UNIVERSIDAD DE CONCEPCIÓN

FACULTAD DE INGENIERÍA
DEPARTAMENTO DE INGENIERÍA ELÉCTRICA

ACELERADOR HARDWARE PARA ESTIMACIÓN DE CUANTILES EN TRÁFICO DE REDES

POR

Carolina Sofía Gallardo Pavesi

Memoria de Título presentada a la Facultad de Ingeniería de la Universidad de Concepción para optar al título profesional de Ingeniero(a) Civil Electrónico(a)

Profesores Guía:

Dr. Miguel Figueroa T.

Dra. Cecilia Hernández R.

Abril 2024
Concepción - Chile

"We shall not cease from exploration

And the end of all our exploring

Will be to arrive where we started

And know the place for the first time."

T.S. Elliot

Agradecimientos

Al final de esta etapa me gustaría agradecer a esas personas que han sido importantes en mi vida:

- En primer lugar, quiero agradecer a mis papás. Gracias por todo el apoyo, el amor incondicional y cada detallito que hizo este proceso un poco más fácil. No podría haberlo hecho sin ustedes. Los amo un montón.

- A mis hermanos, quienes me han visto crecer y siempre han estado ahí para mí de una manera u otra, alentándome a cada paso.

- A mis abuelos, por mostrarme que con esfuerzo y perseverancia se puede lograr cualquier cosa. Espero estén descansando en algún lugar bonito.

- A mis amigos de siempre... Catita: tú sabes lo que significas para mí, eres y siempre has sido una luz en mi vida, gracias por siempre entenderme cuando estaba ocupada, y por recibirme como si nada cuando nos volvíamos a ver después de harto tiempo. Tenemos muchos momentos hermosos juntas y espero se vuelvan muchos más. Cris y Leo: fue un gusto compartir este camino con ustedes y poder apoyarnos mutuamente. Definitivamente, las cosas eran mejores cuando trabajábamos juntos. Gracias por las risas, las horas de clases compartidas, las papitas del Assuan e incluso los momentos de estrés vividos juntos. Les deseo lo mejor siempre.

- A los amigos nuevos que me regaló el lab de VLSI, Wladi, Sofi, Mónica, Juan Felipe y Bárbaro. Pero sobre todo, a Yaime, por ser mi partner incondicional estos años, por alentarme e intentar cuidarme siempre y a Javier, por siempre estar dispuesto a tenderme una mano cuando lo necesité.

- A mis tíos, a mis colegas de Zembia y a todo aquel que fomentó mi curiosidad y me hizo enamorarme más de lo que hago y querer aprender más.

- A mis profesores, Miguel Figueroa y Cecilia Hernández. Me siento muy honrada de trabajar con ustedes, los admiro mucho. Gracias por guiarme, apoyarme en cada paso y confiar en mí incluso más de lo que yo lo hago, inspirándome a ser mejor y a crecer todo lo que pueda.

- Y por último, pero definitivamente no menos importante, a mi pololo, mi Martincito. Gracias por ser mi serendipia, por siempre estar allí para apoyarme y abrazarme cuando lo necesito, por aguantarme y apañarme en los días malos y darles siempre un toque de alegría. Gracias por amarme como soy, creer en mí, escucharme y hacerme reír. Te amo.

Resumen

Las propiedades estadísticas del tráfico, tales como los cuantiles, entre los cuales se encuentran la mediana y el percentil 99, y la función de distribución acumulativa, nos ayudan a entender la distribución del tráfico y detectar anomalías a corto y largo plazo. Sin embargo, calcular de manera exacta el valor de estas propiedades es costoso en términos de espacio y tiempo. Particularmente, en el caso de los percentiles distintos de 0 y 100, es imposible encontrar una solución exacta en una pasada sin almacenar todos los datos. Utilizar un acelerador hardware basado en arreglos de lógica programable (FPGA, del inglés Field Programmable Gate Array) nos permite enfrentar la necesidad de procesar un stream de paquetes a alta velocidad y producir resultados con baja latencia. Sin embargo, la memoria interna disponible en estos dispositivos es limitada, por lo que almacenar todos los elementos no es una opción viable. Para responder a este problema, se han propuesto algoritmos basados en sketches: estructuras probabilísticas compactas que utilizan un espacio de memoria sublineal para estimar una propiedad con una cota de error predeterminada.

Este trabajo presenta la arquitectura de un acelerador hardware para la estimación de cuantiles usando un algoritmo basado en sketches. La arquitectura propuesta se basa en la estructura general del sketch KLL presentado por Karnin et al. [1]. Implementamos la arquitectura en una tarjeta de desarrollo Xilinx Alveo U55C, la cual contiene un FPGA Virtex™ XCU55 UltraScale+, obteniendo un tiempo de ejecución total por lo menos 35 veces menor que el tiempo de ejecución total de la implementación en software del algoritmo modificado, y una aceleración mínima de 37 veces en cuanto al tiempo de inserción de elementos al sketch. El acelerador puede operar a una frecuencia de reloj de máxima 325.2 MHz, lo que equivale a una tasa mínima de 166 Gbps (gigabits por segundo). Mostramos empíricamente que las estimaciones mantienen una buena precisión, respetando las cotas de error cuando el número de paquetes de la traza es lo suficientemente parecido al número de elementos para el que dimensionamos el sketch.

Summary

Statistical properties of traffic, like quantiles (e.g. the median, the 99th percentile) and the cumulative distribution function (CDF), can help us better understand the distribution of network traffic and detect short-term and long-term anomalies. However, computing the exact value of these properties is expensive both in execution time and storing space. In particular, it is impossible to find the exact value of a percentile different from 0 and 100 in a one-pass fashion without storing the totality of incoming elements. Using a hardware accelerator based on Field Programmable Gate Arrays (FPGA) allows us to face the challenge of processing a stream of packets at high-speed while producing results with low latency. Nevertheless, the internal memory available in these devices is highly limited, so storing all elements is not feasible. To respond to this problem, numerous algorithms using sketches (compact probabilistic data structures with sublineal memory usage that allow us to estimate the value of a property with a pre-determined error bound) have been proposed.

This work presents the architecture of a hardware accelerator for quantile estimation using a sketch-based algorithm. The proposed architecture is inspired by the general structure of the KLL sketch presented by Karnin et al. [1]. We implement this architecture on a Xilinx Alveo U55C board, which features a Virtex™ XCU55 UltraScale+ FPGA, obtaining a total execution time at least 35 times shorter than the total execution time of the software implementation of the modified algorithm. Also, we obtain a minimal speed up of 37 times in element insertion time. The accelerator can operate at a maximum clock frequency of 325.2 MHz, which translates to a minimal line-rate of 166 Gbps (gigabits per second). We show empirically that the estimations maintain a good precision, staying within the error bounds when the number of packets of the trace is sufficiently close to the number of packets used to calculate the parameters of the sketch.

Tabla de contenidos

Lista de figuras

Lista de tablas

Lista de algoritmos

Capítulo 1. Introducción

1.1. Introducción general

La medición y análisis de tráfico en redes de datos de alta velocidad es una tarea importante dado que nos permite monitorear el estado de la red y detectar posibles fallas o amenazas a la seguridad de esta [2]. Durante este análisis, se busca obtener métricas que describan el comportamiento estadístico del tráfico medido, permitiéndonos detectar cambios anómalos en el comportamiento de la red.

Una de las métricas que nos ayudan a describir el comportamiento del tráfico son los cuantiles. Un cuantíl p es el valor x para el cual, dado un conjunto de elementos $x_1, ..., x_n$, $Pr[x_i \leq x] = p$. El valor de estos cuantiles, entre los cuales se encuentran la mediana y el percentil 99, en conjunto con la función de distribución acumulativa, nos ayudan a entender la distribución del tráfico y detectar tanto anomalías a corto plazo como cambios a largo plazo en la distribución [3]. Por ejemplo, un cambio brusco en la distribución del tamaño de los paquetes o un aumento en el porcentaje de paquetes que provienen de un rango de direcciones IP podría indicar un comportamiento anómalo de la red.

La medición y análisis en línea del tráfico de red requiere procesar un stream de paquetes a alta velocidad y producir resultados con baja latencia para poder tomar acciones correctivas de ser necesario [4]. Debido a los grandes volúmenes de tráfico y la velocidad de los enlaces modernos, esto se vuelve cada vez más costoso computacionalmente [5], por lo que un procesador de propósito general se vuelve insuficiente para abordar el problema. Los switches programables de alto desempeño poseen procesadores especializados que nos permiten abordar algunas de estas tareas. Sin embargo, los modelos de programación que estos utilizan limitan los algoritmos que se pueden ejecutar de manera eficiente en ellos [6]. Considerando esto, los aceleradores hardware de propósito específico se vuelven una opción atractiva.

Uno de los principales tipos de aceleradores hardware son aquellos basados en dispositivos FPGA (Field-Programmable Gate Array) los cuales contienen bloques de lógica cuya interconexión y

funcionalidad se puede programar múltiples veces. Estos dispositivos se caracterizan por su alta capacidad máxima de cómputo y flexibilidad debido a que poseen una gran cantidad de lógica programable. Además, su alta frecuencia de operación y su paralelismo de grado fino hacen que sea posible ejecutar algoritmos de manera más rápida y eficiente que en un procesador de propósito general [7]. No obstante, el desempeño real de los FPGA se ve frecuentemente limitado por la memoria disponible en el chip y por el desbalance entre su capacidad máxima de cómputo y su ancho de banda a memoria externa [8, 9]. Esto se vuelve un desafío, ya que el espacio necesario para calcular de manera exacta el valor de las propiedades importantes de los datos, entre las cuales se encuentran la entropía, los cuantiles y la cardinalidad, supera el espacio disponible en estos dispositivos. Particularmente, en el caso de los percentiles distintos de 0 y 100, es imposible encontrar una solución exacta en una pasada sin almacenar todos los datos [10]. Afortunadamente, para muchas aplicaciones es suficiente trabajar con estimaciones de estas propiedades, las que se pueden calcular sin incurrir en el alto costo de determinar los valores exactos [2].

Se vuelve necesario entonces utilizar métodos que nos permitan estimar las propiedades del tráfico, reduciendo la cantidad de memoria necesaria y manteniendo una buena precisión de la estimación. Frente a este problema, se introducen dos ideas principales: muestreo y sketches. El muestreo consiste en seleccionar un subconjunto representativo de los datos originales, a partir del cual podamos inferir las características del conjunto completo. Esta técnica es eficiente en términos de tiempo de procesamiento y consume una cantidad baja de memoria, además de ser fácil de implementar ya que solo requiere definir algún criterio para selecionar 1 de cada k elementos, donde k es la tasa de muestreo. Sin embargo, el método es sensible a valores atípicos y al sesgo en los datos, necesitando una tasa de muestreo alta para obtener una buena precisión, lo que consume una cantidad importante de recursos [2, 11]. Por otro lado, un sketch es una estructura probabilística compacta que utiliza un espacio de memoria sublineal para estimar una propiedad con una cota de error predeterminada. Los algoritmos basados en sketches utilizan todos los datos de entrada para obtener una estimación precisa con un costo de memoria acotado [2, 7]. Además, estos algoritmos son especialmente apropiados para datos obtenidos por streaming debido a que procesan los datos de entrada en una sola pasada sin necesidad de almacenarlos todos. Adicionalmente, en muchos casos, los algoritmos basados en sketches exhiben un alto grado de paralelismo intrínseco.

Sin embargo, muchos algoritmos basados en sketches requieren de la ejecución de tareas computacionalmente caras, como por ejemplo la ejecución de funciones Hash, por lo que al implementarlos en procesadores de propósito general, no alcanzarían el flujo necesario para aplicaciones en tiempo real.

Considerando todo lo mencionado anteriormente, este trabajo propone una arquitectura basada en un algoritmo de estimación de cuantiles usando sketches. La arquitectura está basada en la estructura general del sketch KLL (sigla que proviene del apellido de los autores del paper que lo presenta [1]: Karning, Lang y Liberty). La arquitectura fue implementada en una tarjeta de desarrollo Alveo U55C, la cual contiene un FPGA Virtex™ XCU55 UltraScale+ y la precisión de las estimaciones fue comprobada empíricamente para 10 trazas de red que contienen datos reales y fueron obtenidas de repositorios públicos [12, 13, 14, 15, 16].

Este capítulo presenta los objetivos del trabajo, sus alcances y limitaciones. Luego, el capítulo 2 presenta un análisis bibliográfico de algoritmos basados en sketches para la estimación de cuantiles y de aceleradores hardware que usan algoritmos basados en sketches, para finalmente discutir las conclusiones de esta revisión. El capítulo 3 presenta la descripción y funcionamiento del algoritmo en el cual basamos nuestra arquitectura, el sketch KLL. En el capítulo 4, se presenta la descripción de la arquitectura del acelerador hardware, para luego escoger los parámetros que usamos en nuestra implementación en el capítulo 5. Para terminar, el capítulo 6 presenta los resultados obtenidos por la arquitectura, tanto en precisión de las estimaciones como en uso de recursos y frecuencia de reloj.

1.2. Objetivos

1.2.1. Objetivo general

Diseñar un acelerador hardware para estimación de cuantiles de propiedades de flujos de redes usando algoritmos basados en sketches.

1.2.2. Objetivos específicos

- Diseñar la arquitectura del acelerador hardware explotando el paralelismo disponible en el algoritmo.
- Escribir una implementación RTL del acelerador.
- Implementar y validar el acelerador sobre un dispositivo FPGA.
- Utilizar el acelerador para estimar propiedades estadísticas del tráfico.

1.3. Alcances y limitaciones

- Se utilizarán para la implementación los dispositivos disponibles en el laboratorio de VLSI.
- No se utilizará tráfico en línea para realizar la validación del acelerador, sino que se probará usando trazas de tráfico disponibles en repositorios públicos.

1.4. Productos y resultados esperados

- Informe de memoria de título explicando los procedimientos de diseño, implementación y prueba del acelerador.
- Un algoritmo con paralelismo de grado fino explotable.
- El diseño de la arquitectura de un acelerador hardware para estimación de cuantiles.
- La implementación de esta arquitectura sobre un dispositivo FPGA.

Capítulo 2. Análisis bibliográfico

2.1. Algoritmos basados en sketches para estimación de cuantiles

Dado que encontrar el valor exacto de un cuantíl requiere espacio y tiempo $O(n)$, el problema de la estimación de cuantiles ha generado interés en los ultimos años y se han publicado numerosos trabajos que intentan resolver este problema mejorando su precisión o reduciendo su costo en memoria.

En la publicación pionera sobre este tema, Munro y Paterson [10] demostraron que, en un contexto de streaming, para calcular una mediana en p pasadas sobre los datos se necesita un espacio de $\Omega(n^{1/p})$. Además, proponen un algoritmo iterativo para encontrar la mediana, cuya probabilidad de fallo es menor a ε y utiliza un espacio de $\Omega(n^{1/2p})$.

Ocho años más tarde, Manku et al. [17] proponen un algoritmo basado en [10]. Este algoritmo consiste en b buffers de tamaño k. A cada buffer se le asocia un entero $l(X)$ que representa su nivel. Sea l el menor de los niveles de los buffers actualmente llenos. Si hay al menos dos buffers vacíos, estos se llenan con los elementos del stream, y se les asigna el nivel 0. Si hay exactamente un buffer vacío, este se llena y se le asigna el nivel l. Si no hay buffers vacíos, entonces se colapsan los buffers de nivel l y se le asigna al buffer de salida un nivel $l + 1$. El peso de este buffer de salida es la suma de los pesos de los buffers colapsados. Este algoritmo determinístico, conocido como MRL98 (del apellido de los autores Manku, Rajagopalan y Linsday, y el año de publicación 1998), logra encontrar un valor aproximado en ε del cuantil Φ (ε-approximate quantile). Si δ es la probabilidad de que el error aditivo supere ε y n es el número de elementos del stream, entonces $b = O(\log \varepsilon n)$ y $k = O(\frac{1}{\varepsilon} \log \varepsilon n)$, obteniendo una complejidad espacial de $O(\frac{1}{\varepsilon} \log^2(\varepsilon n))$. Esta complejidad espacial es similar a la de [10] pero la constante por la que se multiplica la complejidad es menor. Además, los autores proponen la opción de utilizar muestreo, obteniendo una complejidad espacial de $O(\frac{1}{\varepsilon} \log^2(\frac{1}{\varepsilon} \log \frac{1}{\delta}))$.

En una continuación de este trabajo, los autores publicaron una mejora al algoritmo, MRL99

[18]. Este nuevo algoritmo, a diferencia de los descritos anteriormente, no requiere conocimiento previo del número de elementos del stream, lo cual es útil en aplicaciones realistas. Este algoritmo combina el MRL original con una técnica de muestro no-uniforme, es decir, no todos los elementos tiene la misma probabilidad de ser seleccionados. Ahora, en vez de insertar directamente cada elemento en un buffer, se selecciona aleatoriamente 1 de cada r elementos, donde r es la tasa de muestreo. Usando este algoritmo se mantiene la complejidad espacial obtenida en el trabajo anterior, pero se elimina la dependencia del conocimiento del tamaño del stream.

Greenwald y Khanna [19] presentan un algoritmo determinístico con complejidad espacial de $O(\frac{1}{\varepsilon}\log \varepsilon n)$. Este algoritmo permite tener cotas superiores e inferiores para cada cuantil en vez de una sola para todos los cuantiles. El algoritmo está basado en estructuras llamadas "summaries". Cada una de estas estructuras almacena una tupla que contiene 3 valores: v_i, un valor que corresponde a uno de los elementos de la secuencia vista hasta ahora, g_i, la diferencia entre $r_{min}(v_i)$ y $r_{min}(v_i - 1)$ y Δi, la diferencia entre $r_{max}(v_i)$ y $r_{min}(v_i)$, donde r_{min} y r_{max} son, respectivamente, la cota inferior y superior del rank de v_i en los elementos vistos hasta ahora. Esta estructura puede utilizarse para estimar cuantiles con un error máximo de $\frac{max_i(g_i+\Delta i)}{2}$.

Más tarde, Felber y Ostrovsky [20] proponen combinar estas estructuras con un sampler de tipo Bernouilli donde cada elemento se escoge con una probabilidad de m/n, donde n es el número de elementos. Para evitar tener que saber el número de elementos del stream con antelación, se trabaja con varias filas, donde cada fila r tiene una copia del sketch GK y es un resumen de los primeros $2^r 32m$ elementos del stream. Además, cada fila $r \geq 1$ tiene un sampler Bernouilli. Cada vez que llega $\frac{1}{64}$ de los elementos que representará una fila $r \geq 1$, esta se activa y recibe un resumen de los elementos vistos anteriormente construido con los elementos almacenados en la fila de abajo. Este resumen se construye consultando por $\frac{8}{\varepsilon}$ cuantiles y duplicando estos $m\frac{8}{\varepsilon}$ veces. Una vez que esto pasa, la fila de abajo ya no se necesita y se puede eliminar. Este sketch disminuye la complejidad espacial a $O(\frac{1}{\varepsilon}\log\frac{1}{\varepsilon})$. Sin embargo, este sketch no permite una unificación completa entre estructuras (operación de unión).

