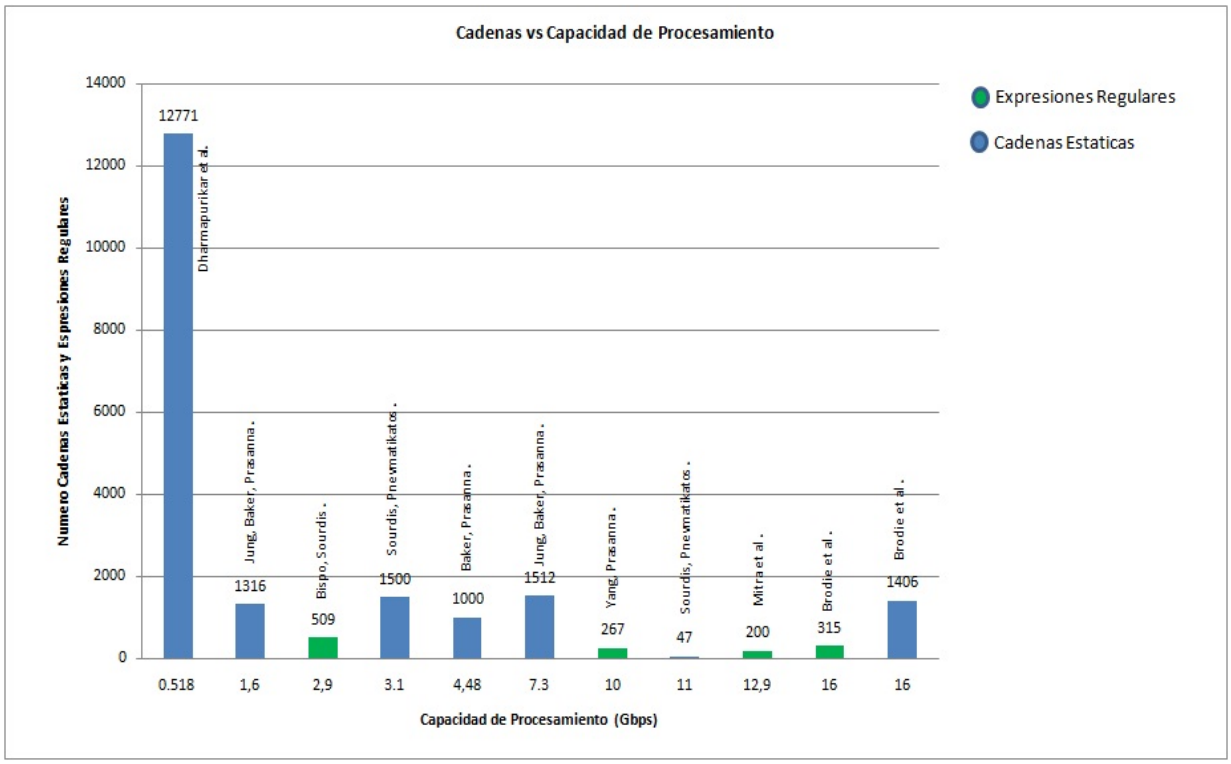


Fig. 22. Número de caracteres vs capacidad de procesamiento de las propuestas analizadas.

alcanzadas con arquitecturas multibyte. Mitra y otros autores en [30] con 16 módulos de 14 FSMs cada uno y distribuyendo la carga de cada paquete entre estos módulos llega ha 12,9$Gbps$ pero el elevado número de replicas de hardware sólo le permite buscar 200 expresiones regulares, donde más claramente se puede observar el costo de una arquitectura multibyte es en la propuesta basadas en FSM, con predecodificación y codificación OHE de Clark et al [17], donde para 8 $bits/ciclo$ se tiene una capacidad de 17537 caracteres, mientras que para 32$bit/ciclo$ esta es de 50004 caracteres.

La familia dispositivos FPGA, Virtex-5, posee frecuencia máxima de operación de 550MHz, y en su dispositivo con más capacidad de lógica, el XC5VLX330 con un número de 207360 celdas lógicas. Para superar los 10$Gbps$, suponiendo un caso ideal en el que no existe penalización en frecuencia divido a la latencia introducida por la implementación de la arquitectura, este debería procesar como mínimo 19$bits$ por cada ciclo de reloj, si la penalización en frecuencia fuera del 50 % entonces serian, 37$bits/ciclo$.

Para una arquitectura que soporte 108112 caracteres, tendríamos a nuestra disposición 1,91 células lógicas por carácter, este sería el costo límite para una arquitectura que pretenda reconocer ese conjunto

de cadenas, si para procesar más de un byte por ciclo el nivel de lógica adicionado supera este costo ya no se podría implementar completamente en este dispositivo, y por tanto no lograríamos capacidades de procesamiento del orden de los *Gbps*.

Este cálculo aproximado da una idea de que aún se necesitan soluciones prácticas y económicas dado que la tecnología subyacente no ofrece por sí misma toda la capacidad y velocidad requerida, y, que además se prevé un constante crecimiento tanto en la velocidad como en el conjunto de cadenas ha encontrar.

