Universidad de Concepción
Facultad de Ingeniería
Departamento de Ingeniería Eléctrica

Profesores Guía:
Dr. Miguel E. Figueroa T.
Dra. Cecilia P. Hernández R.

Algoritmo y acelerador hardware para la estimación de cuantiles en propiedades de flujos de red

Carolina Gallardo Pavesi

Informe de Tesis

Magíster en Ciencias de la Ingeniería c/m en Ingeniería Eléctrica

Resumen

La medición y análisis del tráfico de datos es fundamental para la gestión y seguridad de redes de datos, ya que nos permite detectar e identificar elementos o eventos que no coinciden con el comportamiento típico, llamados anomalías. Estas anomalías nos ayudan a detectar posibles fallas o ataques. Una red de datos puede ser modelada por conexiones entre dos puntos llamadas flujos. Al analizar la distribución de ciertas propiedades de estos flujos, tales como el número de paquetes (denominado como frecuencia en la literatura) o la cantidad de bytes útiles enviados (payload), podemos detectar anomalías en el corto y largo plazo. Estas distribuciones pueden ser caracterizadas usando cuantiles, donde el cuantil de un elemento es la fracción de elementos del conjunto que son menores o iguales a este. No obstante, para poder tomar acciones correctivas oportunamente, el análisis de esta distribución debe realizarse de manera rápida y en línea. Dado los grandes volúmenes de tráfico y la velocidad de los enlaces modernos, esto es costoso computacionalmente, por lo que un procesador de propósito general es insuficiente para abordar el problema y utilizar aceleradores hardware se vuelve una opción atractiva. Sin embargo, estos dispositivos poseen una cantidad de memoria interna reducida, lo que hace imposible calcular en línea el valor exacto de estos cuantiles para trazas de tamaño típico en redes de alta velocidad, ya que implica almacenar las frecuencias para cada flujo. Como una forma de abordar este problema, se han propuesto estructuras de datos probabilísticas, llamadas sketches, que permiten estimar propiedades de grandes volúmenes de datos, tales como los cuantiles, con una eficiencia espacial alta. Sin embargo, dado que ciertas propiedades de los flujos, tales como la frecuencia, se acumulan a través de múltiples paquetes, éstas deben actualizarse en el sketch cada vez que llega un nuevo paquete. Los algoritmos que permiten trabajar con este modelo asumen un universo (el rango al que pueden pertenecer los elementos insertados) fijo y el espacio utilizado depende del tamaño de este universo, por lo que utilizarlos se vuelve poco conveniente. Por otra parte, se puede utilizar un sketch de estimación de frecuencia que permita acumular el número de paquetes por flujo y realizar el análisis de cuantiles sobre esta información. No obstante, estimar la frecuencia de todos los flujos mediante este sketch no es compatible con las restricciones de memoria ya que requeriría almacenar el identificador de cada flujo en la traza.

En trabajos recientes, se ha demostrado que, almacenando únicamente la frecuencia de los flujos más grandes, es posible aproximar la distribución de la frecuencia del resto de los flujos mediante suposiciones respecto a su distribución estadística [1]. En este trabajo, proponemos un algoritmo de estimación de cuantiles que combina un sketch de estimación de frecuencia junto con otras estructuras de datos para estimar la frecuencia de los flujos más grandes. Usando estas frecuencias, estimamos los parámetros de una distribución estadística de tipo power-law que describe la frecuencia del resto de los flujos. Usando esta aproximación y las frecuencias reales de los top-K, se responden dos tipos de consultas de cuantiles: (i) Dado un valor de cuantil, determinar la frecuencia que se encuentra en ese cuantil. (ii) Dada una frecuencia, encontrar el cuantil en el que ésta se encuentra. Adicionalmente, presentamos la arquitectura de un acelerador hardware que implementa este algoritmo sobre un dispositivo FPGA.

El algoritmo propuesto responde consultas sobre 9 trazas de CAIDA [2] con un error absoluto promedio entre 0.0004 % y 0.0122 % para consultas de cuantil dado un valor de frecuencia y un error absoluto promedio entre 0.43 y 2.09 cuentas para consultas de frecuencia dado un valor de cuantil. La arquitectura propuesta fue implementada en un FPGA AMD XCU280 UltraScale+, alcanzando una frecuencia máxima de 392MHz en el procesamiento de paquetes. El acelerador puede recibir un nuevo paquete en cada ciclo de reloj, soportando una tasa de recepción de peor caso de 200Gbps. Por otro lado, la latencia máxima de consulta es de 38.5 μs para consultas de frecuencia dado un valor de cuantil y 4.2 μs para consultas de cuantil dado un valor de frecuencia.

“There are no big breaks. There are only a series of tiny, little breaks. The key is to work your hardest and do your best at every little break.”

- Tara Schuster

Durante las diferentes etapas de este trabajo he recibido el apoyo financiero del Programa de Becas para Estudios de Magíster Nacional de la Agencia Nacional de Investigación y Desarrollo de Chile (ANID) del Gobierno de Chile.

Índice general

Índice de figuras

Índice de tablas

Índice de algoritmos

Capítulo 1. Introducción

La medición y análisis de tráfico en redes de datos de alta velocidad es una tarea importante, ya que nos permite monitorear el estado de la red. Esta información nos ayuda a mejorar la gestión de la red además de detectar posibles fallas o amenazas a su seguridad [3]. El tráfico de red puede ser modelado por conexiones entre dos puntos de la red, llamadas flujos, los cuales están compuestos a su vez por paquetes que comparten características comunes. Más específicamente, los paquetes de un mismo flujo comparten hasta cinco propiedades características: dirección IP de origen, dirección IP de destino, puerto de origen, puerto de destino y protocolo de comunicación. Debido a que un flujo contiene la información de varios paquetes, el análisis de estos flujos nos entrega información más relevante y completa que el análisis de los paquetes individuales. Durante este análisis de los flujos de red, se busca obtener métricas que describan el tráfico medido. Una de las métricas que nos ayudan a describir el comportamiento del tráfico es la distribución de algunas de sus propiedades. Una forma de caracterizar esta distribución es el cálculo de cuantiles. El cuantil q en el que se encuentra un elemento x se define como la fracción de elementos de un conjunto de n elementos $\{x_1, \ldots, x_n\}$ tales que $x_i \leq x$. Inversamente, dado un cuantil q, el elemento que se encuentra en este cuantil corresponde al menor valor x tal que la fracción de elementos menores o iguales a x sea mayor o igual a q. Adicionalmente, una métrica relacionada a los cuantiles es el rank de x, definido como la cantidad de elementos que tienen un valor menor o igual a x. El valor de estos cuantiles, entre los cuales se encuentran la mediana y el percentil 99, nos ayuda a entender la estructura del tráfico y detectar tanto anomalías en el corto plazo como cambios en el largo plazo en la distribución [4].