El trabajo de Agarwal et al. [21] propone un tipo de resumen que soporta la operación de unión. La estructura se construye en capas, donde cada capa contiene un resumen con exactamente $k_\varepsilon =$

$O(\frac{1}{\varepsilon}\sqrt{\log \frac{1}{\varepsilon\delta}})$ elementos. El número de capas es igual a $\log \frac{n}{k_\varepsilon}$. Partiendo por la capa de abajo, si la capa se llena, se escogen los elementos con indice par o impar de manera aleatoria y se insertan en la capa inmediantamente superior con el doble de peso. Este elemento aleatorio es la principal contribución al trabajo de Manku et al. [18], y hace que el error esperado sea 0, ya que se sobreestima o subestima el valor del rank de un elemento con igual probabilidad. Este algoritmo tiene una complejidad espacial de $O(\frac{1}{\varepsilon}\log^{3/2}\frac{1}{\varepsilon})$.

En 2016, Karnin et al. [1] proponen el sketch KLL. Este sketch toma el trabajo propuesto en [18] y [21] y hace algunas modificaciones. El algoritmo tiene como elemento principal una estructura llamada compactor, la cual puede almacenar k_h elementos con un peso w. El valor máximo de k_h es k_H y está dado por $k_H = O(\frac{1}{\varepsilon}\log\log(\frac{1}{\varepsilon\delta}))$. También puede compactar sus k_h elementos a $k_h/2$ elementos. Para lograr esto, se ordenan en primer lugar los elementos. Luego, se escogen aleatoriamente los elementos con posiciones pares o impares, y el resto son descartados. El peso de los elementos escogidos es multiplicado por 2. Adicionalmente, el algoritmo propuesto usa distintas capacidades para compactors en distintas alturas. El tamaño puede, por ejemplo, decrecer exponencialmente en una constante c a medida que la altura disminuye dejando $k_h = k_H c^{H-h-1}$. Por otro lado, la capacidad mínima de un compactor es de 2 elementos, lo que puede contribuir $O(H) = O(\log n/k)$ a la complejidad espacial, donde H es la cantidad máxima de compactors para un stream de largo n. Para evitar esto, se pueden remplazar estos compactors por un sampler que cumpla la misma función. Adicionalmente, el sketch KLL propone que los $\log\log(\frac{1}{\delta\varepsilon})$ compactors superiores mantengan el tamaño máximo debido a que estos tienen la mayor influencia sobre la estimación de cuantiles. Esta estructura soporta operaciones de unión (merge) y tiene una complejidad espacial menor a los algoritmos propuestos anteriormente: $O(\frac{1}{\varepsilon}\log^2\log\frac{1}{\delta\varepsilon})$. Los autores hacen una última mejora remplazando los compactors superiores por un GK Sketch, esto disminuye la complejidad espacial a $O(\frac{1}{\varepsilon}\log\log\frac{1}{\varepsilon\delta})$ pero se pierde la posibilidad de hacer operaciones de unión entre estructuras.

Existen otros trabajos recientes que usan enfoques distintos para la estimación de cuantiles. Por ejemplo, el Moments Sketch propuesto por Gan, et al [22] almacena el mínimo x_{min}, el máximo x_{max}, k momentos μ y k momentos logarítmicos ν del dataset, donde k es un número pequeño, usualmente entre 10 y 22, y es llamado el orden del sketch. Dado un dataset D, el momento μ_i se define como $\frac{1}{n}\sum_{x\in D} x^i$ y el momento logarítmico ν_i equivale a $\sum_{x\in D}\log^i(x)$, donde n es el número de elemen-

tos del dataset e $i \in 1, ..., k$. Con esta información, el sketch selecciona la distribución $p(x)$ que se corresponda con los datos utilizando el Método de Momentos. Sin embargo, puede haber más de una distribución que se corresponda con estos datos, por lo que se necesita un criterio de selección. Este trabajo propone utilizar el principio de máxima entropía para seleccionar la solución final, el cual afirma que se debe seleccionar la distribución menos informativa que sea consistente con los datos observados. La distribución seleccionada puede ser utilizada posteriormente para estimar cualquier cuantíl.

DDSketch (Distributed Distribution Sketch) es un algoritmo determinístico basado en histogramas propuesto por Masson, et al. [23] en 2018. El sketch está formado por buckets que cuentan el número de elementos observados en el stream que se encuentran entre $(\gamma^{i-1}, \gamma^i]$ donde γ depende del error relativo máximo α y se calcula usando la siguiente relación: $\gamma = \frac{1+\alpha}{1-\alpha}$. Cada elemento x es mapeado a un bucket específico i, donde i es el resultado de $\lceil \log_\gamma(x) \rceil$. Debido a esto, el histograma puede manejar un amplio rango de valores en un número reducido de buckets.

Epicopo, et al [24] proponen el Uniform DDSketch o UDDSketch, una modificación del DDSketch descrito anteriormente. Su principal diferencia radica en como se realiza la operación de extensión de rango, colapsando todos los pares de buckets adyacentes. Este cambio hace que el deterioro del error relativo sea más uniforme que en el DDSketch con un número fijo de buckets.

El sketch Relative Error Quantile Sketch o ReqSketch propuesto por Cormode et al. [25] es similar al sketch KLL en que almacena una muestra de los datos observados en una serie de compactors llamados compactors-relativos, los cuales tienen un tamaño $B = 2k\lceil \log \frac{n}{k} \rceil$ donde k es un entero par. A diferencia del KLL, al realizar la operación de compactación en un nivel h, solo se consideran los L elementos más grandes almacenados en el buffer (L siempre es menor a la mitad del tamaño del compactor), de los cuales se seleccionan aleatoriamente los elementos de índice par o impar del compactor y se promueven al nivel $h + 1$. Los elementos no considerados dentro de los L elementos mayores se mantienen en el buffer de nivel h. Adicionalmente, el número de elementos que se consideran en cada compactor cambia según un cronograma de compactación. Para determinar L, se divide la mitad superior del compactor en secciones de largo k y se numeran de manera tal que la primera sección contenga los elementos más grandes. El trabajo propone lo siguiente: un segmento i se verá implicado

en una de cada 2^{i-1} compactaciones. Esto se hace para que los elementos mayores de un buffer sean compactados con mayor frecuencia que los menores. ReqSketch tiene una complejidad espacial de $O(\frac{\log^{1,5} \varepsilon n}{\varepsilon})$ y garantiza la siguiente cota de error: $|\hat{R}(x) - R(x)| \leq \varepsilon R(x)$.

Por otro lado, en 2022, Ivkin et al. [26] proponen KLL Sweep, una modificación del sketch KLL que reduce el error máximo a la mitad y baja la latencia de peor caso de $O(\frac{1}{\varepsilon})$ a $O(\log \frac{1}{\varepsilon})$. En este trabajo se sugiere que todos los compactors compartan la memoria disponible y se haga una compactación solo cuando ésta esté saturada. De esta manera, se reduce el número de compactaciones, ya que cada compactación se aplica a un número potencialmente más grande de elementos que en el algoritmo original.

Teniendo en cuenta el analísis del estado del arte que acabamos de realizar, podemos concluir que basarnos en el sketch KLL para la estimación de cuantiles es una buena opción por las siguientes razones: En primer lugar, su complejidad espacial no depende del número de elementos del stream y es menor a la de los algoritmos propuestos anteriormente. Por otro lado, las estructuras utilizadas por este algoritmo soportan operaciones de unión (merge) completas entre ellas, es decir, si construimos dos sketches a partir de dos sets de datos y realizamos una operación de unión, se pueden obtener estimaciones sobre la unión de ambos sets de datos sin perder precisión. Esta posibilidad nos permite explotar de mejor manera el paralelismo disponible en los dispositivos FPGA. Finalmente, al usar este sketch tenemos una cota de error aditivo concreta, por lo que podemos determinar los parámetros a usar en función de la precisión deseada.

2.2. Aceleradores hardware para algoritmos basados en sketches

La aceleración hardware consiste en utilizar distintas plataformas basadas en hardware, tales como dispositivos FPGA y switches programables, para obtener una ejecución más rápida de ciertos algoritmos con respecto al tiempo de ejecución en un procesador de propósito general. Estas plataformas son dedicadas y realizan procesamiento paralelo, obteniendo un alto rendimiento con baja

consumo de energía [27]. Sin embargo, como mencionamos anteriormente, la capacidad de procesamiento se ve limitada por el ancho de banda a memoria. Es por esto que varias investigaciones buscan aprovechar los algoritmos basados en sketches para diseñar arquitecturas que permitan acelerar la estimación de ciertas propiedades.

Los dos tipos principales de aceleradores hardware son los basados en switches programables y los basados en Field Programmable Gate Arrays.

2.2.1. Aceleradores hardware en switches programables

Actualmente, existen switches programables basados en ASICs (Application Specific Integrated Circuits), los cuales permiten reconfigurar el comportamiento de su plano de datos. Estos switches utilizan un lenguaje de programación de alto nivel llamado P4, el cual describe como son procesados los paquetes por el plano de datos. Además, le permiten al programador generar las interfaces que el plano de control puede usar para configurar el plano de datos o comunicarse con este. Aprovechando esta posibilidad, se desarrollaron varios trabajos que implementan algoritmos basados en sketches en switches programables. Hay dos trabajos principales que implementan la estimación de cuantiles en el plano de datos, de los cuales solo uno está basado en sketches:

En 2019, Ivkin et al. proponen QPipe [3], una modificación del sketch KLLSweep [26] implementable en el plano de datos de un switch programable. Mantiene la idea de utilizar una técnica de muestro, pero usa los paquetes no seleccionados como "trabajadores", para llevar algunos valores y finalizar algunas operaciones necesarias para mantener la estructura de datos. La implementación logra una mejora de 91.09 veces en cuanto al error máximo de aproximación que se obtendría con una técnica de muestro bajo las mismas condiciones de memoria.

En 2023, Wang et al. presentan un algoritmo para la estimación del valor de un cuantíl p en particular, el cual debe conocerse con anterioridad. La idea se centra en estimar el valor de $\hat{q}$ incrementandolo o decrementandolo en un valor λ_n. El sketch tiene dos modos de operación. El modo "average" se usa cuando el cuantíl es bajo, y usa un $\lambda_n = 2\frac{avg_n}{n-1}$, donde avg_n es el promedio de los

datos en el stream. Por otro lado, en el modo "max-min", $\lambda_n = \frac{x_{max}-x_{min}}{n-1}$. Este modo se usa cuando el cuantíl buscado es alto. Cuando llega un nuevo elemento x_n, se verifica si este es mayor o menor al cuantíl estimado $\hat{q}_n$. Sea c_- y c_+ el número de elementos menores y mayores que $\hat{q}_n$, respectivamente. Si x_n es menor al estimado, y $c_- + 1 > np$, entonces se decrementa el valor estimado en λ_n y se incrementa c_+. Por otro lado, si $c_- + 1 \leq np$, se incrementa c_-. En el caso de que el elemento entrante sea mayor al estimado, se verifica si $c_- + 1 > n(1-p)$. De ser el caso, se incrementa el estimado en λ_n y se decrementa c_-. Si no, se incrementa c_+. Si bien este algoritmo no está basado en sketches, creí importante mencionarlo ya que es un trabajo reciente que trata sobre la estimación de cuantiles en hardware. La mayor ventaja de esta solución es su bajo costo en memoria (36 bytes) y su baja utilización de recursos en el switch.

Adicionalmente, también se han propuesto implementaciones en P4 para otros algoritmos basados en sketches con aplicaciones distintas a la estimación de cuantiles. Algunos de estos trabajos son los mencionados a continuación.

En 2022 se publicaron dos trabajos para la estimación de entropía utilizando interpolación tabular en switches programables. El trabajo de Yu-Kuen Lai et al. [28] logra alcanzar una velocidad de procesamiento de 100Gbps tanto en una tarjeta Xilinx U200 como en un switch programable Tofino. Adicionalmente, Yu-Kuen Lai et al. [29] logra implementar un algoritmo basado en sketches en un ASICs programable en P4, transformando computos complejos en tablas pre-computadas en tablas "match-action". Este esquema es luego implementado en un switch Barefoot Tofino2, pudiendo estimar con precisión la entropía de tráfico de red con una velocidad de 400 Gbps.

En 2023, Yang et al.[30] proponen Tower Sketch, un sketch de estimación de frecuencia basado en la estructura del Count Min, pero con contadores de largo variable e implementable en P4. Este sketch permite realizar distintas mediciones (estimación del tamaño de un flujo, detección de heavy-hitters, estimación de entropía, estimación de cardinalidad, entre otros). Además, proponen una arquitectura de procesamiento usando switches que soporten INT (In-Band Network Telemetry), método que consiste en insertar información predefinida adicional a los paquetes. Se obtienen mejoras en la utilización de memoria. Se obtienen resultados bastante similares al CountMin y el CountMin-CU y un tiempo de inserción levemente mayor.

2.2.2. Aceleradores Hardware en FPGA

En 2018, los autores Da Tong y Prassana [7] proponen una arquitectura general para acelerar algoritmos basados en sketches en FPGA. Al probarla en dos sketches ampliamente usados (Count-Min y K-ary) obtuvieron un aumento en el rendimiento de hasta 400x comparado con el estado del arte, alcanzando velocidades mayores a 150Gbps.

En el mismo año, Saavedra et al. [31] diseñan una arquitectura basada en el Countmin-CU para la detección de Heavy Hitters. La arquitectura opera con un reloj de 300MHz, obteniendo una mejora en velocidad de 768 veces comparado con un computador moderno de escritorio.

En una publicación del año 2020, Soto et al. [32] proponen una arquitectura para estimación de entropía utilizando los k elementos más frecuentes. Se utiliza un sketch Countmin-CU el cual alimenta un arreglo de colas de prioridad que busca almacenar los elementos más frecuentes con los cuales será calculada la entropía. Al implementar la arquitectura en una Xilinx Zynq UltraScale+ MPSoC ZCU102, los autores logran obtener un flujo de procesamiento superior a 181 Gbps, con un consumo de potencia de 511mW. En un trabajo siguiente [33], los autores proponen una modificación a la estimación de la entropía, asumiendo que los elementos que no son los más frecuentes siguen una distribución uniforme. La implementación en condiciones similares al trabajo anterior soporta un flujo mínimo de 204 Gbps. Fernández et al. [34] propone asumir que los elementos no pertenecientes a los top-k siguen una distribución de tipo power-law. Manteniendo el mismo flujo de procesamiento, la nueva estimación obtiene errores de estimación promedios 3.5 veces menores que el algoritmo en [33] y más de 15 veces menor que [32].

Otro trabajo propuesto en 2021 por Sateesan et al. [35] propone un sketch llamado Approximate Count Min o ACM, el cual utiliza contadores aproximados. Se modifica el algoritmo de contadores aproximados para hacerlo más amigable con el hardware de un dispositivo FPGA. Además, se usa una distribución de la memoria interna optimizada para una máxima frecuencia de operación. De esta manera, se obtiene una frecuencia de operación de 454.5 MHz para un contador de 16 bits. Para el mismo tamaño de contador, se obtiene una velocidad de procesamiento mínima de 212Gbps.

Adicionalmente, Ebrahim [36] propone un algoritmo optimizado para FPGA que permite encontrar los top-k Heavy Hitters en un stream. Implementado en una tarjeta Stratix 10, se alcanza un flujo de 542 millones de elementos por segundo.

2.2.3. Discusión

De los trabajos antes mencionados se desprende que el diseño de aceleradores hardware que usan algoritmos basados en sketches para análisis de tráfico de red es un área de investigación activa. Ambos tipos de aceleradores tienen sus ventajas y desventajas. Por un lado, al usar ASICs, los switches programables en P4 permiten obtener velocidades de procesamiento más altas que las que se pueden obtener en un dispositivo FPGA. No obstante, la poca flexibilidad de su modelo de programación limita los algoritmos que se pueden implementar en un switch programable. En este sentido, gracias a la lógica programable, los dispositivos FPGA permiten más flexibilidad a la hora de implementar cualquier algoritmo. Adicionalmente, como lo demuestran los trabajos recientes, sigue siendo posible obtener tasas de procesamiento cercanas o superiores a los 200 Gbps con arquitecturas implementadas en FPGA. Dado esto, desarrollamos nuestra arquitectura para un dispositivo FPGA, aprovechando el paralelismo y flexibilidad disponible en esta clase de dispositivos.

Por otro lado, NetFPGA es un framework de código libre que permite crear y simular distintos dispositivos de red en un dispositivo FPGA. Esto permite validar de mejor manera los diseños que utilicen datos de red. La tarjeta que utilizaremos, la Alveo U55C, forma parte de la misma familia que las tarjetas soportadas por NetFPGA y posee la misma arquitectura que los modelos soportados, por lo que más adelante podemos incluir el uso de esta plataforma para validación.

Capítulo 3. Descripción y funcionamiento del sketch KLL

Debido a las características discutidas en la sección 2.1, nos basaremos en la estructura general del sketch KLL para diseñar la arquitectura de un acelerador que nos permita estimar cuantiles en datos de tráfico de red. En este capítulo presentamos el funcionamiento en detalle del sketch. En primer lugar, explicamos el problema específico que resuelve el sketch. Luego, describimos la estructura de datos, mencionando como calculamos los parámetros que determinan su tamaño y error de estimación. Luego, explicaremos los algoritmos que definen las operaciones soportadas por el sketch.

3.1. Problema a resolver

Para entender el problema que busca resolver el sketch KLL, debemos en primer lugar comprender la definición de la operación rank. El rank de x, $R(x)$, está definido como la cantidad de elementos de un conjunto $x_1, ..., x_n$ tal que $x_i \leq x$.

Considerando lo mencionado anteriormente, existen dos problemas relacionados con la estimación de rank, los cuales tienen exigencias distintas:

- El problema de aproximación de un único cuantíl, o *single quantile approximation problem*: Dado un conjunto de elementos $x_1, ..., x_n$ obtenidos por streaming en un orden arbitrario, se debe construir una estructura para calcular $\hat{R}(x)$. Al final del stream, se recibe un único elemento x y se calcula $\hat{R}(x)$ tal que $|\hat{R}(x) - R(x)| \leq \varepsilon n$ con probabilidad $1 - \delta$.
- El problema de aproximación de todos los cuantiles, o *all quantiles approximation problem*: Dado un conjunto de elementos $x_1, ..., x_n$ obtenidos por streaming en un orden arbitrario, se debe construir una estructura para calcular $\hat{R}(x)$. Al final del stream, para todos los valores de x simultáneamente, se debe cumplir que $|\hat{R}(x) - R(x)| \leq \varepsilon n$ con probabilidad $1 - \delta$.