10. Propuesta para la disminución del costo en arquitectura multibyte

A continuación mostramos una propuesta de arquitectura para procesar dos bytes por ciclo que hemos simulado y comprobado su correcto funcionamiento. Esta posee como limitante la imposibilidad de reconocer aquellas cadenas que contengan los dos primeros caracteres iguales.

Para nuestra propuesta dividimos una cadena en dos subcadenas, cada subcadena va a formar parte del lenguaje de un autómata trie, con esto obtenemos dos tries independientes formados a partir de las cadenas pares e impares.

Cada sublista o subgrupo de cadenas se conforman con los caracteres impares y pares de cada cadena. Luego estas sublistas son descritas mediante una estructura arbórea sobre un conjunto de cadenas o *trie*, como se definió en la sección 3.4.1, figura 23. Los estados finales de trayectorias en ambos *tries* para una misma cadena se encuentran relacionados, digamos, mediante un AND, de manera que identificaremos una ocurrencia cuando ambos son activados. La presencia del caracter ε, indica que para cualquier caracter leído se avanzara al próximo estado, se puede concebir también como un retardo o registro de desplazamiento, su inclusión se debe a que el número de caracteres de la cadena es impar de manera que si no lo incluyéramos hubiera un desfasaje respecto al instante en que ambos *tries* alcanzan los estados finales para dicha cadena.

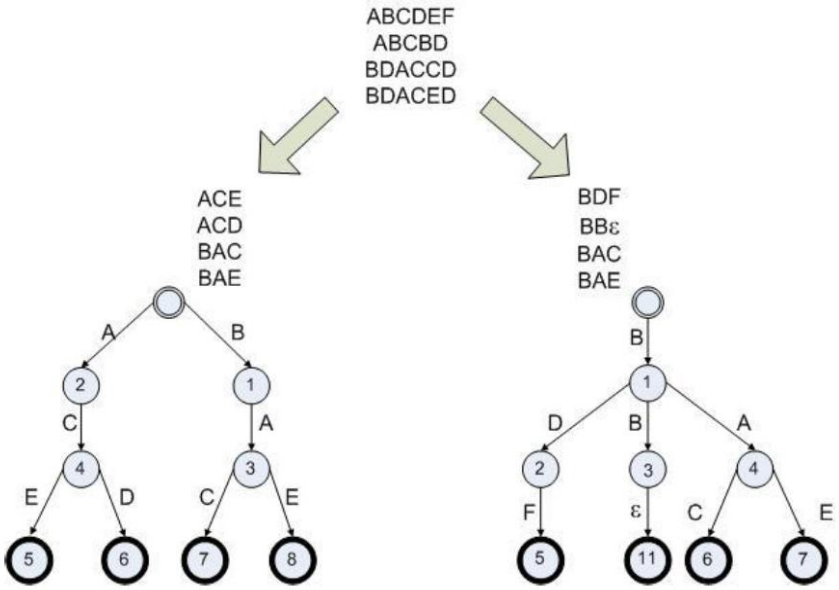


Fig. 23. Tries formados a partir de las secuencias pares e impares del conjunto de cadenas.

Luego, basados en esos dos *tries* para cada cadena, se incluye en el *trie* correspondiente la subsecuencia del *trie* opuesto, de esta manera se comparten los recursos utilizados por cada trie para codificar los estados. La lectura de dos caracteres por ciclo se realiza a través de un pipeline de dos niveles, uno desfasado con respecto al otro, y cada *trie* recibe las líneas de los caracteres de los dos niveles. La arquitectura diseñada para cada nodo del *trie* es tal que cuando se comienza a leer de un nivel del pipeline, se continúa leyendo de este hasta detectar o no la ocurrencia de la cadena. De esta forma se resuelve el problema del desalineamiento.

Cuando una secuencia determinada activa los estados finales de ambos tries para una misma cadena se detecta una ocurrencia.

La principal deficiencia encontrada siguiendo este esquema se debe a que las cadenas que tiene los dos primero caracteres iguales pasan desapercibidas, esto se debe a que los *tries* no continúan aceptando caracteres del nivel del pipeline que contiene la secuencia alineada correctamente.

Aún con esta deficiencia el objetivo de no replicar el hardware tantas veces como caracteres se adquieran a la entrada se cumple. Reconocemos que este puede ser un punto de partida para futuros trabajos, en este caso no se trata de descubrir la alineación de la cadena sino de compartir los recursos dedicados en los estados de cada *trie*, en vez de replicarlos enteramente.

11. Conclusiones

Recapitulando, el reconocimiento de un número grande cadenas en flujos de datos a elevadas velocidades, como en el caso de la seguridad en las redes informáticas, es una tarea que requiere de un gran esfuerzo computacional. Esto implica la necesidad de alcanzar una eficiencia tal, que permita reconocer de manera eficaz y sin afectar la velocidad en dicho flujo de datos, un conjunto de cadenas relacionadas en patrones de comportamiento que representan la ejecución de un procedimiento malicioso en contra de la seguridad e integridad de la red.