En particular, buscamos estimar cuantiles de frecuencia para los flujos de la red. La frecuencia de un flujo es el número de paquetes pertenecientes a un flujo en un intervalo de observación. Este intervalo puede variar según la aplicación. En el caso específico de este trabajo, utilizamos ventanas de observación de 60 segundos, que es la duración de las trazas de prueba utilizadas.

Sin embargo, la medición y análisis del tráfico de red requiere procesar en línea una secuencia de paquetes a alta velocidad y producir resultados con baja latencia para poder tomar acciones correctivas de ser necesario [5]. Debido a los grandes volúmenes de tráfico y la velocidad de los enlaces modernos, esto se vuelve cada vez más costoso computacionalmente [6], por lo que un procesador de propósito general se vuelve insuficiente para abordar el problema. Los switches programables de alto desempeño nos permiten abordar algunas de estas tareas, pero los modelos de programación que éstos utilizan limitan los algoritmos que se pueden ejecutar de manera eficiente [7]. Por lo tanto, los aceleradores hardware de propósito específico se vuelven una opción atractiva debido a que se diseñan para realizar una tarea específica de la manera más eficiente posible, por lo que podemos alcanzar altas velocidades de procesamiento.

Una clase importante de aceleradores hardware son aquellos basados en dispositivos FPGA (Field-Programmable Gate Array). Estos dispositivos se caracterizan por su flexibilidad y alta capacidad máxima de cómputo, debido a que poseen una gran cantidad de lógica programable. Además, su paralelismo de grano fino hace que sea posible ejecutar algoritmos de manera más rápida y eficiente que en un procesador de propósito general [8]. 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 [9, 10]. Esto se debe a que el ancho de banda a memoria externa es considerablemente menor al ancho de banda agregado logrado al utilizar los múltiples bloques de memoria interna, los cuales son un recurso escaso. Esto implica 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 ampliamente el espacio disponible en estos dispositivos. Particularmente, en el caso de los cuantiles, es imposible encontrar una solución exacta en una pasada sin almacenar todos los datos [11].

Afortunadamente, para muchas aplicaciones es suficiente trabajar con estimaciones, sin incurrir en el alto costo de calcular los valores exactos [3]. Se vuelve necesario entonces utilizar métodos que nos permitan estimar las propiedades deseadas, reduciendo la cantidad de memoria necesaria sin perjudicar mayormente la 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. Sin embargo, el método es sensible a elementos con valores atípicos y al sesgo en los datos, necesitando una tasa de muestreo muy alta para obtener una buena precisión, lo que consume una cantidad importante de recursos [3, 12]. Alternativamente, 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 permiten obtener una estimación precisa con un bajo costo en memoria [3, 8]. Los algoritmos basados en sketches usan todos los datos de entrada, lo que los hace sensibles a valores atípicos. Además, estos algoritmos son especialmente apropiados para datos obtenidos por streaming, es decir, conjuntos grandes de datos en los que solo podemos acceder a ellos una vez. Estos algoritmos exhiben, además, un alto grado de paralelismo intrínseco.

Aunque actualmente existen diferentes algoritmos basados en sketches que permiten la estimación de cuantiles, la frecuencia de un flujo es una propiedad que se construye a través de múltiples paquetes recibidos en línea ya que la frecuencia de un flujo se incrementa cada vez que llega un nuevo paquete perteneciente a ese flujo. Si queremos trabajar en línea, todo incremento debe verse reflejado en el sketch. Esto equivale a insertar la nueva frecuencia del flujo y eliminar la anterior del sketch cada vez que llega un nuevo paquete, lo que implica un número de eliminaciones prácticamente igual al número de inserciones. Una forma de abordar esto es trabajar con sketches que soporten un número arbitrario de eliminaciones. Sin embargo, el espacio utilizado por los sketches que soportan este modelo depende generalmente del tamaño del universo de elementos insertados, salvo en algunos algoritmos que mantienen un tamaño fijo a costa de una disminución en la precisión de la estimación. Este universo puede ser considerablemente grande según la aplicación, resultando ineficiente en términos de recursos utilizados. Por otro lado, estos sketches tienen una complejidad de actualización mayor que aquellos que trabajan solo con inserciones [13]. Por ejemplo, estos requieren calcular logaritmos del elemento entrante en su proceso de inserción, lo que es complejo de implementar en aceleradores hardware.

Debido a lo mencionado anteriormente, se vuelve interesante explorar la posibilidad de combinar un sketch de estimación de cuantiles con otras estructuras de datos probabilísticas, tales como sketches de estimación de frecuencia. Estas estructuras nos permiten obtener el valor de propiedades acumuladas de los flujos, tales como el número de paquetes procesando cada paquete según el flujo al que pertenece. Sin embargo, utilizar estos sketches para estimar la frecuencia de todos los flujos requiere almacenar ya sea el valor estimado por cada flujo, o el identificador de cada flujo, lo que supera la memoria interna disponible. Por otro lado, los sketches de frecuencia suelen sobreestimar la frecuencia de los flujos más pequeños debido a colisiones en la función hash utilizada para indexar el sketch [14], entregando peores estimaciones para estos flujos. No obstante, para el caso de la entropía, otra métrica de análisis de tráfico de redes, es posible estimar solo los flujos con mayor frecuencia (los top-K) y utilizar una parte de ellos para aproximar la frecuencia del resto de los flujos asumiendo una distribución estadística [15, 1] sin perjudicar mayormente la estimación. Las frecuencias de estos flujos top-K no son fácilmente aproximables mediante una distribución estadística; sin embargo, en conjunto, permiten estimar la distribución de frecuencias del resto de los flujos.

Considerando lo anterior, en este trabajo presentamos un algoritmo para la estimación de cuantiles de frecuencia de flujos de red. Para esto, utilizamos una versión modificada del TowerSketch [16] junto a un arreglo de colas de prioridades (PQA) [1] para estimar la frecuencia de los flujos más grandes. La frecuencia

del resto de los flujos se aproxima mediante una distribución estadística power-law, cuyos parámetros son calculados usando estos flujos más frecuentes. Una vez determinados estos parámetros, estimamos los cuantiles mediante una fórmula matemática obtenida de la distribución estadística. Además, usamos un sketch de cardinalidad HyperLogLog [17] para estimar el número de flujos en la traza de red.