El KLL está centrado en resolver el problema de aproximación de un único cuantíl. Sin embar-

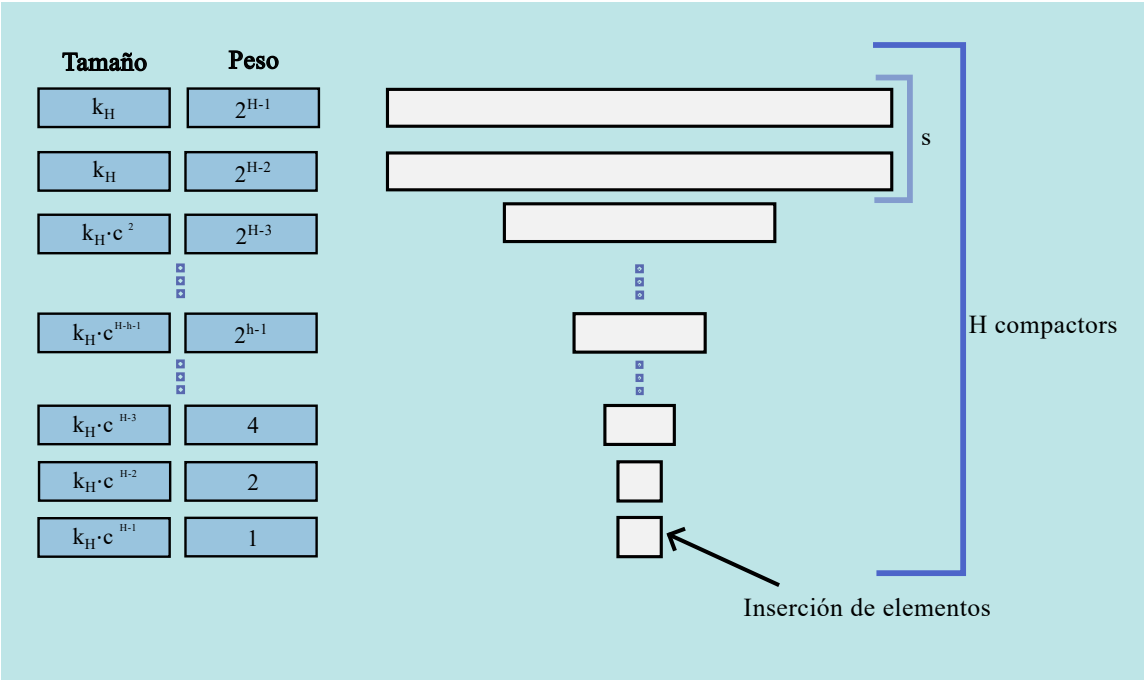

Figura 3.1: Estructura del KLL Sketch

go, aproximar un set de $O(\frac{1}{\varepsilon})$ consultas es suficiente para resolver el problema de aproximación de todos los cuantiles. Es por esto que resolver el primer problema con una probabilidad de fallo de $\varepsilon\delta$ constituye una solución válida para el problema de todos los cuantiles. Este segundo problema es el que queremos resolver, dado que queremos tener aproximaciones de varios cuantiles con una buena precisión, para poder entender mejor la distribución del tráfico de red.

3.2. Estructura del sketch

El sketch KLL está compuesto por estructuras que almacenan elementos llamadas compactors.

La Figura 3.1 muestra la estructura del sketch.

La cantidad máxima de compactors está dada por el parámetro H_{max}, el cual se calcula de la siguiente manera:

$$H_{max} = \log(\varepsilon n),$$

donde n es el número de elementos para el que está dimensionado el sketch.

Cada compactor de altura h, con $0 \leq h < H$, tiene un tamaño k_h , siendo k_H el tamaño máximo. Este tamaño máximo se determina base a los parámetros ε y δ usando la expresión:

$$k_H = O(\frac{1}{\varepsilon} \log \log(\frac{1}{\varepsilon\delta})),$$

donde, como mencionamos anteriormente, δ es la probabilidad de que el error aditivo de la estimación de rank supere εn.

Los $s = O(\log \log(\frac{1}{\varepsilon\delta}))$ compactors superiores mantienen el tamaño máximo. Luego de esto, el tamaño de cada compactor decrece según una constante c, es decir $k_h = k_h c^{H-(h+1)}$. Cabe mencionar que la altura H del sketch crece a medida que se insertan elementos, por lo que el tamaño de un compactor de altura h irá decreciendo, pero manteniendo siempre un tamaño mínimo de 2. En la figura se observa que los dos compactors inferiores tiene el mismo tamaño. Esto es debido a que no pueden tener un tamaño inferior a 2, por lo que el tamaño se conserva.

Los elementos son insertados en el compactor más pequeño (el de altura menor) y al llenarse este, por medio de un proceso llamado compactación, el cual se verá en detalle en la siguiente subsección, elige de manera aleatoria la mitad de sus elementos y los inserta en el compactor de un nivel más arriba. Si no existe un nivel superior, entonces este se crea y la altura H del sketch aumenta en 1. El peso de los elementos almacenados en cada compactor aumenta al doble según la altura.

3.3. Muestreo

Dado que el proceso de compactación selecciona aleatoriamente la mitad de los elementos y los empuja al compactor inmediatamente superior, una secuencia de H'' compactors de tamaño 2 equivale a realizar muestreo con una tasa de muestreo de $2^{H''}$. Es por esto que, si el compactor inferior decrece lo suficiente como para que su tamaño sea 2, el parámetro H'' del sampler aumenta en 1. Cada $2^{H''}$ elementos, se selecciona uno y se inserta en el compactor de peso $2^{H''}$. Considerando lo anterior, es

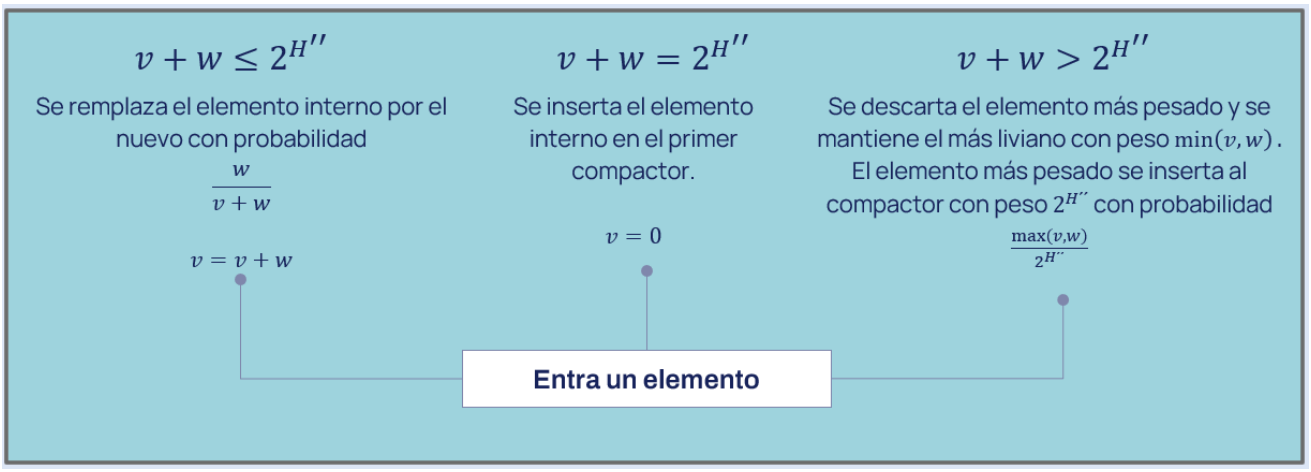

Figura 3.2: Funcionamiento del sampler

posible remplazar los compactors de tamaño 2 por una estructura a la que llamaremos "sampler", la cual selecciona 1 elemento de cada $2^{H''}$ (nos referimos de aquí en adelante al parámetro H'' como la altura de este sampler).

La Figura 3.2 ilustra el procedimiento a seguir para esta selección, donde w es el peso del elemento entrante y v es el peso del elemento interno almacenado en el sampler.

Si la suma del peso del elemento entrante y el peso del elemento interno es menor a $2^{H''}$, entonces el elemento almacenado por el sampler es remplazado con probabilidad $\frac{w}{v+w}$. Sin importar si el elemento fue remplazado o no, se incrementa el peso del elemento interno del sampler en w. Por otro lado, si $v + w = 2^{H''}$, se inserta el elemento en el compactor de altura H''. Cabe mencionar que, en el caso de la inserción de elementos por streaming, w siempre será igual a 1. Debido a esto, la condición en que $v + w > 2^{H''}$ no puede cumplirse. Esta condición se cumplirá solo en caso de que se esté realizando una operación de unión entre estructuras, operación que no discutiremos en este trabajo.

En las siguientes secciones usaremos H''_{max} para referirnos a la altura que alcanzará el sampler después de que el sketch haya recibido n elementos. Este parámetro se calcula como $H''_{max} = H_{max} - \lceil \log_c \frac{2}{k_H} \rceil$

Al usar este sampler, el sketch disminuye su complejidad espacial a $O((\frac{1}{\varepsilon}) \log^2 \log(\frac{1}{\epsilon\delta}))$ para el caso de todos los cuantiles.

3.4. Compactación

El algoritmo de compactación le permite a un compactor seleccionar aleatoriamente la mitad de sus elementos e insertarlos en el compactor de arriba cuyo peso es el doble. Aunque el paper no entrega una definición formal de este algoritmo, los algoritmos contenidos en este informe reflejan nuestra compresión del sketch y son consistentes con la manera en la que implementamos el algoritmo. Para todos los algoritmos presentes en este informe, consideramos que el primer compactor después del sampler tiene altura $h = 0$ y que por ende el tamaño de un compactor de altura h, k_h, es igual a $k_H * c^{H-h-H''-1}$. Como se muestra en el Algoritmo 1, se usa un bit aleatorio para decidir si se insertarán las posiciones pares o impares al compactor de arriba. Luego, se recorren los elementos del compactor y se insertan los seleccionados en el compactor superior. Además se realizan procesos de compactación en el compactor superior de ser necesario. Finalmente, se borran todos los elementos del compactor y para finalizar el proceso de compactación, se elimina el compactor inferior en caso de que este se haya agregado al sampler. En caso de que estemos compactando el compactor de más arriba, debemos hacer crecer el sketch y agregar un compactor. Esta operación se muestra en el Algoritmo 2. Si la altura del sketch es igual a $H_{max} - H''_{max}$, se aumenta la altura del sampler en 1 y se agrega un compactor superior. De lo contrario, se aumenta la altura H en 1. Finalmente, se inserta un nuevo compactor en la parte superior y se disminuye el tamaño de los compactors inferiores multiplicándolos por c en caso de que su altura sea menor a $H - s$.

3.5. Inserción de un elemento al sketch

El Algoritmo 3 presenta la operación de inserción de un elemento x al sketch. En primer lugar, se inserta este elemento al sampler. En caso de que el sampler expulse un elemento, se inserta este elemento en el compactor 0. Si no hay espacio para insertar este elemento, se compacta el compactor 0 y luego se inserta.

Algoritmo 1: Algoritmo de compactación para un compactor de altura h

```
sort(Compactors[h]);
select=random(0,1) ; // Selección del offset
;
if Compactors.length() ≤ h + 1 then
    Compactors.grow();
if Compactors[h + 1].length() = k_h then
    Compactors[h + 1].compact();
i = 0;
while i <Compactors[h].length() do
    Compactors[h + 1].push(Compactors[h][i + select]) ; // Insertamos en el
     compactor superior los elementos con índice par si select = 0
     y aquellos con índice impar si select = 1
    i = i + 2;
if Compactors[h + 1].length() > k_h then
    Compactors[h + 1].compact();
Compactors[h].clear() ; // Vaciamos el compactor de altura h
 if sampler_level_added and h == 0 then
    Compactors[0].compact(); // compacta el compactor inferior,;
    Compactors.pop(0); // y lo elimina.;
return;
```

3.6. Estimación

Para estimar el rank de un elemento x usando el sketch KLL, se debe comparar este elemento x con cada elemento contenido en los compactors. En caso de que el elemento contenido en el compactor sea menor o igual a x, se le suma el peso del elemento a la estimación. Este peso equivale a $w_h = 2^{H''+h}$ Esta operación se presenta en el Algoritmo 4.

Por otro lado, es bueno mencionar que si queremos saber en qué cuantíl se encuentra un elemento x, basta con dividir la estimación de rank por el número de elementos insertados al sketch.

Algoritmo 2: Algoritmo de crecimiento del sketch (Compactors.$grow()$)

if $H = H_{max} - H''_{max}$ **then** // Si ya alcanzamos el número máximo de compactors H_{max}
    $H'' = H'' + 1$; // Añade 1 a la altura del sampler,
    $Compactors.resize(H + 1)$; // Crea el nuevo compactor superior
else
    $H = H + 1$;
    $k.resize(H)$;
    $Compactors.resize(H)$; // Crea el nuevo compactor superior
$i = 0$;
while $i < Compactors.length()$ **do** // Disminuye el tamaño de los compactors inferiores
    if $i \geq H - s$ **then**
        $k[i] = k_H$;
    else
        $k[i] = k_H * c^{H-(i+1)}$;
    $i = i + 1$;
return;

Algoritmo 3: Algoritmo de inserción

input : Un elemento x
Sampler.$insert(x)$;
if new_sample **then**
    if *Compactors[0].*$length() = k_0$ **then**
        Compactors[0].$compact()$;
    Compactors[0].$push(sampled\_element)$;
return;

Algoritmo 4: Algoritmo de estimación de rank

input : Un elemento x
output: $R(x)$, el rank del elemento.
$h = 0$;
$rank = 0$;
while $h < Compactors.length()$ **do**
    $w_h = 2^{H''+h}$;
    foreach $item$ *in* $Compactors[h]$ **do**
        if $item \leq x$ **then**
            $rank = rank + w_h$;
    $h = h + 1$;
return $rank$;

Capítulo 4. Arquitectura del acelerador hardware

Como mencionamos anteriormente, nuestra arquitectura está basada en el algoritmo del sketch KLL. Sin embargo, con el objetivo de eliminar el componente dinámico de esta estructura para hacerla más compatible con las características estáticas del hardware, tomamos la estructura que se genera en el sketch después de insertar un número fijo de elementos. Este algoritmo basado en el KLL fue implementado además en C++, para verificar los errores obtenidos y poder dimensionar los parámetros. La elección de estos parámetros así como también el número de elementos para el que dimensionamos la arquitectura está explicado en el capítulo 5. El detalle de la arquitectura está descrito en las distintas secciones de este capítulo.

4.1. Arquitectura general

La Figura 4.1 muestra la arquitectura propuesta para un KLL con $H - H'' = 7$ y $s = 2$. Podemos observar que los elementos se insertan en el sampler, el cual, a su vez, inserta los elementos en la cola FIFO que se comunica con el compactor C_0. Esta cola se agrega para compensar las diferencias instantáneas entre la velocidad a la que el sampler entrega elementos y la velocidad a la que el compactor C_0 puede recibir elementos. Una vez que se llega al compactor C_0, los elementos subirán a compactors superiores solo en caso de sobrevivir a un proceso de compactación. Los RNG (Random Number Generators) entregan un número aleatorio a cada uno de los compactors por cada compactación y un número aleatorio por cada inserción en el caso del sampler. A su vez, cada compactor se comunica con el módulo de estimación del rank, entregándole la cantidad de elementos contenidos en ellos que son menores o iguales a x.

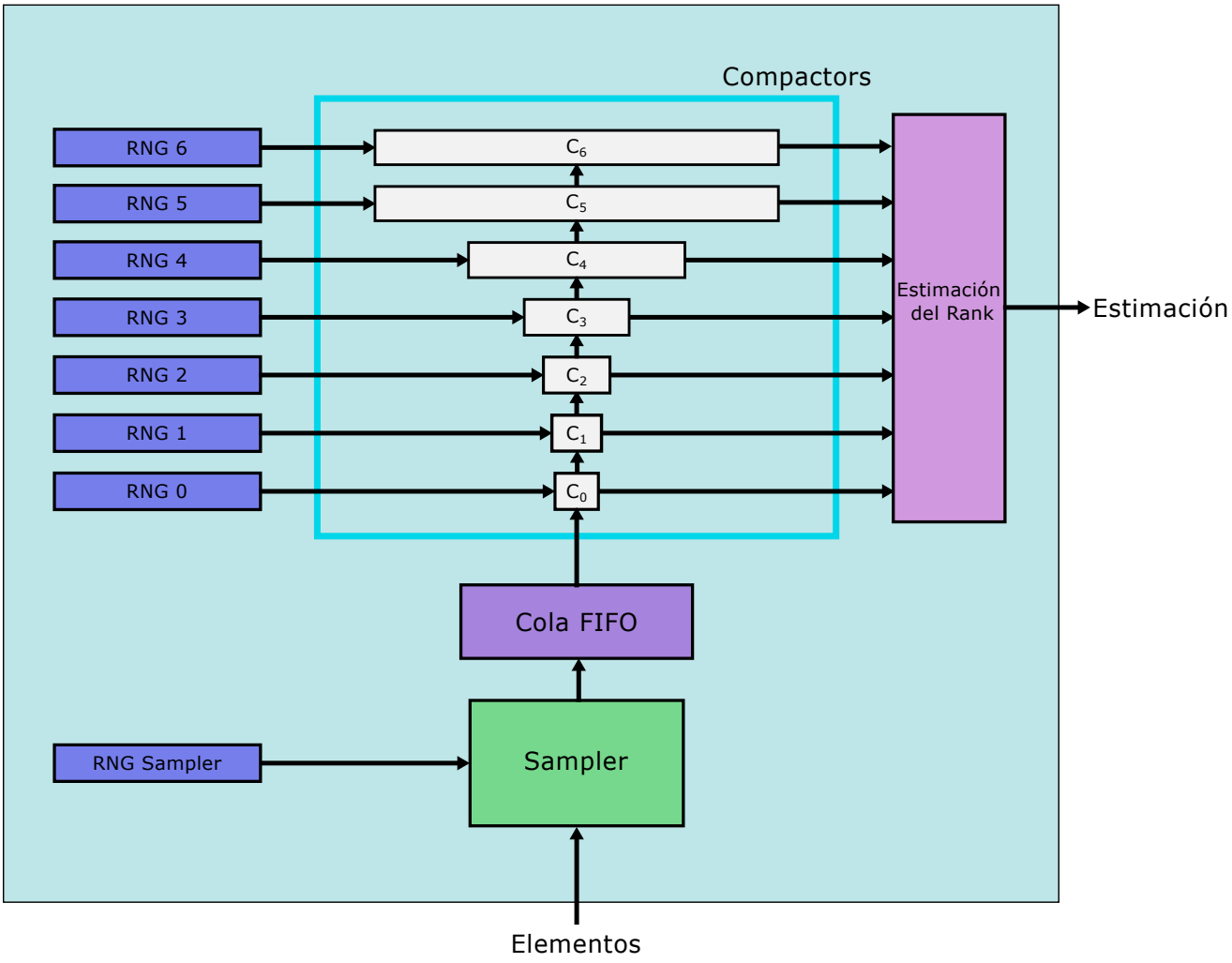

Figura 4.1: Arquitectura general

4.2. Generador de números pseudoaleatorios con LFSR

Teniendo en cuenta la naturaleza aleatoria del KLL, debíamos buscar la manera de poder generar números pseudoaleatorios en un dispositivo FPGA. Debido a lo amigable de este método con el hardware, decidimos utilizar un LFSR, o registro de desplazamiento con retroalimentación lineal. Un LFSR es un registro de desplazamiento cuyo bit de entrada es una función lineal de su estado anterior. Su estado inicial se denomina semilla o “seed”, y debe ser distinta de 0. Un LFSR tiene asociado un polinomio característico. Este polinomio representa los bits que, luego de pasar por una compuerta XOR, determinarán el bit de feedback.