Los dos requerimientos esenciales a que se enfrentan este tipo de aplicaciones, se encuentran en constante crecimiento debido a la natural evolución tecnológica, estos son: las velocidades actuales y futuras en los flujos de datos de las redes informáticas, y el aumento de los patrones de comportamiento delictivos hacia la seguridad, calidad de servicio o política de acceso de la red, que redundan en un aumento constante y considerable del conjunto de cadenas que se requiere detectar.

Debido al comportamiento esencialmente secuencial de los GPP aún cuando estos poseen elevadas frecuencias de trabajo, (orden de los GHz), no se logran satisfacer en su totalidad estos requerimientos. La velocidad en los flujos de datos supera la capacidad de procesamiento de los mismos, induciendo a la ineficacia en la detección aún cuando se utilizan los algoritmos más eficientes. Esto ha motivado la exploración de otras plataformas tecnológicas que permitan la paralelización en la detección y que a su vez sean flexibles en cuanto a la actualización del conjunto de cadenas.

Los FPGAs, ofrecen la posibilidad de alcanzar una amplia concurrencia de procesos mediante la implementación de funciones con elementos lógico-físicos dedicados. Además poseen la capacidad de modificar una arquitectura ya implementada mediante la reconfiguración del dispositivo. Esto último es una propiedad importante que les permite ser flexibles en cuanto a la actualización del grupo de cadenas, satisfaciendo al menos temporalmente, y en la medida que sus recursos lo permitan, los requerimientos indicados anteriormente.

Aún con las posibilidades del uso de hardware reconfigurable, esta plataforma impone dos restricciones adicionales a las ya mencionadas, estas son: el uso de sus recursos es limitado, y su frecuencia

máxima de operación es baja, (orden de los MHz), con respecto a los GPP. Lo que hace que constituya un reto el desarrollo de arquitecturas hardware que cumplan con estos cuatro requerimientos fundamentales.

Para procesar un flujo de elevada velocidad, con una frecuencia menor de trabajo, es necesario procesar varios elementos de este flujo a la vez, lo que implica un aumento en el paralelismo involucrado y por tanto un aumento en el consumo de recursos. Esto provoca que el costo de hardware para la detección de una cadena se eleve, y por ende la capacidad de cadenas que este pueda detectar se reduzca.

Esta relación inversamente proporcional se logra reducir con la realización de diseños ingeniosos, y prácticas favorables de optimización, que son producto de múltiples investigaciones que han logrado satisfacer los requerimientos al menos temporalmente. En el presente no existe algoritmo cuya arquitectura implementada en hardware los satisfaga totalmente, con lo cual este continua siendo un problema abierto.

Las soluciones con arquitecturas multicaracter, los diseños poco costosos con una relación celda lógica por carácter baja, optimizaciones, y prefiltrado, los sistemas híbridos software-hardware donde la tarea de máximo esfuerzo computacional sea relegado a dispositivos FPGA, y las complejas relaciones que representan las reglas de detección sean asumidas por procesadores secuenciales dotados de gran flexibilidad, representan las bases y paradigmas de diseños en los que pensamos que se puedan lograr mejores resultados, y por ende posibles campos de investigación y desarrollo.

11.1. Análisis crítico

Teniendo en cuenta los requerimientos a los que se enfrenta la ejecución del reconocimiento de cadenas en flujos de datos a elevadas velocidades, las características de la tecnología subyacente, y el estado del arte de las arquitecturas estudiadas, destacamos un conjunto de problemas que consideramos puntos de partida para el análisis de las arquitecturas y la búsqueda de posibles soluciones.

1. No es posible alcanzar capacidades de procesamiento sobre los 10 Giga bits por segundo, si no se procesan más de un carácter por ciclo de reloj. Esto se debe a que las frecuencias de operación alcanzadas luego de la configuración del dispositivo son relativamente bajas.
2. Aumento exponencial del tamaño de la arquitectura al procesar más de un carácter por ciclo de reloj. Esto se debe esencialmente al problema del desalineamiento, o sea, no es posible conocer de antemano donde comienza la cadena en un flujo lo que conduce a la réplica del hardware con tal de cubrir todos los posibles desalineamientos de la cadena.
3. Disminución de la frecuencia de operación a medida que la arquitectura se hace más grande y utiliza más las líneas de interconexión del FPGA. Esto se debe a que aumenta la longitud de las trayectorias de conexión con otros bloques, y aumenta también el fan-out.
4. El costo en hardware de la arquitectura tiene un impacto negativo en el número de cadenas reconocibles, y en la capacidad de procesamiento alcanzable. Debido a que los recursos del FPGA no son infinitos, se deben establecer compromisos entre estos dos aspectos, siendo estos mejores mientras más eficientemente se usan los recursos disponibles en el dispositivo.
5. La utilización de los elementos de memoria embebidos para el desarrollo de arquitecturas generalmente basadas en autómatas, tiene como ventaja que se facilita la actualización del conjunto de cadenas, pero su desempeño es generalmente bajo. Por el contrario los diseños basados en la configuración de los elementos lógicos programables da mejores desempeños pero es más compleja su actualización debido a que esto implica reconfigurar el dispositivo.