Adicionalmente, presentamos la arquitectura de un acelerador hardware para este algoritmo y lo implementamos en una tarjeta AMD Alveo U280, la cual contiene un FPGA XCU280 con arquitectura UltraScale+.

Evaluamos nuestro algoritmo usando 9 trazas de CAIDA [2]. El algoritmo responde consultas de cuantiles dado un valor de frecuencia con un error absoluto promedio 0.0004 % y 0.0122 % y consultas de frecuencia dado un valor de cuantil con un error absoluto promedio entre 0.43 y 2.09 cuentas. Al implementarlo en una FPGA AMD XCU280 UltraScale+, el procesamiento de paquetes alcanza una frecuencia máxima de 392MHz y procesa un paquete por ciclo, por lo que soporta una tasa de recepción de peor caso de 200Gbps. El procesador, después de haber recibido los top-K y calculado los parámetros de la distribución, puede responder consultas de cuantiles con una latencia máxima de 38.5 μs para consultas de frecuencia dado un valor de cuantil y 4.2 μs para consultas de cuantil dado un valor de frecuencia.

En el Capítulo 2 realizamos un análisis bibliográfico del estado del arte, abarcando las distintas aristas mencionadas hasta el momento. Basándonos en este análisis, generamos nuestra hipótesis, de la cual se desprenden el objetivo general y los objetivos específicos presentados en el Capítulo 3. A continuación, en el Capítulo 4, describimos los principales algoritmos que componen nuestro algoritmo principal (Método de estimación de la distribución, estimación de la frecuencia de los top-K, ordenamiento, estimación de cardinalidad y algoritmos de consulta). Posteriormente, describimos la arquitectura del acelerador hardware del algoritmo, describiendo cada componente en detalle en el Capítulo 5. Finalmente, en el Capítulo 6, presentamos los resultados obtenidos por el algoritmo en cuanto a estimación de frecuencia de los top-K, estimación de cardinalidad, estimación de frecuencia y precisión en la estimación de cuantiles. Además, evaluamos el acelerador en cuanto a uso de recursos, latencia y frecuencia de operación.

Capítulo 2. Análisis del estado del arte

El problema de la aceleración de la estimación de cuantiles para propiedades de flujos en tráfico de red posee dos dimensiones. Por un lado, debemos abordar la dificultad de la capacidad de cómputo necesaria para procesar paquetes a gran velocidad y obtener resultados en línea, lo cual puede ser abordado utilizando aceleradores hardware. Por otro lado, debemos utilizar algoritmos que nos permitan estimar las propiedades buscadas teniendo en cuenta las restricciones de memoria y aprovechando el paralelismo de grano fino de los dispositivos FPGA. Por lo tanto, organizamos el análisis bibliográfico en dos secciones. En primer lugar, exploramos los principales algoritmos basados en sketches para la estimación de cuantiles y de frecuencia. Luego, analizamos distintos trabajos que han combinado estos algoritmos basados en sketches con la aceleración hardware.

2.1. Algoritmos basados en sketches

2.1.1. Sketches de estimación de cuantiles

Dado que encontrar el valor exacto de un cuantil requiere espacio lineal, el problema de la estimación de cuantiles ha generado interés en los últimos años y se han publicado numerosos trabajos que intentan resolver este problema, mejorando la precisión de la estimación o reduciendo el costo en memoria de la solución. Existen dos tipos de algoritmos basados en sketch de estimación de cuantiles. Los algoritmos determinísticos entregan aproximaciones con una cota de error igual a ϵ, la cual se cumple en todos los casos. En cambio, los algoritmos aleatorizados entregan una probabilidad de fallo δ, es decir, cumplen la cota de error ϵ con una probabilidad de $(1 - \delta)$.

En la publicación pionera sobre este tema, Munro y Paterson [11] 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 $\Omega(n^{1/2p})$.

Manku et al. [18] proponen un algoritmo determinístico para la estimación de cuantiles en flujos de datos, conocido como MRL98, basado en el trabajo de Munro y Paterson [11]. El algoritmo utiliza b buffers de tamaño k. Cada uno de estos buffers tiene asociado un nivel y un peso. Los elementos del flujo se insertan en buffers vacíos y, cuando no hay más disponibles, se realiza un proceso de colapso entre los buffers de menor nivel, creando un nuevo buffer de salida con un nivel más, cuyo peso es igual a la suma del peso de los buffers colapsados. Este colapso consiste en replicar los elementos según su peso, ordenarlos y seleccionar k elementos equi-espaciados para formar un nuevo buffer de nivel superior. La estimación del cuantil Φ se obtiene ordenando todos los elementos presentes en los buffers y seleccionando el elemento correspondiente al índice deseado. El algoritmo garantiza una estimación ε-aproximada con probabilidad de error menor a δ. El algoritmo requiere $b = O(\log(\varepsilon n))$ buffers y cada uno tiene tamaño $k = O\left(\frac{1}{\varepsilon}\log(\varepsilon n)\right)$, por lo que la complejidad espacial del algoritmo es de $O\left(\frac{1}{\varepsilon}\log^2(\varepsilon n)\right)$. Además, los autores proponen una variante con muestreo que reduce el espacio a $O\left(\frac{1}{\varepsilon}\log^2\left(\frac{1}{\varepsilon}\log\frac{1}{\delta}\right)\right)$.

Posteriormente, Manku et al. publicaron una mejora al algoritmo, MRL99 [19] que no requiere conocimiento previo del número de elementos del stream. Este algoritmo combina el MRL original con una técnica de muestreo no uniforme, es decir, no todos los elementos tienen la misma probabilidad de ser seleccionados.