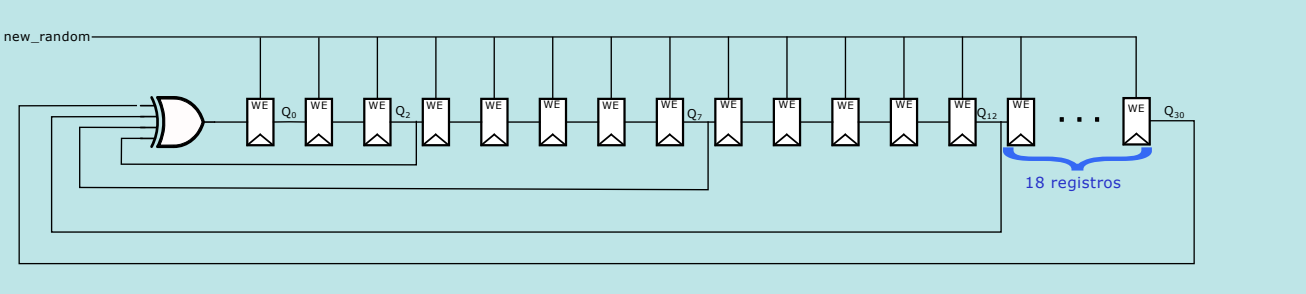

Figura 4.2: LFSR con polinomio característico $1 + x^3 + x^8 + x^{13} + x^{31}$

Estas estructuras nos permiten generar una secuencia pseudoaleatoria con baja correlación, la cual tiene un periodo de $2^N - 1$ elementos cuando el polinomio característico $P(x)$ de grado N es un polinomio primitivo en $GF(2^N)$. Esto quiere decir que el polinomio es irreducible, sus coeficientes son 0 o 1 y el menor entero a tal que $P(x)$ divida a $x^a + 1$ es $a = 2^N - 1$.

Dado que encontrar polinomios primitivos no es una tarea trivial, se utilizan herramientas ya desarrolladas o tablas de polinomios primitivos según el grado, las cuales están disponibles públicamente [37, 38].

Dado que para la implementación del KLL necesitamos generar una gran cantidad de números pseudoaleatorios (se necesita uno por cada inserción y por cada compactación), decidimos utilizar un LFSR de 31 bits. Esto nos permite generar 2.147.483.647 números pseudoaleatorios antes de que el patrón de generación se repita. Para la implementación de este LFSR, seleccionamos el siguiente polinomio primitivo (verificamos con Matlab que efectivamente fuera primitivo en $GF(2^N)$), como polinomio característico:

$$[1 + x^3 + x^8 + x^{13} + x^{31}]$$

Esto se traduce en el circuito lógico de la Figura 4.2 donde la salida de cada registro representa un bit del número aleatorio generado, y la señal “new_random” le indica al módulo que debe generar un nuevo número aleatorio, lo cual produce un desplazamiento en los registros. Para simplificar la figura, se omitieron 16 registros, los cuales están simbolizados por los puntos suspensivos de la figura.

4.3. Sampler

Una parte fundamental del sketch KLL es la posibilidad de realizar muestreo de ser necesario. Es conveniente que el sampler pueda soportar distintas alturas en caso de querer variar la estructura del sketch, o el ancho máximo k_H. Es por esto que hemos diseñado la arquitectura de este módulo para soportar alturas de 0 hasta 6.

Como mencionamos en la sección 3.3, solo realizaremos inserciones y no una operación de unión, es por esto que solo consideramos los dos primeros casos de la Figura 3.2 y asumimos que $w = 1$. Sin embargo, nuestro módulo recibe el logaritmo de w(*wlog*) como una entrada con el objetivo de permitir extender este sampler a un w arbitrario en un futuro.

Recordemos que el elemento interno del sampler se reemplaza con una probabilidad de $\frac{w}{v+w}$. Dado que la división no es una operación soportada nativamente en los FPGA, utilizaremos una LUT, o Look-up Table, la cual almacena el resultado de esta división para los distintos valores de $v \in [0, 63]$ y $w \in [1, 32]$. El índice de esta tabla es una concatenación entre el valor v, el peso interno almacenado en el sampler, y $wlog$, el logaritmo del peso del elemento entrante. Si bien asumimos que $w = 1$, la tabla está diseñada para permitir distintos pesos w, ya que de esta manera no es necesario modificarla en caso de querer extender la funcionalidad del sampler para cualquier valor de w. Es por esto que el índice idx se determina de la siguiente manera según la altura del sampler H'':

$$idx = \{v[H'' - 1 : 0], wlog\},$$

donde $v[H'' - 1 : 0]$ representa los H'' bits menos significativos de v.

Como H'' se mueve entre 0 y 6, el valor máximo de v es $2^6 - 1$, lo cual se puede representar con 6 bits. Por otro lado, $wlog$ se mueve entre 0 y 5, por lo que solo necesitamos 3 bits para representarlo. Es por esto que sabemos que la cantidad máxima de bits que ocupa este índice es de 9 bits. De esta manera, sabemos que la tabla debe contener 512 elementos.

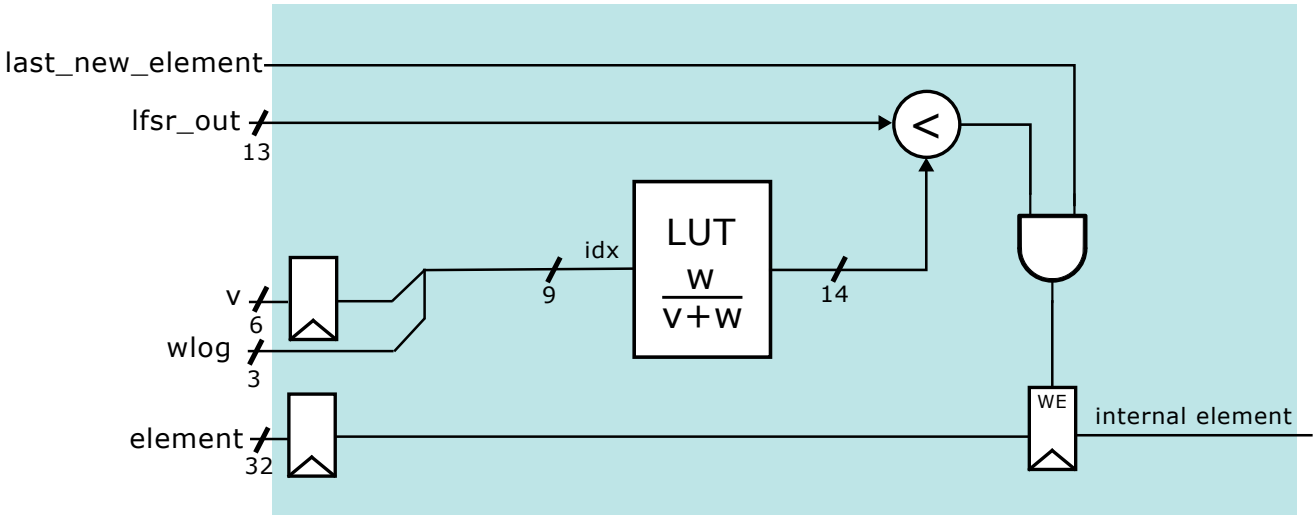

Figura 4.3: Módulo de decisión para el remplazo del elemento interno

Ahora, debemos determinar cuantos bits usaremos para representar el resultado de la división. Para esto, calculamos todos los resultados posibles de esta. Luego, calculamos la diferencia más pequeña que existe entre dos valores consecutivos de estos resultados. La diferencia más pequeña es $0{,}00025$. Si dividimos 1 por este valor, obtenemos la cantidad de elementos distintos entre 0 y 1 que debemos poder representar para distinguir los valores de la probabilidad entre ellos. El resultado de esta división nos informa que necesitamos poder representar por lo menos 4032 valores, para lo que se necesitan al menos 12 bits. Sin embargo, para representar la probabilidad decidimos utilizar una representación de 14 bits, ya que de esta manera podemos realizar una comparación más precisa entre el número aleatorio generado y el resultado de la división. Representaremos entonces el resultado de la división como un entero de 14 bits. Dado que el resultado de esta división está entre 0.015625 y 1, los valores almacenados en la LUT se moverán entre 128 y 8192.

Como podemos observar en la Figura 4.3, cuando llegó un nuevo elemento el ciclo anterior (señal “last_new_element”), el valor que se obtiene de la LUT se compara con la salida de 13 bits del generador de números aleatorios $lfsr\_out$, que varía entre 0 y 8191. En caso de que el número aleatorio generado sea menor que el valor de la LUT, se reemplaza el objeto almacenado en el sampler.

Además, debemos implementar la salida de los elementos seleccionados. El sampler entrega un elemento 3 ciclos de reloj después de la llegada de los $2^{H''}$ elementos que representará ese elemento. Los elementos se procesan con un ciclo de retraso, por lo que a la llegada de un elemento, el elemento

interno y el peso son actualizados 2 ciclos después.

El Extracto de código 1 en SystemVerilog muestra la lógica de actualización del peso interno del sampler y la emisión de un elemento hacia los compactors. Si llegó un nuevo elemento el ciclo anterior y $v = 2^{H''}$, entonces se emite el elemento interno del sampler activando la señal "new_out_element". Si no llegó un nuevo elemento en el ciclo anterior, se verifica si el peso es igual a $2^{H''}$ y, de ser así, se emite el elemento interno. Esta última condición se necesita en caso de que el número de elementos insertados en el sketch sea múltiplo de $2^{H''}$.

Por otro lado, podemos ver que en caso de haya llegado un elemento en el ciclo anterior y no se cumpla la condición para expulsar un elemento del sampler, entonces el peso interno del sampler es incrementado. En caso de que podamos expulsar un elemento del sampler, el peso se reinicia a 1 si llegó un nuevo elemento en el ciclo anterior, ya que este elemento está siendo procesado en el ciclo actual. En caso contrario, el peso se resetea a 0, ya que no hay ningún elemento siendo procesado.

4.4. Compactors

Los compactors tienen 3 modos de funcionamiento: el modo de inserción, donde se insertan y se escriben los elementos en la memoria, el modo de compactación donde se leen los elementos seleccionados y se envían al compactor de arriba y el modo de estimación de rank. Los dos primeros modos se describen en esta sección y el último en la sección 4.6.

Cada compactor, excepto el primero y el último, está conectado a un compactor arriba y un compactor abajo. El primer compactor recibe elementos de la FIFO de entrada cada vez que esté preparado para recibirlos. La Figura 4.4 muestra las interfaces de comunicación entre la FIFO y los 3 compactors inferiores.

Cuando el compactor 0 puede recibir un elemento en el ciclo siguiente, levanta la señal "get", la cual le dice a la FIFO que, a menos que esté vacía, debe entregar un elemento en el ciclo de reloj siguiente. Si hay un elemento disponible a la salida de la FIFO, se levanta la señal "new_el_ fifo" y se

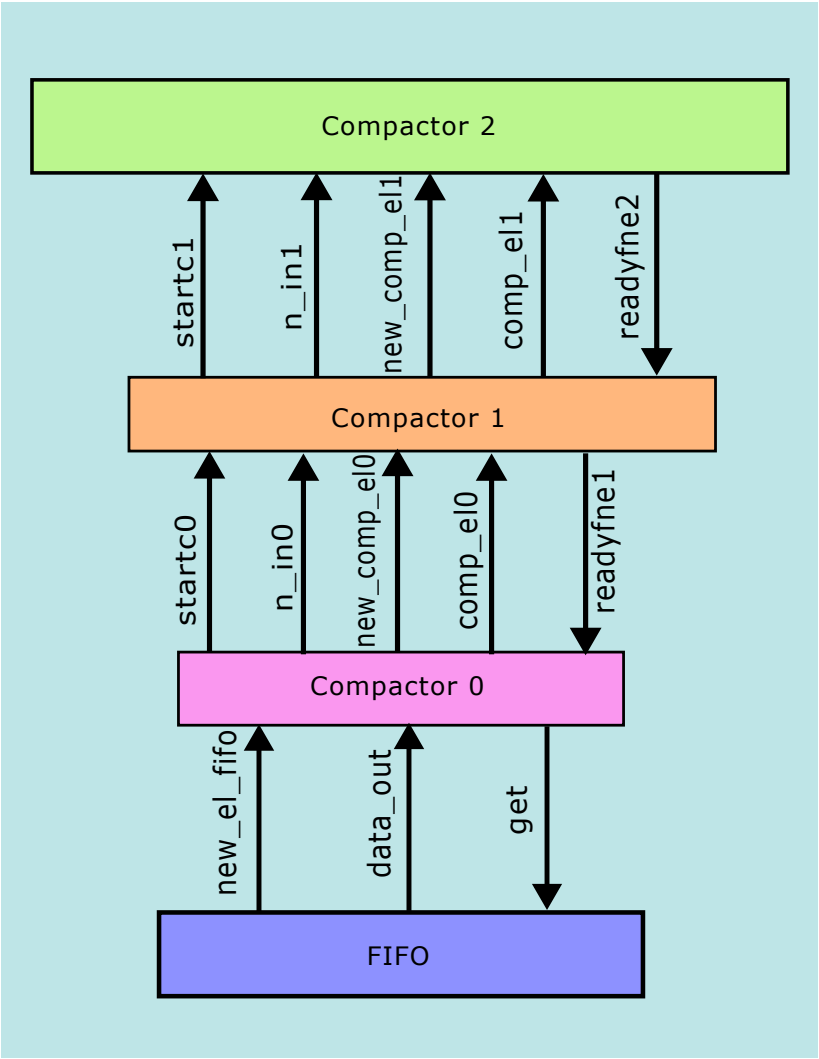

Figura 4.4: Interfaces de comunicación entre módulos compactors y cola FIFO

guarda el elemento en “data_out”. El compactor 0 toma este elemento y lo inserta en su memoria.

Las señales “startc0” y “startc1” son señales que le indican al compactor de arriba que el compactor inferior comenzó su proceso de compactación y se setean en 1 durante un ciclo de reloj cuando el compactor está lleno y llega un nuevo elemento.

En el proceso de compactación, un compactor escribe en las señales “n_in” cuántos elementos va a insertar en el compactor inmediatamente superior. Luego, los envía uno a uno levantando la señal “new_comp_el” y asignando a “comp_el” al valor del elemento que se desea enviar. Luego, espera a que el compactor superior levante la señal “readyfne” para enviar el siguiente elemento, hasta que se

hayan enviado todos.

4.4.1. Inserción

El proceso de inserción consiste en insertar los elementos uno a uno de manera que todos los elementos del compactor queden ordenados de menor a mayor. Decidimos insertar los elementos de manera ordenada usando un algoritmo de ordenamiento in-place, ya que esto nos permite simplificar el proceso de compactación y no usar memoria adicional para ordenar. El Algoritmo 5 describe el método que usamos para la inserción, asumiendo que el compactor tiene suficiente espacio para insertar todos los elementos. De lo contrario, se realiza un proceso de compactación, para luego insertar los elementos faltantes.

Diseñamos la memoria en la que se escriben los elementos para que pueda ser implementada como una Block RAM. Esto implica que el resultado de la lectura estará disponible en el ciclo de reloj siguiente. Es por esto que este proceso fue implementado como una máquina de dos estados:

- El estado de llegada. En este estado se verifica si el compactor está vacío y de ser el caso, se inserta directamente el elemento en la primera posición del compactor y la máquina se mantiene en este estado. En caso contrario, se determina el puntero de lectura y escritura en base a los elementos distintos de 0 contenidos en el compactor y los elementos a insertar.
- El estado de comparación. En este estado se compara lo leído en el ciclo anterior con el elemento entrante, para determinar qué será escrito en la posición apuntada por el puntero de escritura. Se permanece en este estado hasta que el elemento entrante haya sido escrito.

La Figura 4.5 muestra las entradas y salidas de la BRAM. En primer lugar, están las señales de habilitación de escritura y lectura “we” y “re”. Luego, están los punteros de escritura y lectura “wr_ptr” y “rd_ptr”. Finalmente, la señal “write” contiene el elemento que se quiere escribir en la memoria y el registro “read” contendrá el contenido de la posición en memoria apuntada por el puntero de lectura del ciclo de reloj anterior.

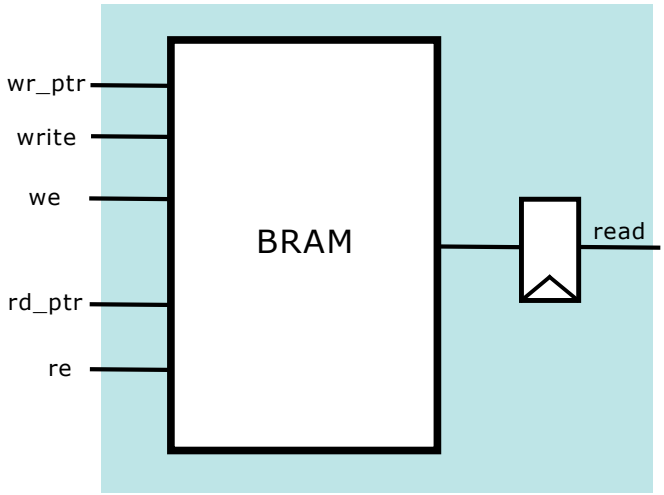

Figura 4.5: Entradas y salidas de la Block RAM

La mayor parte del problema se traduce entonces a determinar los punteros de lectura y escritura, y a determinar qué se escribirá en la memoria.

En caso de estar en el primer estado, solo se escribirá algo en caso de que el compactor esté vacío. De ser así, se escribe en memoria el elemento entrante. La Figura 4.6 muestra el circuito que decide qué elemento se escribirá en la memoria si nos encontramos en el estado de comparación y la señal de habilitación de escritura está activada.

De la figura podemos extraer que si el elemento a insertar es mayor que el contenido leído de la memoria o el puntero de escritura apunta a la última posición en la que se puede escribir el elemento (la cual es igual a la cantidad de elementos que faltan por insertar, “elements_left ”), entonces se selecciona el elemento entrante para escribir en memoria. De no ser así, se selecciona el elemento leído.

En cuanto a las señales de habilitación, la señal de habilitación de escritura está activa en el estado de llegada sólo cuando el compactor esté vacío. En el estado de comparación, esta se encuentra siempre activa. Por otro lado, la señal de habilitación de lectura está activa cuando no se ha escrito el elemento entrante. Esto se cumple en las siguientes condiciones:

- Si estamos en el estado de llegada y el compactor no está vacío.