La capacidad de procesamiento, el costo de área programable, y la capacidad de cadenas detectables por un dispositivo se encuentran estrechamente relacionados por la cantidad de caracteres procesados por ciclo,

la latencia introducida y la frecuencia de operación alcanzada debido a esta. Estas relaciones se pueden ver más claramente en el diagrama de estrella en la figura 24.

La capacidad de procesamiento aumenta en la medida que se eleva la frecuencia de operación de la arquitectura y se procesan más caracteres por ciclo de reloj. Aumentar el paralelismo mediante el procesamiento de múltiples caracteres por ciclo introduce un costo adicional a la arquitectura, con lo cual, los recursos de hardware destinados al volumen de cadenas serán menores. Por otro lado la latencia que una arquitectura posea depende del *fan-out* al que estén expuestos los elementos lógico-físicos y a la latencia debido a las trayectorias.

Teniendo en cuenta estos aspectos podemos definir como arquitectura óptima aquella que cubre el área sombreada en la figura 24. Este es el objetivo fundamental que se persigue en las investigaciones por desarrollar arquitecturas que ejecuten reconocimiento de cadenas, y que además, satisfagan los requerimientos de volumen de cadenas y capacidad de procesamiento impuestos por el avance tecnológico. Por lo tanto la eficiencia con la que los recursos disponibles de los dispositivos FPGAs sean utilizados va a ser determinante. Otro aspecto importante que no se debe dejar de tener en cuenta, responde a la adaptación a los cambios para enfrentar nuevos tipos de ataques, o contenidos maliciosos. Específicamente, se hace necesario que la actualización del conjunto de cadenas no sea un proceso que tome mucho tiempo, puesto que la seguridad de la red podría estar expuesta y vulnerable. Aquí se nos presenta otra situación compleja.

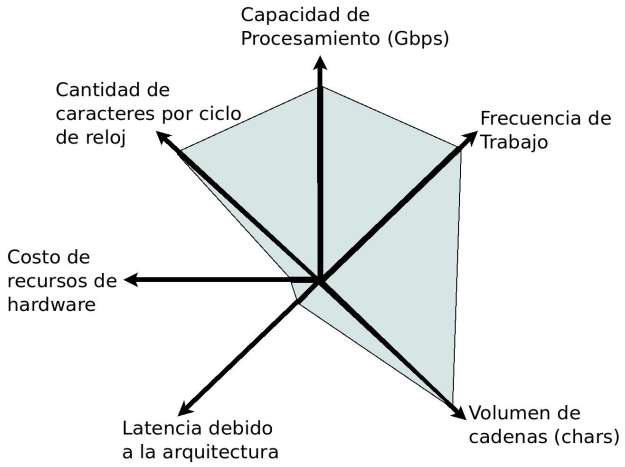


Fig. 24. Diagrama en estrella que muestra la interrelación entre los diferentes aspectos a tener en cuenta a la hora de implementar algoritmos para el reconocimiento de cadenas en FPGA. Los vectores perpendiculares no son influyentes entre ellos, los no perpendiculares inciden negativa o positivamente en dependencia de la orientación.

Si bien las arquitecturas que utilizan memorias embebidas para almacenar las cadenas son más fáciles de actualizar, puesto que solo requieren de la introducción de las cadenas en memoria mediante accesos a estas, el uso de tales memorias disminuye la frecuencia de operación conque trabaja el circuito. Esto se debe a que el acceso a las direcciones de memorias durante el proceso de detección de la cadena, es más lento comparado con las implementaciones que utilizan lógica configurable exclusivamente. Estas últimas en cambio, para actualizar el volumen de cadenas requieren de la reconfiguración del dispositivo, lo cual es un proceso más lento, y requiere de conocimientos en la programación de hardware.

La mayoría de los trabajos iniciales fueron encaminados a reducir el costo de la arquitectura más que a alcanzar grandes capacidades de procesamiento. En la arquitectura presentada por Sidhu y Prassana en [35], la introducción de expresiones regulares es una muestra de ello. La consabida capacidad de agrupar varias cadenas en el lenguaje reconocido por una expresión regular, se reflejaba en una reducción elevada del consumo de recursos y por ende más número de cadenas podían ser reconocidas por el dispositivo.

En el trabajo de D. Caver y otros autores en [11] el hecho de implementar en hardware autómatas de expresiones regulares totalmente independientes y que funcionaban de forma concurrente imprimió un nivel de paralelismo elevado, el cual puede verse como un esquema de *múltiples instrucciones, un solo dato*, (*MISD, multiple instruction single data*), que es el esquema de paralelismo predominante en los dispositivos FPGA que implementan reconocimiento de cadenas, puesto que el flujo de datos es único y se distribuyen entre los diferentes elementos que reconocen las cadenas de forma independiente, cual si fueran procesadores separados. Esta implementación hardware superaba 600 veces la capacidad de procesamiento alcanzada por los sistemas software del momento.