Además de los parámetros b y k descritos anteriormente, esta versión del algoritmo recibe un parámetro h. Una vez que el nivel máximo de los buffers llega a h, la tasa de muestreo pasa a ser de $r = \frac{1}{2}$ y los buffers llenados con elementos nuevos del stream tienen ahora nivel 1. A continuación, cada vez que se agrega un nuevo nivel de buffers, la tasa de muestreo disminuye a la mitad y el nivel de los buffers llenados con elementos del stream aumenta en 1. 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 [20] presentan GK Sketch, un algoritmo determinístico con complejidad espacial igual a $O(\frac{1}{\varepsilon}\log \varepsilon n)$. Este algoritmo permite tener cotas superiores e inferiores para cada cuantil en vez de una sola para todos los cuantiles. La estructura de resumen almacena tuplas de 3 valores: v_i, un valor visto, $g_i = r_{min}(v_i) - r_{min}(v_i - 1)$ y $\Delta i = r_{max}(v_i) - r_{min}(v_i)$, donde r_{min} y r_{max} son, respectivamente, la cota inferior y superior del rank de v_i en los elementos vistos hasta ahora. Cuando el resumen crece demasiado, el algoritmo comprime la estructura fusionando tuplas adyacentes, siempre y cuando dicha operación no viole la garantía de error. Esta estructura puede estimar cuantiles con un error máximo de $\frac{max_i(g_i+\Delta i)}{2}$. Para estimar el elemento correspondiente a un cuantil Φ, se calcula el rank $r = \lceil \Phi n \rceil$ y se busca v_i tal que $r(v_i) - r_{min}(v_i) \leq \varepsilon n$ y $r_{max}(v_i) - r(v_i) \leq \varepsilon n$. Este sketch no permite una operación de unión completa entre estructuras (*merge*).

Felber y Ostrovsky [21] proponen combinar la estructura del GK Sketch con muestreo 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 es un sketch GK y contiene $2^r \cdot 32m$ elementos del stream. Además, cada fila $r \geq 1$ tiene un muestreador (*sampler*) de tipo Bernouilli. Una vez llegados $\frac{1}{64}$ de los elementos que representará una fila, ésta se activa y construye su resumen inicial a partir de los datos almacenados en la fila inmediatamente inferior. Luego de esta activación, la fila puede ser descartada. Este sketch disminuye la complejidad espacial a $O(\frac{1}{\varepsilon}\log\frac{1}{\varepsilon})$. Sin embargo, tampoco permite la unión entre estructuras.

Agarwal et al. [22] proponen un tipo de resumen compuesto por $\log \frac{n}{k_\varepsilon}$ capas, donde cada capa contiene un resumen con $k_\varepsilon = O(\frac{1}{\varepsilon}\sqrt{\log\frac{1}{\varepsilon\delta}})$ elementos. Cada vez que una capa se llena, se escogen los elementos con índice par o impar de manera aleatoria y se insertan en la capa inmediatamente superior con el doble de peso. La introducción de esta aleatoriedad es la principal mejora al trabajo de Manku et al. [19] 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 $O(\frac{1}{\varepsilon}\log^{3/2}\frac{1}{\varepsilon})$, pero soporta operaciones de unión entre estructuras.

Karnin et al. [23] proponen el sketch KLL. Este sketch toma el trabajo propuesto en [19] y [22] 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 ancho máximo de un compactor está dado por $k_H = O(\frac{1}{\varepsilon}\log\log(\frac{1}{\varepsilon\delta}))$. Cuando se llena un compactor, se ordenan los elementos y se escogen aleatoriamente aquellos con posiciones pares o impares, descartando el resto de los elementos. El peso de los elementos escogidos es multiplicado por 2. Adicionalmente, el algoritmo propuesto usa distintas capacidades para compactors en distintas alturas, disminuyendo el tamaño exponencialmente en una constante c a medida que la altura disminuye tal que $k_h = k_H c^{H-h-1}$. La capacidad mínima de un compactor es de 2 elementos, por lo que en la parte inferior del sketch hay H'' compactors con tamaño 2. Para disminuir el espacio utilizado, se pueden reemplazar estos compactores por un muestreador que seleccione uno de cada $2^{H''}$ elementos. Adicionalmente, el sketch KLL propone que los $\log\log(\frac{1}{\delta\varepsilon})$ compactors superiores mantengan el tamaño máximo debido a que éstos 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 reemplazando 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.

Zhao et al. [13] proponen una modificación al KLL, el KLL±. Esta nueva versión funciona bajo el modelo de eliminaciones acotadas (como máximo $I(1-\frac{1}{\alpha})$ eliminaciones, donde I es el número de inserciones). Se le agrega un bit de signo a los elementos para representar si es una inserción o una eliminación. Adicionalmente, se modifica el algoritmo de compactación para tener en cuenta los pares inserción/eliminación y los signos de los elementos. Debido a esto, el tamaño mínimo de un compactor pasa a ser 3. La complejidad espacial de este sketch es de $O(\frac{\alpha^{1,5}}{\varepsilon}log^2log(\frac{1}{\varepsilon\delta}))$ y obtiene una precisión similar a la del KLL original. Sin embargo, aunque este sketch soporta eliminaciones, no es aplicable a nuestro problema de calcular cuantiles de propiedades acumulativas, debido a que al haber actualizaciones, α tendería a infinito al igual que el espacio utilizado.

Por otro lado, Ivkin et al. [24] proponen KLL Sweep, una modificación del 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. Cuando el número de elementos supere el espacio total disponible en el sketch, se compacta el compactor de menor nivel cuyo número de elementos supere el espacio asignado a éste. 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. Otra modificación propuesta es seleccionar de manera alternada los elementos pares o impares.

Existen otros trabajos recientes que usan enfoques distintos para la estimación de cuantiles. Por ejemplo, el Moments Sketch propuesto por Gan, et al [25] almacena el valor mínimo, el valor máximo, k momentos μ y k momentos logarítmicos ν del dataset, donde k es el orden del sketch y corresponde a un número pequeño, usualmente entre 10 y 22. Con esta información, el sketch selecciona la distribución $p(x)$ que se corresponda con los datos utilizando el método de momentos [26]. 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, escogiendo la distribución menos informativa que sea consistente con los datos observados. Posteriormente, se utiliza la distribución seleccionada para estimar cualquier cuantil.

DDSketch (Distributed Distribution Sketch) es un algoritmo determinístico basado en histogramas propuesto por Masson, et al. [27] 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 $\gamma=\frac{1+\alpha}{1-\alpha}$ y α es el error relativo máximo. Cada elemento x es mapeado al bucket $\lceil\log_\gamma(x)\rceil$. A medida que se van agregando elementos, el algoritmo agrega buckets al histograma hasta exceder el número máximo. Cuando esto ocurre, se colapsan los dos primeros buckets del histograma, dejando espacio para un bucket adicional. Debido a esto, el histograma puede manejar un amplio rango de valores en un número reducido de buckets. Este sketch soporta eliminaciones y un tamaño que no depende del universo. Sin embargo, al colapsar los primeros buckets del histograma, se pierde precisión en los primeros cuantiles.