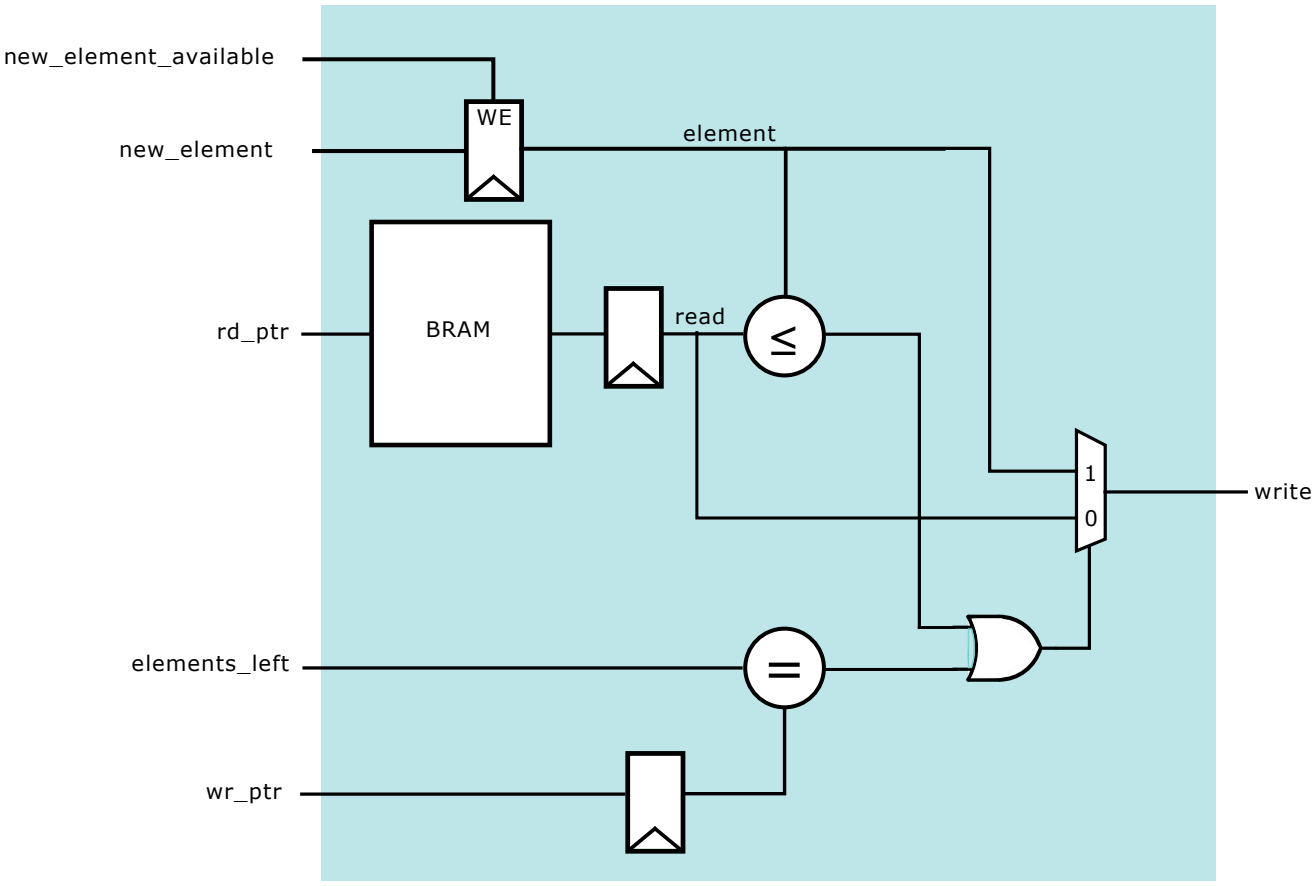

Figura 4.6: Circuito de selección del elemento a escribir

- Si nos encontramos en el estado de comparación y el elemento a insertar es menor o igual que el contenido leído de la memoria.

Finalmente, el Extracto de código 2 muestra como se determinan los punteros de lectura y escritura si nos encontramos en el proceso de inserción. Podemos ver que los valores de estos punteros dependen de 4 variables adicionales al contenido leído y el elemento entrante. En el caso de que el compactor esté vacío, el puntero de escritura se calcula en base a la cantidad de elementos que faltan por insertar “elements_left”. En caso de que nos encontremos en el estado de llegada y el compactor no esté vacío, el valor de los punteros depende de “nze”, la cantidad de elementos contenidos en el compactor, y “eti”, la cantidad de elementos que podemos insertar en el compactor antes de que se llene. Finalmente, en el estado de comparación, el valor de los punteros también depende de “elements_left”. Estos casos sólo son considerados si el compactor no está lleno, por lo que los punteros de escritura y lectura también dependen de la señal “full”.

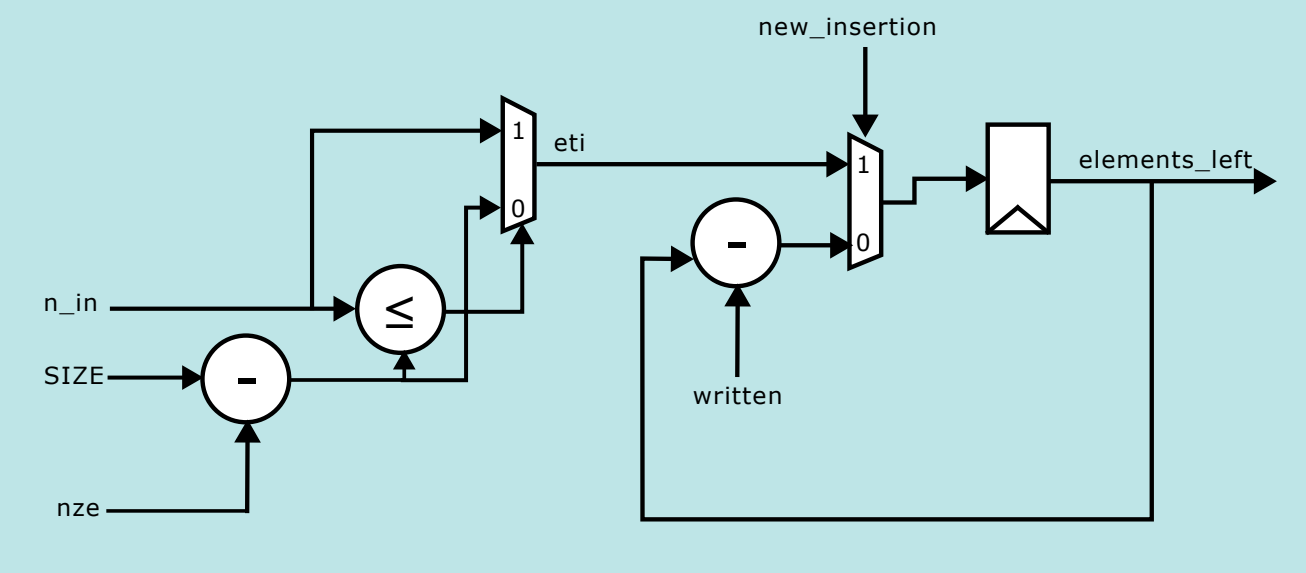

Figura 4.7: Cálculo de elementos a insertar y elementos restantes

La cantidad de elementos contenidos en el compactor "nze" comienza en 0 en el reset y se incrementa cada vez que escribimos un elemento entrante (lo cual es representado por la señal "written").

Por otro lado, como se ilustra en la Figura 4.7, la cantidad de elementos restantes se reinicia a "eti" cuando hay una nueva inserción. Asumiendo que hay espacio para insertar todos los elementos en el compactor, el valor de "eti" es igual a n_{in}. De no haber espacio para insertar todos los elementos, se asigna a "eti" la diferencia entre el tamaño del compactor y el número de elementos contenidos actualmente.

Para finalizar, debemos saber cuándo el compactor está listo para recibir un nuevo elemento. La señal que simboliza esto es la señal "readyfne". El Extracto de código 3 muestra las condiciones en las que se modifica esta señal:

Como describe el código, si llega un elemento a insertar, el compactor está listo para recibir este elemento y este no se escribe directamente, entonces el compactor debe pasar al estado de comparación y no podrá recibir elementos hasta que se escriba el elemento entrante. Una vez que ocurre esto último o termina el proceso de compactación, esta señal vuelve a estar en alto.

4.4.2. Compactación

Para comenzar con el proceso de compactación, debemos primero determinar cuantos elementos insertaremos en el compactor de arriba. Para esto, usaremos el primer bit de la salida del LFSR. Este bit lo almacenamos en un registro llamado “select2”. A partir de este bit también se determina la posición de inicio del contador que usamos como puntero de lectura: “count”, como se muestra en el Extracto de código 4.

Vemos que si el tamaño del compactor es par, entonces se insertan la mitad de los elementos al compactor superior. En cambio, si el tamaño es impar, se usa “select2” como selector. Si “select2” es 1 se insertan $\lfloor \frac{SIZE}{2} \rfloor$ elementos. En caso contrario, se insertan $\lceil \frac{SIZE}{2} \rceil$ elementos al compactor inmediatamente superior.

Al iniciar la compactación, el contador “count” se setea según el primer elemento que queremos enviar. Dado que enviaremos los elementos de mayor a menor para facilitar el proceso de ordenamiento, se selecciona el último índice que debe ser leído. Luego, si el elemento fue enviado con éxito al compactor superior (simbolizado por la señal “sent”) el contador se decrementa en 2. Un elemento se considera enviado con éxito si cuando hay un nuevo elemento compactado disponible, el compactor de arriba está listo para recibir elemento, es decir, si readyfne2 vale 1.

La Figura 4.8 muestra las señales de sincronización para un compactor de 6 elementos con un select2=1. La señal “start_compaction” es una señal que se levanta durante un ciclo de reloj cuando empieza la compactación. Su valor es 1 cuando llega un nuevo elemento, el compactor está lleno y no se está compactando ya. La señal “finish_compaction” marca el final de la compactación y se levanta cuando por primera vez el número de elementos enviados es igual al número de elementos a enviar mientras que la señal “compacting” se mantiene activa mientras el proceso de compactación no haya finalizado. La señal “new_compacted_element” le indica al compactor de arriba que hay un elemento para insertar en “compacted_element”. Esta se activa cuando el compactor de arriba está listo para recibir datos, es decir, si la señal “readyfne2” está activa. Podemos observar que el valor de “sent_elements” se incrementa cada vez que la señal “sent” vale 1.

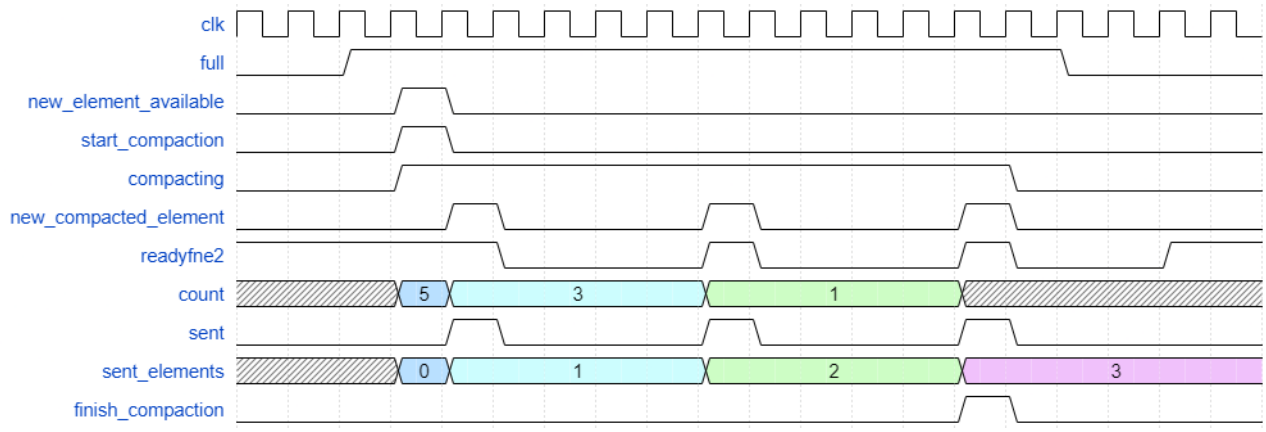

Figura 4.8: Diagrama de tiempo para señales de sincronización de la compactación

Al final de la compactación, se escribe el elemento que no se pudo escribir anteriormente por falta de espacio. Luego, si todavía quedan elementos por insertar, se actualiza el valor de "elements_left" y "eti" para insertar estos.

4.5. FIFO de entrada

La cola FIFO recibe elementos del sampler y los inserta en el compactor 0. El objetivo de usar esta cola es desacoplar el timing de la salida de elementos del sampler (es decir, cada cuantos ciclos de reloj entrega un elemento) con el timing de la inserción de elementos al compactor 0 (cuando este compactor puede recibir elementos) para evitar pérdida de elementos debido a las diferencias instantáneas entre la velocidad de inserción del sampler y la velocidad a la que el compactor 0 puede recibir elementos. Las interfaces de comunicación entre estos tres módulos se presentan en la Figura 4.9.

El compactor pide un elemento con la señal "get" cuando sabe que estará listo para recibirlo en el ciclo siguiente. Esto ocurre cuando se acaba de escribir el elemento entrante en la memoria, cuando termina la compactación y cuando ya estaba listo para recibir elementos de antes y no ha llegado un nuevo elemento. Además. sabemos que habrá un dato válido en "data_out" en el ciclo siguiente y se levantará la señal "new_el_fifo" cuando el compactor pidió un elemento y la cola no estaba vacía o venía llegando un elemento.

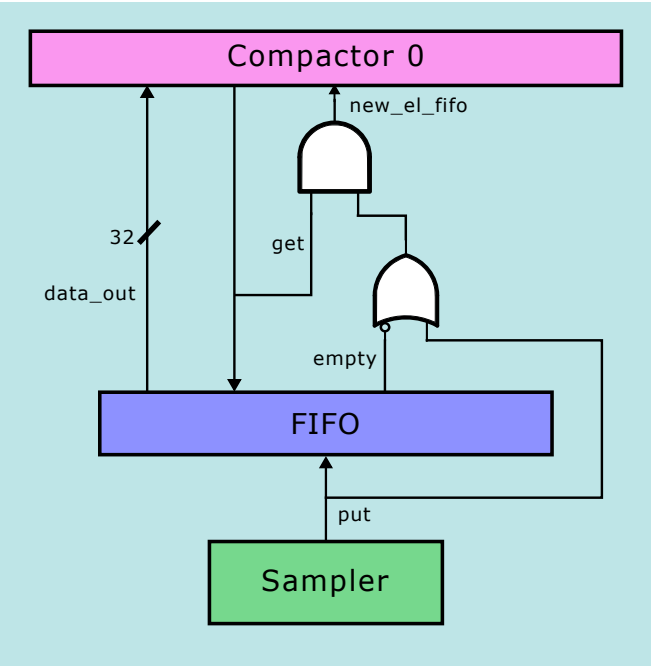

Figura 4.9: Interfaces de comunicación con la cola FIFO

Cabe mencionar que si la FIFO está llena cuando llega un nuevo elemento, y la señal “get” no está activa, entonces el elemento se perderá.

4.6. Estimación

El proceso de estimación de rank consiste en recorrer todos los compactors y sumar los pesos de los elementos menores o iguales al elemento para el cual queremos estimar el rank (En la implementación llamamos a este elemento “query”). En esta arquitectura, el proceso comienza dentro de cada compactor, donde se recorre la memoria y por cada elemento menor o igual a “query”, se incrementa un contador. Para esto, tendremos un contador llamado “count2” que reinicia su valor a 0 cada vez que hay una nueva consulta (señal “new_query”). Este contador se va incrementando a cada ciclo de reloj y se usa como la dirección de lectura de la BRAM. Cuando este contador alcanza el valor de “nze”, es decir, el número de elementos contenidos en el compactor, significa que ya leyó todos los elementos válidos y se levanta la señal “endofread”, la que dará fin al proceso de estimación interno.

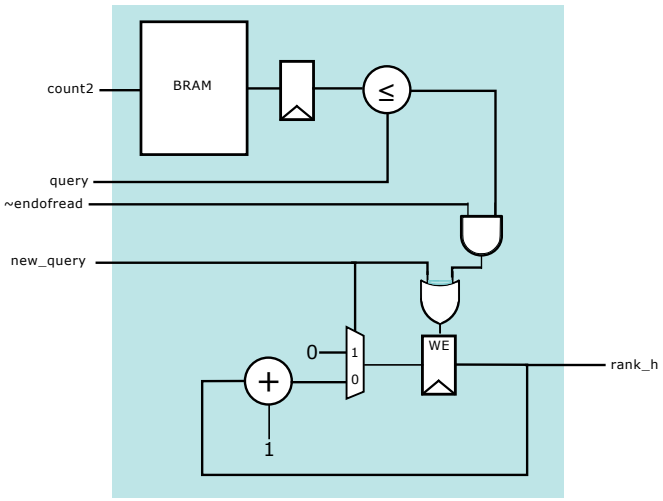

Figura 4.10: Proceso de estimación del rank interno

Este proceso de estimación interno se puede apreciar en la Figura 4.10. El contenido de la BRAM en la posición “count2” se compara con “‘query”. Si el contenido leído es menor o igual a “query” y no se han terminado de leer los elementos, el contador “rank_h” se incrementa.

Una vez que se terminó el proceso de estimación interno, este “rank_h” entregado se suma con la estimación entregada por el resto de los compactors, multiplicando cada una de estas por su peso correspondiente. Este proceso de estimación final se puede observar en el Extracto de código 5, donde “endofread[h]” simboliza que el compactor h terminó la estimación interna, y “processed[h]” significa que la estimación del compactor h ya fue sumada al rank.