Los algoritmos basados en autómatas al escalar de forma natural al procesamiento de múltiples bytes por ciclo crecen exponencialmente, dado que para n bytes el alfabeto Σ aumenta a Σ^n. El trabajo de Clark y Schimmel [17] muestra de manera muy clara este incremento exponencial en función de la cantidad de caracteres procesados por ciclo, en la figura 19 (a), mostramos un esquema de un autómata que reconocería múltiples bytes por ciclo, y se muestra el hardware replicado varias veces de forma que logre reconocer la cadena sea cual sea el comienzo de esta en el conjunto de bytes que se toman de una vez.

En este momento nos gustaría retomar el punto 5 de las observaciones que hicimos al inicio del capítulo. La razón por la que se deba replicar varias veces el hardware es porque no se sabe de antemano el comienzo de la cadena. Con lo cual llegamos a la idea que ya habíamos expuesto en la sección 7.3.

Este es precisamente un frente en donde creemos que se puede investigar para el desarrollo de un diseño óptimo por dos motivos, el primero, se han realizado pocos trabajos en este sentido. Si bien Cho en [14] detecta donde comienza la cadena, no lo hace con el objetivo de procesar múltiples bytes por ciclo sino, para sincronizar la comparación entre los caracteres, Yeim en [12] sí detecta el desalineamiento para conocer donde comienza la cadena en el conjunto de caracteres procesados, desafortunadamente la existencia de falsos positivos obliga a utilizar recursos adicionales.

- *Si supiéramos como está alineada la cadena con respecto al conjunto de bytes procesados, podríamos alinear nuestro autómata de forma tal que esta sea reconocida por un único autómata sin necesidad de replicarlo para todos los posibles desalineamientos.*

El otro motivo por lo que pensamos que puede abordar más en esta línea se debe a que la reducción del costo sería bastante elevado para arquitecturas multibytes, pudiéndose alcanzar un buen compromiso entre capacidad de procesamiento y costo de hardware.

Otra de las forma de concebir autómatas en hardware fue la propuesta por Baker y Prasanna en [5], utilizando la técnica de *shift-and-compare* donde el procesamiento de múltiples bytes también implica un crecimiento exponencial del costo de la arquitectura. En este trabajo además se pone de manifiesto otro frente por el cual se aborda la optimización del diseño este es:

1. Optimizaciones, modificaciones y preprocesamientos sobre el conjunto de cadenas con varios objetivos:
   - Reducción del costo en hardware de la arquitectura.
   - Creación de nuevas formas de representar el conjunto de cadenas, de manera que se generen arquitecturas menos costosas y con elevados desempeños.
2. Optimizaciones en la implementación en hardware con dos objetivos fundamentales:
   - Aumento de la frecuencia de operación alcanzada por la arquitectura mediante la reducción de la latencia introducida.
   - Disminución del costo a través del uso compartido de los recursos de hardware.

En la literatura estudiada no se han encontrado muchos trabajos sobre las posibles optimizaciones que se podrían realizar al conjunto de cadenas que se necesita reconocer. Baker y Prasanna en [5] particionan el

conjunto de cadenas con un criterio de máxima similitud a fin de que se pueda compartir mejor el hardware utilizado.

Más allá de optimizaciones al conjunto de cadenas, Guinde [22] en un trabajo realizado en el 2010, realiza un procesamiento no para optimizarlo, sino más bien para representar las cadenas de manera no convencional, en modo de vectores de bits donde cada bit representa la existencia de una subcadena de tamaño fijo. De esta forma logran comprimir grandemente el volumen de cadenas en un conjunto de vectores de tamaño pequeño.

- *Reconocemos como posible área de investigación la modificación de la representación del conjunto de cadenas, aplicando teoría de conjuntos y otras herramientas matemáticas que que redunden en una reducción del hardware de las arquitecturas implementadas para reconocerlas.*

Sobre las optimizaciones de las implementaciones hardware hemos visto varios ejemplos que sacan provecho de las prestaciones de los dispositivos, como los comparadores de 8 bit en un slice, el uso de las memorias embebidas como memorias CAM, el uso de los SRL16, y otros que buscan compartir los recursos utilizados, como compartición de prefijo y de sufijo.

Otra forma de implementar autómatas en FPGA ha sido mediante el uso de las memorias embebidas dentro de este. Aquí ha jugado un papel fundamental el autómata Aho-Corassick el cual ofrece la posibilidad de ser implementado como una arquitectura basada en memorias que procesa un caracter por ciclo. La ventaja del uso de memorias estriba en que la actualización del conjunto de cadenas es mucho más sencilla, puesto que sólo se requieren accesos de escritura en memoria y no la ejecución de un ciclo de reconfiguración total de la lógica del FPGA. Por otra parte la principal desventaja que enfrenta Aho-Corassick en FPGA es que procesar múltiples bytes por ciclo bajo se hace muy costoso.

Las demás arquitecturas como CAM, y comparadores discretos, tienen en común que la cadena se detecta en su totalidad en un solo ciclo de reloj, puesto que se trata de una ventana de búsqueda de tamaño igual a la cadena de la que se van leyendo todos sus elementos de forma paralela, pero a fin de no perder ocurrencias, el desplazamiento ha de ser de un byte por ciclo. Aquí identificamos otro posible frente de investigación también relacionado con la detección del desalineamiento de la cadena.