Epicopo, et al. [28] proponen el Uniform DDSketch o UDDSketch, una modificación del DDSketch descrito anteriormente. Su principal diferencia radica en cómo se realiza la operación de extensión de rango. En vez de colapsar solo los primeros dos buckets, se colapsan 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 Relative Error Quantile Sketch o ReqSketch propuesto por Cormode et al. [29] es similar al KLL en que almacena una muestra de los datos observados en compactors relativos, los cuales tienen un tamaño $B=2k\lceil\log\frac{n}{k}\rceil$ donde k es un entero par. A diferencia de KLL, al realizar la operación de compactación en un nivel h, solo se consideran los L elementos más grandes almacenados en el buffer, de los cuales se selec-

cionan 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 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 relativo: $\frac{|\hat{R}(x)-R(x)|}{R(x)} \leq \varepsilon$, donde $R(x)$ es el valor real del rank de x y $\hat{R}(x)$ es el valor del rank estimado por el algoritmo.

Gilbert et al. [30] presenta el algoritmo RSS (Random Subset Sum) cuya complejidad espacial y tiempo de actualización es de $O(\frac{1}{\varepsilon^2} log^2 U log \frac{logU}{\varepsilon})$, donde U es el tamaño del universo. Este sketch descompone el universo en $\log U$ capas, donde, en cada capa i, el universo está dividido en $\frac{U}{2^i}$ intervalos de tamaño 2^i. Por ende, el número de intervalos va incrementando en cada capa hasta que la última representa todos los elementos de U. Para estimar el rank de un elemento x, podemos dividir el intervalo $[1, x]$ en varios intervalos y consultar por la frecuencia de estos intervalos en su capa correspondiente, para finalmente sumar todas las estimaciones. Este sketch soporta el modelo turnstile. Sin embargo, la complejidad espacial del sketch crece con el tamaño del universo U.

Cormode et al. [31] proponen el Dyadic Count-Min, que utiliza la misma estructura que el RSS Sketch, pero estima la frecuencia usando un sketch Count-Min en cada capa. Para estimar un cuantil, se realizan $\log U$ consultas a cada sketch para hacer una búsqueda binaria. Este algoritmo mejora el tiempo de actualización a $O(\log U \log \frac{logU}{\varepsilon})$. Luego, Wang et al. proponen usar un Count-Sketch en vez de un Count-Min. A esta modificación se le llama Dyadic Count Sketch o DCS [32].

De esta revisión podemos concluir que el problema de estimar cuantiles con un uso eficiente de recursos ha sido ampliamente estudiado. La mayoría de los algoritmos propuestos no soportan un número de eliminaciones arbitrarias, volviéndose inadecuados para estimar los cuantiles de frecuencia en flujos. Por otro lado, los sketches que soportan eliminaciones arbitrarias pueden ser útiles para dar solución al problema que queremos resolver. Sin embargo, su complejidad espacial y su tiempo de actualización dependen del tamaño del universo de elementos o bien sacrifican precisión al intentar mantener una estructura de tamaño fijo. Debido a estas limitaciones, se vuelve necesario explorar soluciones alternativas para la estimación de la distribución de frecuencia en flujos de red.

2.1.2. Sketches de estimación de frecuencia

La estimación de frecuencia de los elementos de un stream sigue siendo un área de investigación activa. Charikar et al. [33] presentan un algoritmo llamado Count Sketch. Este sketch consiste en un arreglo de contadores de tamaño $t \times b$. Cuando se inserta un elemento, se computa el resultado de t funciones hash las cuales determinan cuál es el bucket que deberá ser modificado en cada fila. Luego, utilizando otra función hash, se incrementa o decrementa aleatoriamente el valor contenido en el bucket. Para obtener la estimación de frecuencia de un elemento, se calcula la mediana entre los buckets. Si consideramos $b = \frac{2}{\varepsilon^2}$ y $t = log(\frac{1}{\delta})$, donde δ es la probabilidad de que el error sea mayor a ε, la complejidad espacial del algoritmo es $O(\frac{1}{\varepsilon^2} \frac{1}{\delta} \log(m))$, donde m es el número de elementos insertados. Este algoritmo puede sobreestimar o subestimar la frecuencia de un elemento.

Cormode et al. [34] proponen un algoritmo similar llamado Count-Min. El algoritmo se diferencia en que al actualizar los contadores, éstos se incrementan en una constante c_i. La frecuencia estimada por el sketch es el valor mínimo almacenado en los contadores correspondientes a un elemento. La complejidad espacial

es de $O(\frac{1}{\varepsilon}\log\frac{1}{\delta}\log(m))$. Este algoritmo nunca subestima la frecuencia estimada. Existe una mejora a este algoritmo, basada en la idea de Conservative Update [35]. Este algoritmo puede obtener mejores estimaciones incrementando solo aquellos contadores que tengan el valor mínimo entre los buckets seleccionados, ya que éstos serán los que tienen una menor cantidad de colisiones. Esta mejora es llamada Count-Min CU.

Yang et al. [36] desarrollan un framework llamado Pyramid Sketch que mejora 3.5 veces la precisión así como también el tiempo de inserción y estimación de los sketches anteriores. Este framework se basa en una estructura en forma de pirámide, en donde la primera fila contiene los contadores del sketch. El tamaño de cada fila va disminuyendo a la mitad con la altura hasta la última fila, la cual tiene tamaño uno. A partir de la segunda fila, cada contador está asociado a dos contadores de la fila inferior, uno a la izquierda y uno a la derecha. Cada uno de estos contadores contiene 3 partes: una bandera que indica si el contador izquierdo de la fila inferior se rebalsó, un contador y una bandera que indica si el contador derecho de la fila inferior se rebalsó. Si al incrementar un contador, éste se rebalsa, se activa la bandera correspondiente y se incrementa el contador asociado de la fila superior. Si este contador se rebalsa a su vez, se repite el proceso hasta que se llegue a una fila donde no haya rebalse. Para estimar la frecuencia de un elemento, se van sumando las estimaciones capa por capa hasta que la bandera de rebalse esté desactivada.