Podemos ver que por cada compactor se le suma a “rank” la estimación interna del compactor multiplicada por el peso de este. Este peso equivale a $2^{H''+h}$. Además, como se desprende del extracto de código, si la señal “endofquery” está activa, significa que la estimación terminó y el valor almacenado en “rank” es la estimación final del rank del elemento consultado.

```
always_ff@(posedge clk) begin

        last_new_element <= new_element;

        if(reset) begin
                v <= 0;
                new_out_element <= 0;
        end
        else begin
                if(last_new_element == 1) begin //Si llegó un elemento
                // en el ciclo anterior...
                        if(v < (1 << height)) begin //y el peso es menor a 2^H''
                                new_out_element <= 0;
                                v <= v+1; // se incrementa el peso interno.
                        end
                        else if (v == (1 << height)) begin//...y el peso es
                        //igual a 2^H''...
                                new_out_element <= 1;
                                out_element <= internal_element;
                                //...se inserta el elemento interno en la FIFO...
                                v <= 1; //... y el peso se resetea a 1.
                        end
                end else begin //Si no llegó un elemento el ciclo pasado...
                    if(v == (1 << height)) begin //y el peso es igual a 2^H''
                            new_out_element<=1;
                            out_element<=internal_element;
                            //...se inserta el elemento interno en la FIFO...
                            v<=0; //...y se resetea el peso interno a 0.
                    end else begin
                            new_out_element<=0;
                    end
                end

        end
end
```

Extracto de código 1: Código para la expulsión de elementos del sampler y actualización del peso interno

Algoritmo 5: Algoritmo de ordenamiento in-place

input : Un arreglo x con n_{in} elementos a insertar ordenados de menor a mayor

```
1  wr_ptr = Compactors[h].length() + n_in − 1; // Partimos escribiendo en la
     última posición del compactor...
2  rd_ptr = Compactors[h].length() − 1; // ...y leyendo en la última
     posición llena del compactor.
3  Compactors[h].resize(Compactors[h].length() + n_in); // Se agranda el
     compactor para recibir los nuevos elementos
4  n_left = n_in;
5  while n_left > 0 do // Mientras queden elementos por insertar...
       // ...se compara el elemento entrante con el elemento leído
       del compactor.
6      if Compactors[h][rd_ptr] ≥ x[n_left] then // Si el elemento leído es
         mayor o igual...
7          Compactors[h][wr_ptr] = Compactors[h][rd_ptr]; // ... se escribe el
             elemento leído en la posición de escritura...
8          Compactors[h][rd_ptr] = 0; // y el elemento en la posición de
             lectura se reemplaza por 0.
9          wr_ptr = wr_ptr − 1;
10         rd_ptr = rd_ptr − 1;// Se decrementan los punteros de lectura
             y escritura.
11     else // Si no...
12         Compactors[h][wr_ptr] = x[n_left]; // ... se escribe el elemento
             entrante.
13         n_left = n_left − 1;
14         wr_ptr = rd_ptr + n_left;
15 return;
```

```
always_comb begin
            unique case(state)
            arrival_state: begin //Estado de llegada
                if( new_element_available == 1 && ~full) begin
                //Si se puede insertar el elemento...
                    if(nze == 0) begin //...y el compactor está vacío...
                        wr_ptr = elements_left-1; //...se inserta directamente
                        rd_ptr = 0; //... y no es necesario leer.
                    end
                    else begin
                        if(elements_left != eti) begin //Si no es el primer
                        //elemento insertado se mantienen los punteros
                            wr_ptr = last_wr_ptr;
                            rd_ptr = last_rd_ptr;
                        end else begin //Sino se resetean...
                            wr_ptr = nze + eti; //a la última posición a llenar
                            rd_ptr = nze - 1; //y a la última posición con un
                           //elemento insertado anteriormente.
                        end
                    end
                end
            end
            compare_state: begin //Estado de comparación
                    if(last_wr_ptr == (elements_left))begin //Si quedan igual
                   //número de posiciones vacías que elementos por insertar.
                        wr_ptr = last_wr_ptr - 1;
                        rd_ptr = 0;
                    end
                    else if(read <= element) begin //Si el elemento leído es
                    //menor o igual que el elemento entrante...
                        rd_ptr = last_rd_ptr;
                        wr_ptr = last_rd_ptr + elements_left;
                    end else begin
                        rd_ptr = last_rd_ptr - 1;
                        wr_ptr = last_wr_ptr - 1;
                    end
            end
end

always_ff @(posedge clk) begin

            last_rd_ptr <= rd_ptr;
            last_wr_ptr <= wr_ptr;

end

```

Extracto de código 2: Código en SystemVerilog para el cálculo de punteros de escritura y lectura

```
        always_ff @ (posedge clk) begin
                if(new_element_available && readyfne && ~written) begin
                        readyfne <= 0;
                end

                if(written || finish_compaction) begin
                        readyfne <= 1;
                end
        end

```

Extracto de código 3: Código en SystemVerilog para la señal readyfne

```
        always_comb begin

            half = SIZE >> 1;
            if(SIZE[0] == 0) begin
                n_elements_out = half;
            end else begin
                n_elements_out = select2 ? half : half+1;
            end

            if(start_compaction) begin
                if(SIZE[0] == 0) begin
                   count = select2 ? SIZE-1 : SIZE-2;
                end else begin
                   count = select2 ? SIZE-2 : SIZE-1;
                end
                sent_elements = 0;
            end else if(sent) begin
                count = last_count - 2;
                sent_elements = last_sent_elements + sent;
            end else begin
                    count = last_count;
                sent_elements = last_sent_elements + sent;
            end

        end

```

Extracto de código 4: Código en SystemVerilog el número de elementos de salida y el contador de lectura.

```
always_ff@(posedge clk) begin
    if(new_query || reset) begin
        processed<=0;
        rank<=0;
        endofquery<=0;
    end
    else if(endofread[0] && ~processed[0]) begin
        processed[0]<=1;
        rank<=rank+(rank0<<height);
    end
    else if(endofread[1] && ~processed[1]) begin
        processed[1]<=1;
        rank<=rank+(rank1<<(height+1));
    end
    else if(endofread[2] && ~processed[2]) begin
        processed[2]<=1;
        rank<=rank+(rank2<<(height+2));
    end
    else if(endofread[3] && ~processed[3]) begin
        processed[3]<=1;
        rank<=rank+(rank3<<(height+3));
    end
    else if(endofread[4] && ~processed[4]) begin
        processed[4]<=1;
        rank<=rank+(rank4<<(height+4));
    end
    else if(endofread[5] && ~processed[5]) begin
        processed[5]<=1;
        rank<=rank+(rank5<<(height+5));
    end
    else if(endofread[6] && ~processed[6]) begin
        processed[6]<=1;
        rank<=rank+(rank6<<(height+6));
    end
                        .
                        .
                        .
    else if(endofread[14] && ~processed[14]) begin
        processed[14]<=1;
        rank<=rank+(rank14<<(height+14));
    end
    else if(endofread[15] && ~processed[15]) begin
        processed[15]<=1;
        rank<=rank+(rank15<<(height+15));
    end
    else if(endofread[16] && ~processed[16]) begin
        processed[16]<=1;
        rank<=rank+(rank16<<(height+16));
    end
    else if(processed==17'hfffff) begin
        endofquery<=1;
    end
end

```

Extracto de código 5: Código en SystemVerilog para la estimación del rank de un elemento.

Capítulo 5. Selección de parámetros

Como mencionamos en la sección 3.2, la estructura y tamaño del sketch están determinados por los parámetros ε, δ, c y el número de elementos n. Debido a esto, es necesario fijar los valores de estos parámetros a la hora de implementar la arquitectura descrita anteriormente en un dispositivo FPGA. Adicionalmente, debemos seleccionar cuál es el tamaño más adecuado para la memoria FIFO de entrada, descrita en la sección 5.2. Durante este capítulo, seleccionamos los valores de estos parámetros en base a distintos criterios.

En primer lugar, escogemos el número de elementos para el cual dimensionaremos el sketch en base una tasa de 100 Gbps.

Luego, para la selección de ε y δ, tenemos en cuenta 2 criterios:

- El espacio utilizado por el sketch en función de los parámetros escogidos, expresando este espacio en kilobytes.
- La altura del sampler obtenida. Esto debido a que si la altura del sampler disminuye, también disminuye el número mínimo de ciclos de reloj entre cada elemento insertado a la FIFO. Esto puede causar que la FIFO se llene más rápido debido a que ahora los compactors tienen menos tiempo para realizar su compactación antes de que llegue un nuevo elemento. Esto puede ocasionar una mayor pérdida de elementos, disminuyendo así la precisión.

Para la selección del parámetro c, tomamos en cuenta además el resultado de 11 consultas de rank sobre el largo de los paquetes para 4 trazas de red y las usamos para evaluar el comportamiento del error. Para esto, procesamos con anterioridad las trazas de red y extragimos el largo de los paquetes a un archivo, el cúal será nuestro archivo de entrada.

Por otro lado, habiendo ya escogido los valores de los parámetros anteriores, debemos determinar el tamaño de la FIFO presente entre el sampler y el primer compactor. Para esto, repetimos las 11

consultas anteriores para distintos largos de la cola. Realizamos estas consultas sobre la implementación en hardware del algoritmo.

5.1. Selección de parámetros del sketch

5.1.1. Estimación del número de elementos n

Considerando una tasa de 100 Gbps, el peor caso se obtiene cuando todos los paquetes son de largo mínimo, es decir, de 64 bytes. Esto nos entrega un máximo de aproximadamente 200 millones de elementos por segundo. Considerando una ventana de observación de 10 segundos, tendríamos 2 mil millones de paquetes. No obstante, es importante observar que este caso no es realista. En la práctica, los paquetes son de un largo mayor. Es por esto que, para dimensionar el sketch, elegimos a arbitrariamente considerar $\frac{1}{10}$ de este número, es decir n =200.000.000. Este número de elementos coincide con el orden de magnitud del número de paquetes de las trazas más grandes con las que evaluamos nuestro diseño.

5.1.2. Selección de ϵ y δ

Para escoger las cotas de error, evaluamos el espacio en kilobytes utilizado por el sketch para distintas combinaciones de ε y δ , como también el ancho del compactor más grande k_H, la altura del sketch H y la altura del sampler H''. Para calcular estos datos tomamos en cuenta el n de 200 millones seleccionado en la subsección anterior y elegimos un valor de $c = 0{,}66666$, debido a que es el valor de c usado por el paper original [1]. El valor de k_H fue calculado multiplicando por 2 la expresión $\frac{1}{\varepsilon} loglog(\frac{1}{\varepsilon\delta})$ con el objetivo de aumentar la precisión de las estimaciones. Los valores H_{max} y s fueron calculados como se explicó en la sección 3.2. Dado que estas expresiones pueden dar resultados no enteros, estos resultados fueron aproximados al entero superior más cercano. Los resultados de estos experimentos se muestran en la Tabla 5.1.

Tabla 5.1: Espacio utilizado en función de ε y δ

ε	δ	k_H	**H**	**H"**	**Espacio Total en KB**
0,050	0,050	125	24	13	2,30
0,010	0,050	691	21	6	12,70
0,005	0,050	1433	20	3	26,34
0,002	0,050	3733	19	0	68,59
0,001	0,050	7674	18	0	140,95
0,050	0,010	139	24	13	2,56
0,010	0,010	747	21	6	13,73
0,005	0,010	1535	20	3	28,21
0,002	0,010	3965	19	0	72,86
0,001	0,010	8108	18	0	174,94
0,050	0,005	144	24	13	2,64
0,010	0,005	768	21	6	14,11
0,005	0,005	1574	20	3	28,94
0,002	0,005	4054	19	0	87,50
0,001	0,005	8277	18	0	178,58
0,050	0,001	154	24	13	2,84
0,010	0,001	811	21	6	17,51
0,005	0,001	1656	20	3	35,75
0,002	0,001	4243	19	0	91,58
0,001	0,001	8634	18	0	186,28

Al observar la Tabla 5.1, podemos ver que para los valores de ε $0{,}002$ y $0{,}001$, el espacio utilizado aumenta entre $2{,}5$ y 3 veces comparado con el espacio utilizado para $\varepsilon = 0{,}005$. Por otro lado, la altura del sampler se vuelve 0, por lo que estaríamos recibiendo un dato en cada ciclo de reloj. Esto perjudicaría la precisión, ya que estaríamos escribiendo en la FIFO a una tasa mucho más alta de la que podríamos leer, por lo que la perdida de datos sería grande. Es por esto que consideramos adecuado seleccionar un valor de ε de $0{,}005$.

Debemos ahora seleccionar el valor de δ. Seleccionaremos un valor de $\delta = 0{,}01$, lo que equivale a tener una probabilidad de fallo de un 1 %. De esta manera, obtenemos una confiabilidad del 99 % sin hacer crecer demasiado el espacio.

5.1.3. Selección del parámetro c

Una vez fijados los parámetros ε y δ debemos usar estos valores para determinar cuál será el valor de c que utilizaremos. Para esto realizamos 11 consultas de rank sobre el largo de los paquetes para 4 trazas de red con tamaños distintos, las cuales están disponibles los repositorios públicos CAIDA y MAWILab [12, 13, 15]: Chicago 20080319 ($3.938.771$ paquetes), Chicago 20110608 ($27.049.728$ paquetes), Mawi 201911021400 ($67.530.945$ paquetes) y Mawi 20209011400 ($123.897.490$ paquetes) Repetimos estas consultas en sketches con distintos valores de c.

Los elementos para los cuales se estima el rank están dados por la siguiente expresión, donde $0 \leq i < 11$, min es el tamaño del paquete más corto y max es el tamaño del paquete más largo:

$$c_i = \begin{cases} max \; si \; i = 10 \\ \frac{(max-min)*i}{10} + min \; en \; otro \; caso \end{cases}$$

Obteniendo los resultados de estas consultas, comparamos el resultado de la estimación con el rank real de los elementos consultados para obtener el error absoluto. Dado que la cota de error entregada por el sketch es $|R(x) - \hat{R}(x)| \leq \varepsilon n$, tiene sentido dividir el error absoluto por el número de elementos para poder compararlo mejor con el parámetro ε. Este error normalizado es el que usamos para evaluar el desempeño del sketch y se calcula de la siguiente manera:

$$\frac{|R(x) - \hat{R}(x)|}{n},$$

donde $\hat{R}(x)$ es la estimación del rank entregada por el sketch, $R(x)$ es el valor real del rank y n es el número de elementos insertados al sketch.

Las Tablas 5.2 a 5.5 muestran los resultados de estos experimentos, así como también el espacio que ocupa el sketch en kilobytes y la altura del sampler del sketch.

Tabla 5.2: Errores y espacio utilizado en función de c para Chicago 20080319

			Consultas										
c	**Kilobytes**	**H”**	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**
0,5100	25,41	10	0,0000	0,0123	0,0126	0,0112	0,0122	0,0114	0,0135	0,0138	0,0110	0,0120	0,0075
0,6667	28,20	3	0,0000	0,0046	0,0042	0,0040	0,0043	0,0049	0,0045	0,0058	0,0042	0,0042	0,0042
0,7500	32,29	0	0,0000	0,0023	0,0029	0,0029	0,0023	0,0027	0,0029	0,0024	0,0028	0,0028	0,0028
0,9000	57,42	0	0,0000	0,0005	0,0006	0,0008	0,0007	0,0007	0,0006	0,0006	0,0005	0,0007	0,0007

Tabla 5.3: Errores y espacio utilizado en función de c para Chicago 20110608

			Consultas										
c	**Kilobytes**	**H”**	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**
0,5100	25,41	10	0,0001	0,0041	0,0043	0,0051	0,0054	0,0048	0,0046	0,0044	0,0044	0,0045	0,0028
0,6667	28,21	3	0,0001	0,0018	0,0020	0,0019	0,0020	0,0020	0,0022	0,0021	0,0024	0,0020	0,0016
0,7500	32,29	0	0,0001	0,0016	0,0015	0,0010	0,0011	0,0010	0,0011	0,0017	0,0013	0,0012	0,0012
0,9000	57,42	0	0,0001	0,0006	0,0007	0,0005	0,0006	0,0006	0,0005	0,0005	0,0006	0,0005	0,0009

Tabla 5.4: Errores y espacio utilizado en función de c para Mawi 201911021400

			Consultas										
c	**Kilobytes**	**H”**	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**
0,5100	25,41	10	0,0030	0,0030	0,0016	0,0014	0,0014	0,0014	0,0014	0,0014	0,0014	0,0014	0,0014
0,6667	28,21	3	0,0014	0,0013	0,0014	0,0014	0,0014	0,0014	0,0014	0,0014	0,0014	0,0014	0,0014
0,7500	32,29	0	0,0007	0,0008	0,0009	0,0007	0,0007	0,0007	0,0007	0,0007	0,0007	0,0007	0,0007
0,9000	57,42	0	0,0005	0,0004	0,0006	0,0006	0,0006	0,0006	0,0006	0,0006	0,0006	0,0006	0,0006

Tabla 5.5: Errores y espacio utilizado en función de c para Mawi 202009011400

			Consultas										
c	**Kilobytes**	**H'**	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**
0,5100	25,41	10	0,0028	0,0011	0,0011	0,0011	0,0011	0,0011	0,0011	0,0011	0,0011	0,0011	0,0011
0,6667	28,21	3	0,0013	0,0014	0,0013	0,0013	0,0013	0,0013	0,0013	0,0013	0,0013	0,0013	0,0013
0,7500	32,29	0	0,0008	0,0007	0,0007	0,0007	0,0007	0,0007	0,0007	0,0007	0,0007	0,0007	0,0007
0,9000	57,42	0	0,0005	0,0003	0,0004	0,0004	0,0004	0,0004	0,0004	0,0004	0,0004	0,0004	0,0004

Al observar las tablas, podemos ver que si usamos un valor de c mayor o igual que $0{,}75$, la altura del sampler correspondiente es 0. Esto puede ocasionar una pérdida grande de datos, lo que contrarresta la mejora en la precisión observada. Además, podemos constatar que, a diferencia del sketch que usa $c = 0{,}51$, al definir un $c = 0{,}66666$ todas las consultas menos una cumplen la cota de error establecida, incluso probando con una traza que tiene un número de elementos mucho más pequeño que el n para el cual dimensionamos el sketch. Este valor de c también implica que el sampler del sketch tiene altura 3, lo cual es un valor manejable usando una memoria FIFO a la entrada del primer compactor. Concluimos entonces que $c = 0{,}66666$ es un valor adecuado para nuestra implementación.

Considerando los parámetros escogidos anteriormente, la estructura que usamos en la modelación RTL (Register Transfer Level, que es una abstracción de diseño que modela un circuito digital sincrónico en función de las señales digitales entre registros y las operaciones lógicas realizadas sobre estas señales) usa los siguientes parámetros $H = 20$, $H'' = 3$, $s = 4$ y $k_H = 1535$.

5.2. Selección del tamaño de la FIFO de entrada

Finalmente, debemos seleccionar el tamaño de la cola FIFO de nuestra arquitectura. Para esto, repetimos las consultas en la implementación en hardware del sketch, construido con los parámetros anteriores y variando únicamente el largo de la cola FIFO. Usamos la implementación en hardware debido a que debemos considerar el número de paquetes que se pierden debido a que el primer compactor no puede recibir elementos por encontrarse en proceso de compactación. Como mencionamos en la sección 4.5, el número de paquetes perdidos tiene que ver con la cantidad de ciclos del reloj que se demora la compactación. Es por esto que consideramos más adecuado hacer estos experimentos en la implementación en hardware utilizando simulación.

Para hacer estos experimentos se consideró el peor caso, en que en cada ciclo de reloj llega un nuevo paquete y estos son de tamaño mínimo. Para este peor caso, verificamos experimentalmente que para las trazas de prueba que usamos, una cola de 4096 elementos basta para no tener perdida de elementos. Sin embargo, usar una cola de este tamaño es tomar esta decisión basándose en una sobreestimación. En condiciones reales, los paquetes no llegan en todos los ciclos de reloj y no son todos

de tamaño mínimo. Es por esto que decidimos determinar el tamaño de la FIFO experimentalmente, considerando el error producido y la perdida de elementos en este peor caso.