- *Sería viable encontrar una manera de aumentar este desplazamiento sin tener que replicar el hardware, eso se lograría si encontráramos una manera de saber si lo que se está leyendo actualmente en la ventana forma parte o no de la cadena que buscamos.*

Similarmente las arquitecturas que realizan *hash* sobre una ventana por la que se desplaza un flujo no pueden procesar múltiples bytes por ciclo sino replican el hardware utilizado, si bien con ellas se pueden asumir un gran número de cadenas, como en el caso de los filtros bloom, siempre existe el inconveniente de los falsos positivos. Solucionar estos inconvenientes ha sido el propósito de varios trabajos, en los que se proponen esquemas de *perfect-hashing*, donde generalmente se realiza hash como etapa de preprocesamiento.

El preprocesamiento de los datos de entrada ha sido otro frente importante de investigación de los últimos trabajos. Los esquemas de prefiltrado de cadenas, se presentan con dos objetivos fundamentales:

1. Reducir el volumen de cadenas en el flujo.
2. Configurar en un próximo paso la arquitectura de manera que responda a las posibles y no comprobadas existencias de cadenas de forma que esta pueda ser compartida temporalmente por un subconjunto pequeño del conjunto total de cadenas.

En este segundo punto divisamos otro posible frente de investigación no muy explorado, y que queremos establecer de la siguiente manera:

- *Es posible modificar o adaptar la arquitectura en tiempo real en dependencia del resultado de un filtrado previo, con la posibilidad de no contener en el interior del dispositivo el conjunto total de cadenas, sino un subconjunto de este de forma temporal, en el que sea posible que se encuentren las cadenas sugeridas por la fase de filtrado.*

El punto anterior tiene varias implicaciones ventajosas si se toma en cuenta que, primero, los recursos del FPGA son compartidos de manera temporal, segundo, la posibilidad de que el tamaño del conjunto de cadenas no sea limitado por la capacidad de la arquitectura, y por último, una visible integración entre sistemas software y arquitecturas hardware para el reconocimiento de cadenas.

11.2. Posibles áreas de investigación

Identificamos como posibles líneas de investigación para el desarrollo de arquitecturas óptimas las siguientes:

1. **Detección del desalineamiento:** El desalineamiento de las cadenas en arquitecturas multibytes es el responsable del incremento exponencial del costo de las mismas. Pensamos que en mucho se puede reducir el hardware si como parte de un preproceso logramos identificar la posición de la cadena en el flujo, y alinear la arquitectura de forma que que reconozca las cadenas convenientemente. Hemos encontrado al menos un resultado importante en este sentido [12], que demuestra la validez de esta estrategia.
2. **Preprocesamiento del conjunto de cadenas:** Como preprocesamiento del conjunto de cadenas, entenderemos una serie de acciones que realizaremos sobre este conjunto que nos permita, la posterior generación de una arquitectura óptima.
3. **Desarrollo de arquitecturas multibytes con reducido costo de hardware:** El desarrollo de arquitecturas multibytes, debe ser el punto principal de partida. Ha quedado bien claro que las limitantes tecnológicas no permiten elevadas capacidades de procesamiento si no es de esta forma. El objetivo se debe centrar entonces en reducir el costo de estas de forma que pueda aumentar el conjunto de cadenas detectables por una arquitectura.
4. **Prefiltrado del flujo de datos:** La mayoría de los diseños se conciben sólo para ser implementado en un dispositivo FPGA que reconozca todo el conjunto de cadenas que sea capaz de contener en su lógica configurable, aún cuando sólo se tiene a la entrada un conjunto bien reducido de estas temporalmente. Esto hace además que el número de cadenas reconocibles por la arquitectura sea dependiente de la capacidad del dispositivo.
   Pensamos que mucho se puede hacer si logramos que el dispositivo contenga sólo temporalmente un subconjunto pequeño de cadenas dentro del cual se encuentren las posibles cadenas de entrada. El principal reto y objetivo a superar en este sentido es, que la rapidez con que se debe actualizar la arquitectura no debe afectar la capacidad de procesamiento.

Referencias Bibliográficas

1. www.gocsi.com.
2. *Snort Users Manual 2.8.5*, 2009.
3. Alfred V. Aho and Margaret J. Corasick. Efficient string matching: an aid to bibliographic search. *Commun. ACM*, 18(6):333–340, 1975.
4. C. Allauzen and M. Raffinot. Factor oracle of a set of words, technical report. Technical report, Institut Gaspard-Monge, Université de Marne-Ia- Vallée, 1999.