Roy et al. [37] proponen el Augmented Sketch, el cual combina los sketches mencionados anteriormente con una etapa de pre-filtrado, mejorando la estimación de los flujos más frecuentes. El filtro almacena las frecuencias de los L flujos más frecuentes en una tabla hash, donde L es un número pequeño. Cuando llega un nuevo elemento, se revisa si este elemento está contenido en el filtro. De ser así, se actualiza la frecuencia almacenada en el filtro para ese flujo. De lo contrario, si el filtro no está lleno, se agrega este flujo al filtro con una frecuencia de 1. En caso de que el filtro esté lleno y no contenga el elemento buscado, se actualiza la frecuencia de este flujo en el sketch de estimación de frecuencia. Luego, si la frecuencia del flujo entrante es mayor a la frecuencia mínima del filtro, se reemplaza el flujo con esta frecuencia mínima en el filtro por el flujo entrante y se actualiza el flujo reemplazado en el sketch.

El Elastic Sketch propuesto por Yang et al.[14] propone combinar un sketch Count-Min con una tabla Hash. La tabla Hash permite almacenar frecuencias exactas para los flujos más frecuentes o "elephant flows". Se utiliza un sistema de votos positivos y negativos para determinar si un flujo debe permanecer en la tabla o insertarse en el Count-Min. De esta manera se pueden obtener estimaciones más precisas y disminuir las colisiones en el Count-Min.

Recientemente, Liu et al. propusieron el Funnel Sketch [38]. La propuesta se basa en la idea de separar las estimaciones de flujos menos frecuentes y los flujos más frecuentes. La arquitectura del sketch se divide en 3 partes con contadores con un número de bits distinto. La parte superior es un Count-Min CU para estimar el tamaño de los flujos ratones. La parte central contiene una matriz de buckets, donde cada bucket contiene una estructura en forma de embudo. Esta estructura está compuesta por dos arreglos de contadores y un par de elementos que almacena un identificador de flujo y la frecuencia de un candidato a flujo elefante. Finalmente, la parte inferior es un arreglo de contadores más pequeño. Al insertar un elemento, se intenta insertar primero en la parte superior. Si el contador a incrementar no está rebalsado, éste se incrementa en uno. De lo contrario, se inserta en la parte central, en el primer arreglo de contadores. Si ese contador se rebalsa, se marca el flujo como un posible candidato a flujo elefante. Si no queda espacio para almacenar más candidatos, se inserta el paquete en el segundo arreglo de contadores del bucket. Finalmente, si el contador más pequeño entre la estimación de frecuencia almacenada y el contador del segundo arreglo está rebalsado, se inserta el paquete en la parte inferior.

Yang et al. [16] proponen TowerSketch, un sketch de estimación de frecuencia basado en la estructura del Count-Min, pero donde cada fila ocupa la misma cantidad de memoria, pero tiene contadores de distinto

tamaño. De esta manera, existen menos contadores para almacenar las frecuencias grandes y una cantidad mayor de contadores para almacenar los flujos más pequeños, lo que es consistente con el comportamiento sesgado del tráfico, donde la mayoría de los flujos son pequeños y solo unos pocos alcanzan altas frecuencias. La estimación entregada por el sketch es el valor mínimo de los contadores asociados al flujo que no estén rebalsados. De esta manera, se disminuye el efecto de las colisiones en la función hash y se reduce la sobreestimación de la frecuencia. Además, los autores utilizan la idea de Conservative Update para mejorar la precisión del algoritmo.

Podemos observar que además de los sketches clásicos como el Count-MinCU y el Count Sketch, se han propuesto nuevos algoritmos que buscan tomar en cuenta la distribución sesgada del tráfico para aprovechar el espacio de manera más eficiente. Por esto, se vuelve interesante explorar estas propuestas recientes para mejorar la estimación de frecuencia en los flujos top-K. La estimación de frecuencia de flujos usando los 5 campos descritos anteriormente como identificador es más desafiante que la estimación de frecuencia de flujos usando como identificador solamente, por ejemplo, las direcciones IP de origen y destino. Esto se debe a que, al aumentar el número de campos considerados en el identificador, la cardinalidad aumenta sustancialmente, lo que puede aumentar las colisiones en el sketch. Considerando esto, resulta relevante para nuestro trabajo evaluar como se comportan los sketches de estimación de frecuencia bajo estas circunstancias.

2.1.3. Algoritmos para detección y estimación de la frecuencia de flujos top-K

Como mencionamos anteriormente, el enfoque de este trabajo consiste en utilizar la frecuencia de los K flujos más frecuentes (top-K) para estimar la distribución de la frecuencia del resto del tráfico. Por lo tanto, es fundamental detectar y poder estimar la frecuencia de estos flujos con la mayor precisión posible. Varios algoritmos han sido propuestos en la literatura para abordar esta problemática.

HeavyKeeper [39] es un algoritmo propuesto por Yang et al. en 2019. Utiliza un sketch compuesto por d filas de w contadores, donde cada contador almacena un identificador y un contador. Cada flujo se mapea a d buckets distintos mediante funciones hash independientes y se modifica el contenido de estos buckets según el caso. Si el bucket está vacío, se inserta el flujo entrante en el bucket. Si el bucket contiene el mismo flujo entrante, se incrementa su contador en 1. Finalmente, si el flujo almacenado es distinto, se decrementa el contador con una probabilidad que sigue un decaimiento exponencial. Si el contador llega a 0, el flujo almacenado se reemplaza por el flujo entrante. Para estimar la frecuencia de un flujo, HeavyKeeper selecciona el contador con mayor valor entre los buckets que actualmente almacenan este flujo en su identificador. El sketch estima la frecuencia del flujo cada vez que se inserta un paquete nuevo. La detección de los top-K flujos se hace usando un min-heap para almacenar los flujos más frecuentes. Con 100 KB de memoria, obtienen una precisión mayor al 94 % en la detección de los top-1000 flujos en trazas de CAIDA, utilizando direcciones IP de origen y destino como identificador. Además, obtiene un error relativo promedio (ARE) cercano a 0.01 y un error absoluto promedio (AAE) de aproximadamente 60 en la estimación de frecuencia.

Cuckoo Counter [40] es una estructura compuesta por dos arreglos de w buckets cada uno. Cada uno de estos buckets contiene B entradas, que almacenan un identificador de flujo y un contador. Cada flujo se mapea a un bucket en cada arreglo, los cuales son escaneados en busca del flujo entrante. Si se encuentra una entrada vacía o si el flujo ya está presente, se actualiza su contador. De lo contrario, se inicia un proceso de expulsión iterativa, en el que uno de los elementos existentes es desalojado para hacer espacio al nuevo. Este proceso continúa hasta que tanto el nuevo flujo como los elementos expulsados sean insertados satisfactoriamente, o hasta que se alcance un número máximo de intentos. Además, se utiliza un heap para almacenar los flujos más frecuentes y se verifica si hay colisiones en la función hash. En ese caso, se compara la diferencia en

frecuencia con un umbral para decidir si se debe reportar el flujo como un top-K. El algoritmo logra un 95 % de precisión en la detección de los 1000 flujos más frecuentes en trazas de CAIDA, usando 100 KB de memoria.