Las Tablas 5.10 a 5.13 presentan los resultados de la experimentación, mostrando los errores absolutos normalizados por el número de elementos insertados en el sketch, el número de elementos que se perdieron a la salida del sampler, y el porcentaje de paquetes que estos elementos representan, asumiendo un peor caso donde llega un paquete por cada ciclo de reloj. Dado que tenemos un sampler de altura 3, se selecciona 1 paquete de cada 8, por lo que el porcentaje de paquetes perdidos está dado por:

$$\frac{100 * (8 * e_p)}{n_{real}},$$

donde e_p representa la cantidad de elementos perdidos y n_{real} representa la cantidad de elementos insertados en el sketch.

Tabla 5.6: Errores y pérdida de datos para Chicago 20080319

	Consultas												Pérdida de datos	
FIFO	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**Media**	**Elementos**	**% de paquetes**
32	0,0000	0,0044	0,0066	0,0049	0,0036	0,0040	0,0111	0,0037	0,0034	0,0005	0,0004	0,0039	8	0,0016
64	0,0000	0,0044	0,0061	0,0043	0,0033	0,0038	0,0111	0,0037	0,0030	0,0006	0,0003	0,0037	0	0,0000
128	0,0000	0,0044	0,0061	0,0043	0,0033	0,0038	0,0111	0,0037	0,0030	0,0006	0,0003	0,0037	0	0,0000
256	0,0000	0,0044	0,0061	0,0043	0,0033	0,0038	0,0111	0,0037	0,0030	0,0006	0,0003	0,0037	0	0,0000

Tabla 5.7: Errores y pérdida de datos para Chicago 20110608

	Consultas												Pérdida de datos	
FIFO	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**Media**	**Elementos**	**% de paquetes**
32	0,0000	0,0004	0,0013	0,0044	0,0011	0,0017	0,0006	0,0023	0,0018	0,0009	0,0007	0,0014	324	0,0096
64	0,0000	0,0001	0,0031	0,0008	0,0026	0,0002	0,0006	0,0017	0,0010	0,0002	0,0006	0,0010	62	0,0018
128	0,0000	0,0003	0,0004	0,0017	0,0010	0,0014	0,0020	0,0005	0,0018	0,0006	0,0006	0,0009	0	0,0000
256	0,0000	0,0003	0,0004	0,0017	0,0010	0,0014	0,0020	0,0005	0,0018	0,0006	0,0006	0,0009	0	0,0000

Tabla 5.8: Errores y pérdida de datos para Mawi 201911921400

	Consultas												Pérdida de datos	
FIFO	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**Media**	**Elementos**	**% de paquetes**
32	0,0013	0,0002	0,0003	0,0001	0,0002	0,0002	0,0002	0,0002	0,0002	0,0002	0,0002	0,0003	1.186	0,0140
64	0,0002	0,0016	0,0017	0,0000	0,0001	0,0001	0,0001	0,0001	0,0001	0,0001	0,0001	0,0004	484	0,0057
128	0,0002	0,0002	0,0007	0,0000	0,0001	0,0001	0,0001	0,0001	0,0001	0,0001	0,0001	0,0002	161	0,0019
256	0,0017	0,0002	0,0007	0,0000	0,0000	0,0000	0,0001	0,0001	0,0001	0,0001	0,0001	0,0003	0	0,0000

Tabla 5.9: Errores y pérdida de datos para Mawi 202009011400

	Consultas												Pérdida de datos	
FIFO	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**Media**	**Elementos**	**% de paquetes**
32	0,0002	0,0001	0,0001	0,0001	0,0001	0,0001	0,0001	0,0001	0,0001	0,0001	0,0001	0,0001	2.655	0,0171
64	0,0005	0,0000	0,0000	0,0000	0,0000	0,0000	0,0000	0,0000	0,0000	0,0000	0,0000	0,0001	1.424	0,0091
128	0,0012	0,0001	0,0000	0,0000	0,0000	0,0000	0,0000	0,0000	0,0000	0,0000	0,0000	0,0001	780	0,0050
256	0,0009	0,0001	0,0001	0,0001	0,0001	0,0001	0,0001	0,0001	0,0001	0,0001	0,0001	0,0002	58	0,0003

En los resultados del experimento podemos observar que para una cola FIFO de 128 elementos se obtienen estimaciones muy similares que para una cola de 256 elementos. Las estimaciones también mejoran con respecto a las colas de 32 y 64 elementos. Además, aun considerando el peor caso, tenemos una pérdida de paquetes menor o igual a 0.005 %. Consideramos por ende que una cola FIFO de 128 elementos es adecuada para esta aplicación.

Capítulo 6. Resultados

6.1. Validación de la implementación

Validamos la implementación de la arquitectura a 3 niveles: simulación de tipo "behavioral" (funcional), simulación post-síntesis, e implementación. Para la validación usamos 10 trazas de tráfico de red. De estas, 5 provienen del repositorio de CAIDA (Center for Applied Internet Data Analysis) y fueron recolectadas en el enlace entre Equinix y Chicago [12, 13, 14, 16]. Por otro lado, las otras 5 provienen de MAWILab [15], un repositorio que asiste a investigadores a evaluar métodos de detección de anomalías en el tráfico. La Tabla 6.1 presenta el nombre y número de paquetes de cada una de estas trazas. La versión de Vivado utilizada para realizar estas validaciones es la versión 2023.1.

En primer lugar, validamos la implementación utilizando simulación de tipo funcional, dado que este es el tipo de simulación más rápido que entrega resultados con precisión de un ciclo de reloj, por lo que es útil para verificar la lógica implementada. Usando estas simulaciones, verificamos primeramente que el sampler estuviera escogiendo los mismos elementos que la implementación en software del algoritmo. Luego, verificamos para tres trazas (Mawi 201812011400, Mawi 201711011400 y Chicago 20110608) que los elementos escogidos e insertados al compactor inmediatamente superior para cada compactación fueran los mismos que en la implementación de referencia en todos los compactors. Finalmente, comprobamos que los resultados de rank para cada una de las consultas fueran idénticos a la implementación software para todas las trazas. Las implementaciones software de referencia se pueden encontrar en [39].

Luego, usando simulación post-síntesis, la cual toma en cuenta ya no solo la funcionalidad, sino que simula el hardware generado por el compilador estimando retrasos asociados a este, comparamos los resultados obtenidos en 3 trazas (Chicago 20180319, Chicago 20080515 y Chicago 20150219) con aquellos obtenidos en simulación funcional, observando que no hay diferencias. No hicimos esta comprobación con todas las trazas, ya que el tiempo de simulación es extenso.

Tabla 6.1: Trazas de red usadas para validación y estimación

Nombre	Número de paquetes
Chicago-20080319	3.938.771
Chicago-20080515	12.242.731
Chicago-20110608	27.049.728
Chicago-20150219	15.982.586
Chicago-20160121	32.472.976
Mawi-201612011400	99.795.401
Mawi-201711011400	122.397.817
Mawi-201812011400	80.707.468
Mawi-201911021400	67.530.945
Mawi-202009011400	123.897.490

En vez de eso, implementamos la arquitectura en una tarjeta Alveo U55C. Para esto, usamos la plataforma Vitis, plataforma de desarrollo que provee Xilinx para el desarrollo de aceleradores. Definimos cuatro kernels. Los primeros 3 nos permiten comunicarnos con el host para enviar los datos de entrada, las consultas y poder recibir las estimaciones de rank y el cuarto implementa la arquitectura descrita en este trabajo. Constatamos que, tanto para una cola FIFO de 4096 elementos donde no hay perdida de paquetes, como para una cola FIFO de 128 donde se pierden algunos paquetes, las estimaciones de rank coinciden con las obtenidas anteriormente. De esta manera, consideramos nuestro diseño validado correctamente.

6.2. Descripción del Hardware

La tarjeta de desarrollo en la que implementamos nuestra arquitectura es una Xilinx Alveo U55C, la cual contiene un chip Virtex XCU55 UltraScale+. Esta tarjeta posee alrededor de 1.304.000 tablas Look-Up, 2.607.000 registros y 9.024 bloques DSP. Además, en cuanto a memoria, posee $36{,}7$ Mb de memoria distribuida, $70{,}9$ Mb de Block RAM y 270 Mb de UltraRAM. Esta tarjeta está conectada por PCI Express a un servidor con un procesador Intel Core i9-10980XE de 18 núcleos, 128GB de memoria RAM DDR4 y un disco duro SSD de tipo NVMe de 1TB de capacidad. El host utiliza el

Tabla 6.2: Uso de recursos de la arquitecura

Recurso	Usados	Disponible	Porcentaje
LUT	5.960	1.303.680	0,46
Memoria Distribuida	860	600.960	0,14
Registros	3.675	2.607.360	0,14
Block RAMs	8	2.016	0,40
Entradas/Salidas	103	624	16,51
Buffers	1	1.008	0,10

sistema operativo Ubuntu 20.04.6 LTS con un kernel Linux 5.15.0.

6.3. Uso de recursos y frecuencia de reloj

Al implementar la arquitectura en la tarjeta descrita anteriormente, el uso de recursos reportado por Vivado se detalla en la Tabla 6.2. Podemos observar que el porcentaje de utilización de los recursos es menor al 1 % excepto para las entradas y salidas. Esto se debe a que la interfaz con la memoria para la lectura de datos de entrada ocupa gran parte de estos recursos. El sistema alcanza una frecuencia de reloj de 325.2 MHz lo que nos da un flujo de aproximadamente 325.2 millones de paquetes por segundo. Si consideramos el peor caso en que los paquetes son todos de largo 64 bytes, tendríamos una tasa mínima de 166 Gbps. Al agregar al sistema los kernels que permiten la comunicación del host con la tarjeta de desarrollo, el uso de recursos aumenta como se presenta en la Tabla 6.3. Al comparar ambas tablas, podemos afirmar que el uso de recursos se encuentra dominado por los elementos adicionales que tuvimos que agregar para validar experimentalmente la arquitectura. Sin embargo, en una aplicación realista de monitoreo en tiempo real, muchos de los elementos añadidos, tales como las interfaces de lectura de la memoria, son innecesarios. Además, la frecuencia de reloj alcanzada se reduce a 293.0 MHz, lo que nos entrega un flujo de 293 millones de paquetes por segundo.

Tabla 6.3: Uso de recursos del sistema completo conectado al host

Recurso	Usados	Disponible	Porcentaje
LUT	132.220	1.303.680	10,14
Memoria Distribuida	9.967	600.960	1,66
Registros	175.376	2.607.360	6,73
Block RAMs	281	2.016	13,94
Entradas/Salidas	12	624	1,92
Buffers	42	1.008	4,17

6.4. Tiempos de ejecución

La Tabla 6.4 reporta los tiempos de ejecución en milisegundos de la inserción de elementos y la etapa de consultas para una implementación en software del sketch usando una estructura dinámica (KLLDS), una implementación en software de la estructura estática propuesta usando LFSR (KLLSS) y la implementación en hardware de la arquitectura propuesta conectada con el host descrito anteriormente (KLLHS). Adicionalmente, reportamos en la Tabla 6.5 la aceleración de cada etapa y la aceleración total de la implementación en hardware comparada con la implementación de la estructura estática en software. La aceleración en cada etapa se calcula dividiendo el tiempo de ejecución en software por el tiempo de ejecución de hardware. Además, la aceleración total se calcula de la siguiente forma:

$$\frac{t_{insert_s} + t_{query_s}}{t_{insert_h} + t_{query_h}},$$

donde t_{insert_s} y t_{query_s} son los tiempos de inserción y consulta en la implementación en software del sketch modificado y t_{insert_h} y t_{query_h} son los tiempos de inserción y consulta de hardware.

Podemos ver que la mayor aceleración se produce en la operación de inserción de los elementos al sketch, con una aceleración máxima de $52{,}49$ veces y una aceleración mínima de $37{,}00$ veces. A pesar de que la aceleración en las consultas es menor a 1 para algunas trazas, dado que este tiempo es despreciable comparado con el tiempo de inserción, el tiempo de ejecución total se reduce más de 35 veces para todas las trazas, llegando a una aceleración máxima de $43{,}18$ veces. Además, si calculamos el número de paquetes por segundo podemos ver que obtenemos un número entre 271 y 285 millones.

Tabla 6.4: Tiempo de ejecución (ms)

	KLLDS		KLLSS		KLLHS	
Traza	**Inserción**	**Consultas**	**Inserción**	**Consultas**	**Inserción**	**Consultas**
Chicago 20080319	2.205,73	5,74	760,53	1,17	14,49	3,15
Chicago 20080515	5.339,31	8,01	1.635,22	1,39	43,48	3,16
Chicago 20110608	13.669,57	3,47	3.576,24	2,39	95,44	3,15
Chicago 20150219	7.283,31	5,52	2.089,57	1,81	56,47	3,15
Chicago 20160121	17.304,04	4,26	4.253,82	2,11	114,56	3,40
Mawi 201612011400	70.997,80	6,56	13.017,54	5,36	350,65	3,27
Mawi 201711011400	86.578,83	3,88	16.065,57	3,59	429,98	3,45
Mawi 201812011400	54.873,10	4,18	10.526,61	4,63	283,70	3,14
Mawi 201911021400	43.789,91	4,46	8.824,97	4,09	237,41	3,15
Mawi 202009011400	87.578,79	5,09	16.204,48	4,37	435,22	3,28

Esta ligera diferencia con el número de paquetes por segundo teórico se puede deber al tiempo que se demora en empezar a ejecutarse el kernel, retardos asociados a la interfaz de comunicación con la memoria o simplemente a la precisión con la que se pueden medir los tiempos de ejecución.

6.5. Estimaciones de rank

Las Tablas 6.6 a 6.14 reportan los errores absolutos de estimación normalizados por el número de paquetes insertados para todas las trazas de prueba.Recordemos que estos errores normalizados son calculados de la siguiente manera: $\frac{|R(x)-\hat{R}(x)|}{n}$, donde $\hat{R}(x)$ es la estimación del rank entregada por el sketch, $R(x)$ es el valor real del rank y n es el número de elementos insertados al sketch. Para estas pruebas, usamos como entrada un archivo el cual contiene el largo de los paquetes para cada traza. Una vez insertados estos elemnetos al sketch, se realizan las 11 consultas de rank descritas en las secciones anteriores. Realizaremos una comparación entre los errores para la implementación dinámica del sketch en software “KLLDS”, la implementación en software del sketch modificado para usar una estructura estática “KLLSS”, y la implementación en hardware de la arquitectura que proponemos “KLLSH”. Podemos observar que, aunque los errores aumentan al utilizar una estructura

Tabla 6.5: Aceleración en el tiempo de ejecución

Traza	Inserción	Consultas	Total
Chicago 20080319	52,49	0,37	43,18
Chicago 20080515	37,61	0,44	35,09
Chicago 20110608	37,47	0,76	36,30
Chicago 20150219	37,00	0,57	35,08
Chicago 20160121	37,13	0,62	36,08
Mawi 201612011400	37,12	1,64	36,80
Mawi 201711011400	37,36	1,04	37,07
Mawi 201812011400	37,10	1,47	36,71
Mawi 201911021400	37,17	1,30	36,70
Mawi 202009011400	37,23	1,33	36,96
Promedio	38,77	0,96	37,00

estática, para trazas con un número de paquetes mayor a 16 millones, se cumple la cota de error de $0{,}005$ para todas las consultas. Para la trazas más pequeñas, la cota de error se cumple para 9 de las 11 consultas. En promedio, podemos osbservar que el error obtenido en el sketch estático en software es similar al obtenido en hardware, por lo que podemos afirmar que las pequeñas perdidas de paquetes no afectan de manera significitiva la estimación.

Tabla 6.6: Comparación de errores absolutos normalizados para Chicago 20080319

	Consultas											
Implementación	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**Media**
KLLDS	0,0000	0,0010	0,0011	0,0004	0,0003	0,0013	0,0010	0,0000	0,0008	0,0003	0,0001	0,0006
KLLSS	0,0000	0,0044	0,0061	0,0043	0,0033	0,0038	0,0111	0,0037	0,0030	0,0006	0,0003	0,0037
KLLSH	0,0000	0,0044	0,0061	0,0043	0,0033	0,0038	0,0111	0,0037	0,0030	0,0006	0,0003	0,0037

Tabla 6.7: Comparación de errores absolutos normalizados para Chicago 20080515

	Consultas											
Implementación	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**Media**
KLLDS	0,0000	0,0001	0,0005	0,0009	0,0002	0,0001	0,0002	0,0001	0,0001	0,0001	0,0002	0,0002
KLLSS	0,0000	0,0029	0,0061	0,0003	0,0006	0,0017	0,0058	0,0004	0,0035	0,0035	0,0035	0,0026
KLLSH	0,0000	0,0029	0,0061	0,0003	0,0006	0,0017	0,0058	0,0004	0,0035	0,0035	0,0035	0,0026

Tabla 6.8: Comparación de errores absolutos normalizados para Chicago 20110608

	Consultas											
Implementación	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**Media**
KLLDS	0,0012	0,0002	0,0011	0,0001	0,0006	0,0010	0,0007	0,0005	0,0010	0,0018	0,0012	0,0009
KLLSS	0,0000	0,0003	0,0004	0,0017	0,0010	0,0014	0,0020	0,0005	0,0018	0,0006	0,0006	0,0009
KLLSH	0,0000	0,0003	0,0004	0,0017	0,0010	0,0014	0,0020	0,0005	0,0018	0,0006	0,0006	0,0009

Tabla 6.9: Comparación de errores absolutos normalizados para Chicago 20150219

	Consultas											
Implementación	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**Media**
KLLDS	0,0000	0,0002	0,0006	0,0003	0,0002	0,0004	0,0004	0,0001	0,0006	0,0009	0,0003	0,0004
KLLSS	0,0000	0,0051	0,0004	0,0048	0,0038	0,0024	0,0031	0,0022	0,0001	0,0017	0,0005	0,0022
KLLSH	0,0000	0,0051	0,0004	0,0048	0,0038	0,0024	0,0031	0,0022	0,0001	0,0017	0,0005	0,0022

Tabla 6.10: Comparación de errores absolutos normalizados para Chicago 20160121

	Consultas											
Implementación	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**Media**
KLLDS	0,0011	0,0006	0,0012	0,0001	0,0010	0,0010	0,0005	0,0005	0,0009	0,0013	0,0010	0,0008
KLLSS	0,0008	0,0007	0,0023	0,0004	0,0003	0,0016	0,0031	0,0025	0,0001	0,0011	0,0012	0,0013
KLLSH	0,0012	0,0004	0,0022	0,0008	0,0001	0,0016	0,0025	0,0028	0,0005	0,0008	0,0012	0,0013

Tabla 6.11: Comparación de errores absolutos normalizados para Mawi 201612011400

	Consultas											
Implementación	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**Media**
KLLDS	0,0001	0,0004	0,0004	0,0004	0,0004	0,0003	0,0001	0,0003	0,0002	0,0003	0,0002	0,0003
KLLSS	0,0009	0,0005	0,0001	0,0020	0,0001	0,0012	0,0005	0,0004	0,0001	0,0004	0,0000	0,0006
KLLSH	0,0011	0,0004	0,0002	0,0001	0,0014	0,0009	0,0009	0,0019	0,0006	0,0004	0,0000	0,0007