5. Zachary K. Baker and Viktor K. Prasanna. A methodology for synthesis of efficient intrusion detection systems on fpgas. In *FCCM '04: Proceedings of the 12th Annual IEEE Symposium on Field-Programmable Custom Computing Machines*, pages 135–144, Washington, DC, USA, 2004. IEEE Computer Society.
6. Zachary K. Baker and Viktor K. Prasanna. Automatic synthesis of efficient intrusion detection systems on fpgas. *IEEE Trans. Dependable Secur. Comput.*, 3(4):289–300, 2006.
7. João Bispo, Ioannis Sourdis, João M. P. Cardoso, and Stamatis Vassiliadis. Regular expression matching for reconfigurable packet inspection. In *In Proceeding of Field Programmable Technology, FPT 2006. IEEE International Conference on Volume , Issue*, pages Page(s):119–126, December 2006.
8. Cristophe Bobda. *Introduction to Reconfigurable Computing, Architectures, Algorithms, and Applications*. Springer Berlin / Heidelberg, 2007.
9. Robert S. Boyer and J. Strother Moore. A fast string searching algorithm. *Commun. ACM*, 20(10):762–772, 1977.
10. Benjamin C. Brodie, David E. Taylor, and Ron K. Cytron. A scalable architecture for high-throughput regular-expression pattern matching. *SIGARCH Comput. Archit. News*, 34(2):191–202, 2006.
11. D. Caver, R. Frankling, and B.L. Hutching. Assisting network intrussion detection with reconfigurable hardware. In *FCCM'02*, 2002.
12. Yeim-Kuan Chang, Chen-Rong Chang, and Cheng-Chien Su. The cost effective pre-processing based nfa pattern matching architecture for nids. *Advanced Information Networking and Applications, International Conference on*, 0:385–391, 2010.
13. Christian Charras and Thierry Lecroq. *Handbook of Exact String Matching Algorithms*. King's College Publications, 2004.
14. Young H. Cho and William H. Mangione-Smith. Fast reconfiguring deep packet filter for 1+ gigabit network, 2005.
15. Young H. Cho, Shiva Navab, and William H. Mangione-Smith. Specialized hardware for deep network packet filtering. In *FPL '02: Proceedings of the Reconfigurable Computing Is Going Mainstream, 12th International Conference on Field-Programmable Logic and Applications*, pages 452–461, London, UK, 2002. Springer-Verlag.
16. Christopher R. Clark and David E. Schimmel. *Field-Programmable Logic and Applications*, chapter Efficient Reconfigurable Logic Circuits for Matching Complex Network Intrusion Detection Patterns, pages 956–959. LNCS. Springer Berlin / Heidelberg, 2003.
17. Christopher R. Clark and David E. Schimmel. Scalable pattern matching for high speed networks. *Field-Programmable Custom Computing Machines, Annual IEEE Symposium on*, 0:249–257, 2004.
18. M. Crochemore, A. Czumaj, L. Gasieniec, S. Jarominek, T. Lecroq, W. Plandowski, and W. Rytter. Speeding up two string-matching algorithms. *Algorithmica*, 12:247–267, 1994. 10.1007/BF01185427.
19. Maxime Crochemore and Wojciech Rytter. *Jewels of Stringology*. World Scientific, 2003.
20. Sarang Dharmapurikar, Michael Attig, and John Lockwood. Desing and implementation of a string matching system for network intrussion detection using fpga-based bloom filters. Technical report, Whasington University, 2004.
21. S. Golson. State machine desing techniques for verilog and vhdl. *Synopsys Journal of High Level Design*, pages 1–48, September 1994.
22. Nitesh B. Guinde. *Information Security Theory and Practices. Security and Privacy of Pervasive Systems and Smart Devices*, chapter Novel FPGA-Based Signature Matching for Deep Packet Inspection, pages 261–276. LNCS. Springer Berlin / Heidelberg, 2010.
23. John E. Hopcroft. *Introduction to Automata Theory, Languages, and Computation*. Pearson Addison Wesley, 2007.
24. Hong-Jip Jung, Zachary K. Baker, and Viktor K. Prasanna. Performance of fpga implementation of bit-split architecture for intrusion detection systems. In *In Prpceedins of the Reconfigurable Architectures Workshop at IPDPS (RAW â06)*, 2006.
25. Donald E. Knuth, Jr. James H. Morris, and Vaughan R. Pratt. Fast pattern matching in strings. *SIAM Journal on Computing*, 6(2):323–350, 1977.
26. Gad Landau, Cyril Allauzen, Maxime Crochemore, and Mathieu Raffinot. Efficient experimental string matching by weak factor recognition. In *Combinatorial Pattern Matching*, volume 2089 of *Lecture Notes in Computer Science*, pages 51–72. Springer Berlin / Heidelberg, 2006. 10.1007/3-540-48194-X_5.
27. Cheng-Hung Lin, Chih-Tsun Huang, Chang-Ping Jiang, and Shih-Chieh Chang. Optimization of regular expression pattern matching circuits on fpga. In *DATE '06: Proceedings of the conference on Design, automation and test in Europe*, pages 12–17, 3001 Leuven, Belgium, Belgium, 2006. European Design and Automation Association.
28. Wei Lin and Bin Liu. Pipelined parallel ac-based approach for multi-string matching. In *ICPADS '08: Proceedings of the 2008 14th IEEE International Conference on Parallel and Distributed Systems*, pages 665–672, Washington, DC, USA, 2008. IEEE Computer Society.
29. Jan Van Lunteren. High-performance pattern-matching for intrusion detection. In *Proceedings of IEEE INFOCOM'06*, 2006.
30. Abhishek Mitra, Walid Najjar, and Laxmi Bhuyan. Compiling pcre to fpga for accelerating snort ids. In *ANCS '07: Proceedings of the 3rd ACM/IEEE Symposium on Architecture for networking and communications systems*, pages 127–136, New York, NY, USA, 2007. ACM.
31. J. H. Morris and V. R. Pratt. A linear pattern-matching algorithm. Technical Report 40, University of California, Berkeley, 1970.