En [41, 15, 1], Soto y Fernández et al. usan un sketch de frecuencia (CountMin-CU o Count Sketch) junto a un arreglo de colas de prioridades para estimar la frecuencia de los K flujos más frecuentes. Cada vez que llega un paquete, éste se inserta en el sketch de frecuencia y la estimación entregada por éste se inserta en el arreglo de cola de prioridades. Durante esta inserción, se escoge una de las colas mediante una función hash. En cada una de estas colas se mantienen los elementos más frecuentes insertados en ella, de manera que para extraer los top-K estimados solo debemos leer el contenido de la estructura. Aunque la precisión de la estimación de la frecuencia y detección de top-K no fue evaluada directamente, se obtiene un error menor al 2.42 % al usar una estimación estadística calculada con los top-K para la estimación de la entropía de los flujos.

Yang et al. [42] proponen un algoritmo llamado CTS Sketch para identificar los flujos top-K. El algoritmo está pensado para ser implementado en entornos de red que soporten Software-Defined Networking (SDN) y usen un controlador centralizado. En cada switch, se mantiene una tabla hash local que identifica los candidatos a top-K mediante un sistema de votación. De forma periódica, los switches envían estos candidatos a un controlador central que decide si el flujo es realmente parte de los top-K usando el tiempo de vida del flujo. Usando esta técnica, se obtiene una precisión de más del 94 % en trazas de CAIDA [2] con $K = 1000$. Además, cuando el espacio utilizado es mayor a 20 KB, los autores reportan un error relativo promedio en la estimación de frecuencia menor al 0.001.

Cao et al. [43] proponen Bubble Sketch, un sketch para estimación de flujos top-K que está compuesto por dos arreglos de w buckets. Cada bucket contiene B pares <ID, contador>. Cada flujo es mapeado a un bucket en cada arreglo. Si el flujo es encontrado en alguno de los buckets, se incrementa su frecuencia en 1 y se ordenan los elementos del bucket. Si al terminar de ordenar, el elemento insertado es el segundo más grande, su frecuencia supera un umbral Δ y supera la mayor frecuencia del bucket correspondiente en el otro arreglo, se inicia un proceso de reemplazo, donde se inserta este elemento en el bucket alternativo. Obtienen una precisión de 100 % en trazas de CAIDA para $500 \leq K \leq 5000$ con 250 KB de memoria. Sin embargo, la reubicación basada en umbral requiere lógica adicional, lo que puede complicar su implementación en dispositivos hardware.

Concluimos que actualmente existen en la literatura diversas propuestas para la estimación y detección de los flujos más frecuentes. La solución utilizada por Fernández et al. es conveniente debido a que permite estimar la frecuencia para valores grandes de K, sin implementar estructuras costosas computacionalmente como una cola de prioridades o un min-heap.

2.2. Aceleradores hardware

La aceleración hardware consiste en utilizar distintas plataformas basadas en hardware, tales como dispositivos FPGA, ASICs y procesadores programables de dominio específico, para ejecutar más rápidamente 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 bajo consumo de energía [44]. Sin embargo, como mencionamos anteriormente, la capacidad de procesamiento de estas plataformas se ve limitada por su reducido ancho de banda a memoria externa, el cual es mucho menor al ancho de banda agregado que se logra al utilizar la memoria interna del chip, la cual permite un acceso rápido y paralelo a los datos. Sin embargo, esta memoria interna es un recurso escaso, por lo que varias investigaciones

buscan aprovechar los algoritmos basados en sketches para diseñar arquitecturas que permitan acelerar la estimación de ciertas propiedades, utilizando exclusiva o principalmente memoria interna. A continuación, presentaremos algunos aceleradores hardware implementados en switches programables y dispositivos FPGA.

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 utilizando el lenguaje de programación P4 (Programming Protocol-Independent Packet Processors), el cual describe el procesamiento de los paquetes en 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 éste. Aprovechando esta posibilidad, se han desarrollado 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 [4, 45], de los cuales solo uno está basado en sketches [4].

Ivkin et al. proponen QPipe, una modificación del sketch KLLSweep [24] implementable en el plano de datos de un switch programable. Mantiene la idea de utilizar una técnica de muestreo, pero usa los paquetes no seleccionados como "trabajadores", para llevar valores y finalizar operaciones necesarias para mantener la estructura de datos. Evaluan la implementación estimando cuantiles para distintas propiedades, tales como las direcciones IP de origen y de destino en trazas de CAIDA, o los nombres de páginas de Wikipedia. La implementación logra una mejora de 91.09 veces en cuanto al error máximo de aproximación que se obtendría al usar sólo una técnica de muestreo bajo las mismas condiciones de memoria.

Wang et al. [45] presentan EasyQuantile, un algoritmo para la estimación del elemento correspondiente a un cuantil p en particular, el cual debe conocerse con anterioridad. La idea se centra en estimar el valor de $\hat{q}$ incrementándolo o decrementándolo en un paso λ_n, dependiendo de si el rank del elemento estimado es mayor o menor al rank buscado. Se proponen dos modos de operación, uno para cuantiles pequeños, en el cual se usa un paso más pequeño, y un modo para cuantiles grandes, en el cual se utiliza un paso más significativo. Cuando llega un nuevo elemento x_n, se verifica si éste es mayor o menor al valor estimado $\hat{q}_n$. El algoritmo mantiene un contador de elementos menores y mayores que $\hat{q^n}$ y modifica el valor estimado incrementándolo o decrementándolo por λ_n según corresponda. Los autores implementan este algoritmo en un switch programable Barefoot Tofino [46]. Si bien este algoritmo no está basado en sketches, es 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. Sin embargo, este algoritmo solo estima el cuantil de un valor definido con anterioridad. El algoritmo estima cuantiles de RTT (Round-Trip Time) en trazas de CAIDA con un error de cuantil cercano a 0.01 y cuantiles de tamaño de flujo con un error cercano a 0.001.