Tabla 6.12: Comparación de errores absolutos normalizados para Mawi 201711011400

	Consultas											
Implementación	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**Media**
KLLDS	0,0010	0,0004	0,0007	0,0008	0,0001	0,0011	0,0011	0,0006	0,0003	0,0004	0,0000	0,0006
KLLSS	0,0001	0,0006	0,0001	0,0002	0,0016	0,0013	0,0010	0,0007	0,0014	0,0004	0,0000	0,0007
KLLSH	0,0013	0,0020	0,0012	0,0026	0,0016	0,0003	0,0009	0,0006	0,0009	0,0004	0,0000	0,0011

Tabla 6.13: Comparación de errores absolutos normalizados para Mawi 201812011400

	Consultas											
Implementación	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**Media**
KLLDS	0,0005	0,0006	0,0001	0,0005	0,0004	0,0004	0,0004	0,0004	0,0004	0,0004	0,0004	0,0004
KLLSS	0,0016	0,0006	0,0009	0,0006	0,0005	0,0005	0,0005	0,0005	0,0005	0,0005	0,0005	0,0007
KLLSH	0,0002	0,0002	0,0011	0,0006	0,0005	0,0005	0,0005	0,0005	0,0005	0,0005	0,0005	0,0005

Tabla 6.14: Comparación de errores absolutos normalizados para Mawi 201911021400

	Consultas											
Implementación	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**Media**
KLLDS	0,0006	0,0000	0,0007	0,0009	0,0008	0,0008	0,0008	0,0008	0,0008	0,0008	0,0008	0,0007
KLLSS	0,0017	0,0002	0,0007	0,0000	0,0000	0,0000	0,0001	0,0001	0,0001	0,0001	0,0001	0,0003
KLLSH	0,0002	0,0002	0,0007	0,0000	0,0001	0,0001	0,0001	0,0001	0,0001	0,0001	0,0001	0,0002

Tabla 6.15: Comparación de errores absolutos normalizados para Mawi 20209011400

	Consultas											
Implementación	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**Media**
KLLDS	0,0004	0,0000	0,0000	0,0000	0,0000	0,0000	0,0000	0,0000	0,0000	0,0000	0,0000	0,0000
KLLSS	0,0004	0,0001	0,0001	0,0001	0,0001	0,0001	0,0001	0,0001	0,0001	0,0001	0,0001	0,0001
KLLSH	0,0012	0,0001	0,0000	0,0000	0,0000	0,0000	0,0000	0,0000	0,0000	0,0000	0,0000	0,0001

Capítulo 7. Conclusiones y trabajo futuro

En este trabajo propusimos la arquitectura de un acelerador hardware para la estimación de cuantiles en tráfico de redes. Para diseñar esta arquitectura, nos basamos en la estructura general del sketch KLL, modificándola para hacerla más amigable con las características estáticas del hardware.

Observando los resultados, podemos constatar al comparar ambas implementaciones en software que, aunque al utilizar una estructura estática el error aumenta, cuando el número de paquetes de la traza es mayor al 10 % del número de paquetes estimado que escogimos, se sigue respetando la cota de error de $\varepsilon = 0{,}005$ para todas las consultas realizadas. El usar esta estructura estática nos permite simplificar en gran medida la implementación del sketch. Esto se debe a que para implementar la estructura dinámica, existen dos caminos posibles:

a. Por un lado, se puede usar la estructura final que se genera tras un cierto número de elementos insertados, pero ir variando el compactor de inserción de los elementos en los casos en que la estructura crece en la implementación en software. Esto implica implementar la lógica de control que decida donde se insertarán los elementos. Además, significa tener que mover los elementos de cada compactor un nivel más abajo cada vez que el compactor de mayor altura es compactado. Finalmente, mover estos elementos a un compactor de tamaño inferior puede provocar procesos de compactación adicionales. Estos factores pueden afectar de manera significativa el desempeño del acelerador, debido a que aumenta el número de ciclos en los que no se puede insertar un nuevo elemento al compactor de más abajo, lo que perjudica tanto la tasa de procesamiento efectiva como la precisión de las estimaciones.

b. Por otro lado, se puede tener una estructura de compactors de tamaño máximo, en los cuales el número de elementos que almacenan realmente vaya disminuyendo a medida que el sketch crece y los elementos sean insertados por el compactor inferior. Aunque esto nos ahorra el problema de mover los elementos de un compactor al compactor inmediatamente inferior, aumenta considerablemente el consumo en memoria, la cual estará subutilizada en su mayor parte por lo que es mejor utilizar un algoritmo que aproveche la memoria en su totalidad tal como el MRL [18]

o el propuesto por Agarwal et al. [21]. Además, se mantiene el problema de las compactaciones adicionales provocadas por la disminución del tamaño efectivo de los compactors.

Debido a esto, creemos que utilizar una estructura estática para diseñar la arquitectura fue una decisión acertada para esta aplicación.

Si comparamos el error promedio para la implementación estática en software y la implementación estática en hardware, podemos observar que los errores son similares para la mayoría de las trazas y respetan la cota de error establecida en todas ellas. A partir de esta observación podemos afirmar que los factores diferenciadores entre el software y hardware, es decir, el uso de LFSR para la generación de números aleatorios y la pequeña perdida de paquetes, no perjudican de manera significativa los errores de estimación.

Cabe notar que el uso de recursos de la arquitectura mostrado en la Tabla 6.2 es bajo. Podemos ver que los recursos de memoria tales como Block RAMs y memoria distribuida (LUTRAM) no se ven exigidos, por lo que de querer obtener una mayor precisión, o querer analizar un mayor número de paquetes, se puede aumentar el tamaño del sketch. Además, esto nos permite tener varias copias del sketch en una misma tarjeta, por lo que podríamos analizar múltiples flujos de datos o estimar cuantiles para múltiples propiedades de manera simultánea.

Asimismo, se observa una aceleración de más de dos órdenes de magnitud con respecto al software, tanto en tiempo de ejecución total (tiempo por lo menos 35 veces menor) como en tiempo de inserción de elementos al sketch (por lo menos 37 veces menor). Esta aceleración está limitada por la frecuencia de reloj alcanzada (325.2 MHz en la arquitectura, 293.0 MHz en la validación experimental en el host), la cual está dada por el camino crítico de la arquitectura. Este camino crítico se encuentra en la determinación de los punteros de lectura y escritura en la inserción de elementos para los compactors más grandes, debido a que estos se determinan de manera combinacional. Si queremos obtener frecuencias de reloj más altas, se puede explorar la posibilidad de dividir este procesamiento en dos ciclos de reloj o más.

Este trabajo se puede extender en un futuro en dos direcciones posibles. Por un lado, se puede

explorar la posibilidad de combinar este sketch con otra estructura de datos que permita estimar ranks para propiedades acumuladas del tráfico en redes, tales como el payload o la cantidad de paquetes por flujo, donde un flujo son todos los paquetes que tienen ciertas características en común, como por ejemplo, dirección IP y puerto de origen y dirección IP y puerto de destino. Para esto, se pueden usar sketches de estimación de frecuencia tales como el CountMin o el Count Sketch. Estas estructuras nos permiten estimar la distribución de propiedades de flujo y acumular las prioridades de los paquetes de cada flujo. De esta manera, se puede utilizar un buffer entre el sketch de estimación de frecuencia y el sketch de estimación de cuantiles, el cual almacena solo los elementos con mayor magnitud en el sketch de cuentas. Estos elementos son los que tienen mayor relevancia dado que, al detectar anomalías, se busca detectar flujos con un número inusualmente grande de paquetes o con un payload inusualmente alto. Al usar el sketch de cuantiles para entender la distribución de los elementos con mayor payload o mayor número de paquetes, se pueden detectar estos flujos inusuales. Por otro lado, se puede implementar una arquitectura que utilice la modificación a este sketch llamada KLL$\pm$, la cual permite eliminaciones acotadas. Como describen en el artículo en el que se propuso esta modificación [40], se pueden usar 3 de estos sketches para realizar un estudio de los cuantiles en ventanas deslizantes. Esto es relevante, ya que permite, por ejemplo, detectar cambios bruscos entre una ventana y otra que puedan indicar una anomalía.

Referencias

[1] Z. Karnin, K. Lang, and E. Liberty, "Optimal quantile approximation in streams," in *2016 ieee 57th annual symposium on foundations of computer science (focs)*. IEEE, 2016, pp. 71–78.

[2] H. Han, Z. Yan, X. Jing, and W. Pedrycz, "Applications of sketches in network traffic measurement: A survey," *Information Fusion*, vol. 82, pp. 58–85, 2022.

[3] N. Ivkin, Z. Yu, V. Braverman, and X. Jin, "Qpipe: Quantiles sketch fully in the data plane," in *Proceedings of the 15th International Conference on Emerging Networking Experiments And Technologies*, 2019, pp. 285–291.

[4] V. M. Tellis and D. J. D'Souza, "Detecting anomalies in data stream using efficient techniques: a review," in *2018 international conference on control, power, communication and computing technologies (ICCPCCT)*. IEEE, 2018, pp. 296–298.

[5] A. D'Alconzo, I. Drago, A. Morichetta, M. Mellia, and P. Casas, "A survey on big data for network traffic monitoring and analysis," *IEEE Transactions on Network and Service Management*, vol. 16, no. 3, pp. 800–813, 2019.

[6] F. Hauser, M. Häberle, D. Merling, S. Lindner, V. Gurevich, F. Zeiger, R. Frank, and M. Menth, "A survey on data plane programming with p4: Fundamentals, advances, and applied research," *Journal of Network and Computer Applications*, vol. 212, p. 103561, 2023.

[7] D. Tong and V. K. Prasanna, "Sketch acceleration on fpga and its applications in network anomaly detection," *IEEE Transactions on Parallel and Distributed Systems*, vol. 29, no. 4, pp. 929–942, 2017.

[8] P. Papaphilippou and W. Luk, "Accelerating database systems using fpgas: A survey," in *2018 28th International Conference on Field Programmable Logic and Applications (FPL)*. IEEE, 2018, pp. 125–1255.

[9] J. Zhang and J. Li, "Improving the performance of opencl-based fpga accelerator for convolutional neural network," in *Proceedings of the 2017 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, 2017, pp. 25–34.

[10] J. I. Munro and M. S. Paterson, "Selection and sorting with limited storage," *Theoretical computer science*, vol. 12, no. 3, pp. 315–323, 1980.

[11] G. Cormode, M. Garofalakis, P. J. Haas, C. Jermaine *et al.*, "Synopses for massive data: Samples, histograms, wavelets, sketches," *Foundations and Trends® in Databases*, vol. 4, no. 1–3, pp. 1–294, 2011.

[12] "Anonymized Internet Traces 2008," https://catalog.caida.org/dataset/passive_2008_pcap.

[13] "Anonymized Internet Traces 2011," https://catalog.caida.org/dataset/passive_2011_pcap.

[14] "Anonymized Internet Traces 2015," https://catalog.caida.org/dataset/passive_2015_pcap.

[15] R. Fontugne, P. Borgnat, P. Abry, and K. Fukuda, "Mawilab: Combining diverse anomaly detectors for automated anomaly labeling and performance benchmarking," in *ACM CoNEXT '10*, Philadelphia, PA, December 2010.

[16] "Anonymized Internet Traces 2016," https://catalog.caida.org/dataset/passive_2016_pcap.

[17] G. S. Manku, S. Rajagopalan, and B. G. Lindsay, "Approximate medians and other quantiles in one pass and with limited memory," *ACM SIGMOD Record*, vol. 27, no. 2, pp. 426–435, 1998.

[18] G. S. Manku, S. Rajagopalan, and B. G. Lindsay, "Random sampling techniques for space efficient online computation of order statistics of large datasets," *ACM SIGMOD Record*, vol. 28, no. 2, pp. 251–262, 1999.

[19] M. Greenwald and S. Khanna, "Space-efficient online computation of quantile summaries," *ACM SIGMOD Record*, vol. 30, no. 2, pp. 58–66, 2001.

[20] D. Felber and R. Ostrovsky, "A randomized online quantile summary in $o(\frac{1}{\varepsilon}\log\frac{1}{\varepsilon})$ words," *Theory of Computing*, vol. 13, no. 1, pp. 1–17, 2017.

[21] P. K. Agarwal, G. Cormode, Z. Huang, J. M. Phillips, Z. Wei, and K. Yi, "Mergeable summaries," *ACM Transactions on Database Systems (TODS)*, vol. 38, no. 4, pp. 1–28, 2013.

[22] E. Gan, J. Ding, K. S. Tai, V. Sharan, and P. Bailis, "Moment-based quantile sketches for efficient high cardinality aggregation queries," *arXiv preprint arXiv:1803.01969*, 2018.

[23] C. Masson, J. E. Rim, and H. K. Lee, "Ddsketch: A fast and fully-mergeable quantile sketch with relative-error guarantees," *arXiv preprint arXiv:1908.10693*, 2019.

[24] I. Epicoco, C. Melle, M. Cafaro, M. Pulimeno, and G. Morleo, "Uddsketch: Accurate tracking of quantiles in data streams," *IEEE Access*, vol. 8, pp. 147 604–147 617, 2020.

[25] G. Cormode, Z. Karnin, E. Liberty, J. Thaler, and P. Veselỳ, "Relative error streaming quantiles," in *Proceedings of the 40th ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems*, 2021, pp. 96–108.

[26] N. Ivkin, E. Liberty, K. Lang, Z. Karnin, and V. Braverman, "Streaming quantiles algorithms with small space and update time," *Sensors*, vol. 22, no. 24, p. 9612, 2022.

[27] I. Amezzane, Y. Fakhri, M. El Aroussi, and M. Bakhouya, "Hardware acceleration of svm training for real-time embedded systems: Overview," in *Recent Advances in Mathematics and Technology: Proceedings of the First International Conference on Technology, Engineering, and Mathematics, Kenitra, Morocco, March 26-27, 2018*. Springer, 2020, pp. 131–139.

[28] Y.-K. Lai, C.-L. Tsai, C.-H. Chuang, X.-W. Ku, and J. H. Chen, "Tabular interpolation approach based on stable random projection for estimating empirical entropy of high-speed network traffic," *IEEE Access*, vol. 10, pp. 104 934–104 953, 2022.

[29] Y.-K. Lai, S.-Y. Yu, I.-S. Chan, B.-H. Huang, C.-H. Chang, J. H. Chen, and J. Mambretti, "Sketch-based entropy estimation: a tabular interpolation approach using p4," in *Proceedings of the 5th International Workshop on P4 in Europe*, 2022, pp. 57–60.

[30] K. Yang, S. Long, Q. Shi, Y. Li, Z. Liu, Y. Wu, T. Yang, and Z. Jia, "Sketchint: Empowering int with towersketch for per-flow per-switch measurement," *IEEE Transactions on Parallel and Distributed Systems*, 2023.

[31] A. Saavedra, C. Hernández, and M. Figueroa, "Heavy-hitter detection using a hardware sketch with the countmin-cu algorithm," in *2018 21st Euromicro Conference on Digital System Design (DSD)*. IEEE, 2018, pp. 38–45.

[32] J. E. Soto, P. Ubisse, C. Hernández, and M. Figueroa, "A hardware accelerator for entropy estimation using the top-k most frequent elements," in *2020 23rd Euromicro Conference on Digital System Design (DSD)*. IEEE, 2020, pp. 141–148.

[33] J. E. Soto, P. Ubisse, Y. Fernández, C. Hernández, and M. Figueroa, "A high-throughput hardware accelerator for network entropy estimation using sketches," *IEEE Access*, vol. 9, pp. 85 823–85 838, 2021.

[34] Y. Fernández, J. E. Soto, S. Vera, Y. Prieto, C. Hernández, and M. Figueroa, "A streaming algorithm and hardware accelerator to estimate the empirical entropy of network flows," *Computer Networks*, p. 110035, 2023.

[35] A. Sateesan, J. Vliegen, S. Scherrer, H.-C. Hsiao, A. Perrig, and N. Mentens, "Speed records in network flow measurement on fpga," in *2021 31st International Conference on Field-Programmable Logic and Applications (FPL)*. IEEE, 2021, pp. 219–224.

[36] A. Ebrahim, "Finding the top-k heavy hitters in data streams: A reconfigurable accelerator based on an fpga-optimized algorithm," *Electronics*, vol. 12, no. 11, p. 2376, 2023.

[37] A. Partow, "Primitive polynomial list," https://www.partow.net/programming/polynomials/index.html.

[38] "Primitive polynomials (mod 2)," https://www.ams.org/journals/mcom/1962-16-079/S0025-5718-1962-0148256-1/S0025-5718-1962-0148256-1.pdf.

[39] "Repositorio Implementación Software del KLL," https://github.com/caritogap/ImplementacionKLL.

[40] F. Zhao, S. Maiyya, R. Wiener, D. Agrawal, and A. E. Abbadi, "Kll±approximate quantile sketches over dynamic datasets," *Proceedings of the VLDB Endowment*, vol. 14, no. 7, pp. 1215–1227, 2021.

UNIVERSIDAD DE CONCEPCION – FACULTAD DE INGENIERIA
RESUMEN DE MEMORIA DE TITULO

Departamento	**:** Departamento de Ingeniería Eléctrica
Carrera	: Ingeniería civil electrónica
Nombre del memorista	: Carolina Sofía Gallardo Pavesi
Título de la memoria	**:** Acelerador hardware para la estimación de cuantiles en tráfico de redes
Fecha de la presentación oral	: 29/04/2024
Profesor(es) guía	: Miguel Figueroa T. Cecilia Hernández R.
Profesor(es) revisor(es)	: Sergio Sobarzo G.
Concepto	**:**
Calificación	**:**

Resumen (máximo 200 palabras)

Las propiedades estadísticas del tráfico, tales como los cuantiles, nos ayudan a entender la distribución del tráfico y detectar anomalías a corto y largo plazo. Calcular exactamente el valor de estas propiedades es costoso en espacio y tiempo, ya que es imposible encontrar una solución exacta en una pasada sin almacenar todos los datos. Utilizar un acelerador hardware basado en arreglos de lógica programable (FPGA) nos permite procesar un stream de paquetes a alta velocidad y producir resultados rápidamente. Sin embargo, su memoria interna es limitada y almacenar todos los elementos no es viable. Por esto, se han propuesto algoritmos basados en sketches: estructuras probabilísticas que usan memoria sublineal para estimar una propiedad con una cota de error predeterminada.

Este trabajo presenta la arquitectura de un acelerador hardware para estimación de cuantiles. La arquitectura se basa en el sketch KLL y fue implementada en un FPGA Virtex™ XCU55 UltraScale+, obteniendo un tiempo de ejecución al menos 35 veces menor que la implementación en software. El acelerador opera a 325.2 MHz. Esto equivale a una tasa mínima de 166Gbps. Mostramos empíricamente que las estimaciones mantienen una buena precisión cuando el número de paquetes insertados es parecido al número de elementos estimado.