32. James Moscola, John Lockwood, Ronald P. Loui, and Michael Pachos. Implementation of a content-scanning module for an internet firewall. In *FCCM '03: Proceedings of the 11th Annual IEEE Symposium on Field-Programmable Custom Computing Machines*, page 31, Washington, DC, USA, 2003. IEEE Computer Society.
33. Gonzalo Navarro and Mathieu Raffinot. *Flexible Pattern Matching in Strings*. Cambridge University Press, 2002.
34. R. Pagh and F. F. Rodler. Cuckoo hashing. *Journal of Algorithms*, 51:pp. 122–144, 2004.
35. Reetinder Sidhu and Viktor K. Prasanna. Fast regular expression matching using fpgas. In *FCCM '01: Proceedings of the the 9th Annual IEEE Symposium on Field-Programmable Custom Computing Machines*, pages 227–238, Washington, DC, USA, 2001. IEEE Computer Society.
36. Benfano Soewito, Lucas Vega, Atul Mahajan, Ning Weng, and Haibo Wang. Self-addressable memory-based fsm: A scalable instrusion detection engine. *IEEE Network*, 23, issue:1:14–21, March 2009.
37. Ioannis Sourdis. Efficient and high-speed fpga-based string matching for packet inspecction. Master's thesis, Technical University of Crete, july 2004.
38. Ioannis Sourdis, João Bispo, João M. P. Cardoso, and Stamatis Vassiliadis. Regular expression matching in reconfigurable hardware. *Journal of Signal Processing Systems*, Vol. 51(No. 1):99–121, April 2008.
39. Ioannis Sourdis, Vasilis Dimopoulos, Dionisios Pnevmatikatos, and Stamatis Vassiliadis. Packet pre-filtering for network intrusion detection. In *ANCS '06: Proceedings of the 2006 ACM/IEEE symposium on Architecture for networking and communications systems*, pages 183–192, New York, NY, USA, 2006. ACM.
40. Ioannis Sourdis and Dionisios Pnevmatikatos. Fast, large-scale string match for a 10gbps fpga-based network intrusion. *FPL*, 2003:880–889, 2003.
41. Ioannis Sourdis and Dionisios Pnevmatikatos. Pre-decoded cams for efficient and high-speed nids pattern matching. In *In IEEE Symposium on Field-Programmable Custom Computing Machines*, pages 258–267. IEEE, 2004.
42. Ioannis Sourdis, Dionisios Pnevmatikatos, Stephan Wong, and Stamatis Vassiliadis. A reconfigurable perfect-hashing scheme for packet inspection. In *in Proceedings of 15th Int. Conf. on Field Programmable Logic and Applications*, pages 644–647, 2005.
43. Wun Su and Udi Manber. A fast algorithm for multi-pattern searching. 1994.
44. Tinh Ngoc Tran and Surin Kittitornkun. Fpga-based cuckoo hashing for pattern matching in nids/nips. In *Proceedings of APNOMS 2007*, volume 4773 of *LNCS*, pages pp. 334–343, 2007.
45. Xilinx. *Spartan-3 FPGA Family Complete Data Sheet*, january 2005.
46. Xilinx. *Using Look-Up Tables as Shift Registers (SLR16) in Spartan-3 Generation FPGAs*, 2005.
47. Xilinx. *Virtex-5 FPGA User Guide*, january edition, 2009.
48. Yi-Hua E. Yang and Viktor K. Prasanna. Automatic construction of large scale regular expression engines on fpga. In *In Procedings of International Conference on Reconfigurable Computing and FPGAs*. IEEE Computer Society, 2008.
49. Fang Yu, Zhifeng Chen, Yanlei Diao, T. V. Lakshman, and Randy H. Katz. Fast and memory efficient regular expression matching for deep packet inspection. In *ANCS'06*, 1-59593-580-0/06/0012, December 3-5 2006.
50. S. Yusuf and W. Luk. Bit-wise optimized cam for network intrusion detection system. In *Field Programmable Logic and Applicacions*, August 2005.

RT_015, octubre 2011
Aprobado por el Consejo Científico CENATAV

Editor: Lic. Lucía González Bayona
Diseño de Portada: Di. Alejandro Pérez Abraham
RNPS No. 2143
ISSN 2072-6260
Indicaciones para los Autores:
Seguir la plantilla que aparece en www.cenatav.co.cu
C E N A T A V
7ma. No. 21812 e/218 y 222, Rpto. Siboney, Playa;
La Habana. Cuba. C.P. 12200
Impreso en Cuba