Adicionalmente, también se han propuesto implementaciones en P4 para otros algoritmos basados en sketches con aplicaciones distintas a la estimación de cuantiles. Se han publicado dos trabajos para la estimación de entropía utilizando interpolación tabular en switches programables. El trabajo de Yu-Kuen Lai et al. [47] logra alcanzar una velocidad de procesamiento de 100 Gbps tanto en una tarjeta Xilinx U200 como en un switch programable Tofino. Adicionalmente, Yu-Kuen Lai et al. [48] logra implementar un algoritmo basado en sketches en un ASIC programable en P4, transformando cómputos complejos en tablas pre-computadas en tablas "match-action". Este esquema es luego implementado en un switch Barefoot Tofino 2 [49], pudiendo estimar con precisión la entropía de tráfico de red con una velocidad de 400 Gbps. Soto et al. [50] proponen

un algoritmo para la estimación de la entropía empírica de Shannon, midiendo solamente las frecuencias de los flujos con mayor cantidad de paquetes y aproximando el resto mediante una distribución uniforme. Además, implementan este algoritmo en un switch Behavioral Model Version 2, un switch programado en C++, obteniendo errores de estimación menores al 3.26 %.

Los autores de Elastic Sketch [14] describen cómo implementar su algoritmo en un switch programable con soporte para P4. Por otro lado, los autores de TowerSketch [16] proponen una arquitectura de procesamiento para el TowerSketch usando switches que soporten INT (In-Band Network Telemetry), método que consiste en insertar información predefinida adicional en los paquetes.

2.2.2. Aceleradores hardware en FPGA

Los autores Da Tong y Prassana [8] proponen una arquitectura general para poder acelerar algoritmos basados en sketches en FPGA. Para validar su propuesta, implementaron dos sketches ampliamente utilizados: Count-Min Sketch y K-ary Sketch. Obtuvieron un aumento en el rendimiento de hasta 400 veces comparado con el estado del arte, alcanzando tasas de comunicación mayores a 150 Gbps. Esta arquitectura aprovecha técnicas como pipeline, paralelismo en el cálculo de funciones hash y acceso eficiente a memoria, lo que la hace especialmente adecuada para aplicaciones de red de alta velocidad,

Soto et al. [41] proponen una arquitectura para estimación de entropía utilizando los K elementos más frecuentes. Se utiliza un sketch Count-Min 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 un FPGA Xilinx Zynq UltraScale+ MPSoC ZCU102, los autores logran obtener un flujo de procesamiento superior a 181 Gbps, con un consumo de potencia de 511 mW. En un trabajo posterior [15], 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 tasas de recepción de al menos 204 Gbps. Esta tasa de recepción mínima se alcanza cuando todos los paquetes recibidos tienen el tamaño mínimo (64 bytes). Sin embargo, si los paquetes recibidos tienen mayor tamaño, la tasa de recepción alcanzada será mayor. Fernández et al. [1] proponen 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 promedio 3.5 veces menores que el algoritmo en [15] y más de 15 veces menor que [41].

Otro trabajo propuesto por Sateesan et al. [51] 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, equivalente a una velocidad de comunicación mínima de 212 Gbps.

ElasticSketch [14] posee una implementación en un FPGA Stratix V. Alcanzan una frecuencia de reloj de 162.6 MHz, lo que les permite alcanzar una tasa de comunicación de 83 Gbps. Por otro lado, TowerSketch fue implementado en una tarjeta Xilinx Virtex-7 VC709. La arquitectura propuesta para el TowerSketch está implementada en pipeline, pudiendo recibir un paquete por ciclo de reloj. El diseño soporta una frecuencia de reloj máxima de 365 MHz, lo que equivale a una velocidad de comunicación de 186 Gbps.

Finalmente, en 2024, presentamos una arquitectura para un acelerador hardware para estimación de cuan-

tiles de tamaño de paquetes de red [52]. Este acelerador implementa una versión modificada del KLL [23], la cual permite usar un número fijo de compactors. El acelerador fue implementado en una Virtex XCU55 UltraScale+, con una frecuencia de reloj máxima de 356 MHz y procesa un paquete por ciclo. Además, calcula una estimación de cuantil con una latencia inferior a 4.34 µs. Por otro lado, el 99.32 % de las consultas en trazas de CAIDA presentó un error menor al 1 % en estimaciones del cuantil de un tamaño de paquete.

2.3. Discusión

El análisis del estado del arte nos muestra que la estimación de cuantiles sigue siendo un problema que causa interés en la comunidad científica. A pesar de que existen numerosos algoritmos tanto para la estimación de cuantiles como para la estimación de frecuencia, nuevos sketches se siguen proponiendo en la actualidad. El uso de estos algoritmos basados en sketches en aceleradores hardware sigue siendo explorado y desarrollado, obteniendo velocidades de procesamiento cada vez más grandes.

En el estado del arte existe un trabajo relacionado con la aceleración hardware de estimación de cuantiles usando un sketch KLLSweep en un switch programable. Además, en 2024, presentamos un acelerador hardware para estimación de cuantiles en FPGA usando una versión modificada del sketch KLL. Sin embargo, estos sketches trabajan solo con inserciones, lo que no permite estimar los cuantiles de propiedades acumuladas de los flujos, como por ejemplo, la frecuencia de un flujo, sin calcular estos valores antes de insertarlos al sketch. Este problema no es abordado en la literatura actual. Sin embargo, es posible usar otras estructuras de datos probabilísticas, tales como sketches de estimación de frecuencia, para estimar la distribución de propiedades de los flujos más grandes en línea y usar la frecuencia de una parte de estos flujos más grandes para aproximar la frecuencia del resto de los flujos mediante una distribución estadística, pudiendo así estimar el cuantil de una frecuencia en particular. Los algoritmos propuestos para la estimación de top-K alcanzan una precisión alta en la detección y estimación de la frecuencia de estos flujos. Sin embargo, algunos de estos algoritmos utilizan estructuras difíciles de implementar de manera eficiente en un FPGA, tales como colas de prioridad o min-heaps. Adicionalmente, la mayoría trabaja con valores de K relativamente pequeños ($K \leq 5000$). Para estimar correctamente la distribución del resto del tráfico es necesario usar un mayor número de top-K (en el trabajo de Fernández et al. [1] se usan 8192), por lo que es interesante explorar nuevas formas de estimar la frecuencia de los top-K flujos con mayor precisión, mejorando propuestas tales como la presentada por Fernández et al. [1].

Capítulo 3. Hipótesis y objetivos

3.1. Hipótesis

Midiendo solamente la frecuencia de los flujos más grandes y estimando una distribución estadística para el resto, basada en los flujos medidos, es posible diseñar un algoritmo que estime cuantiles de frecuencia de flujos de red con alta precisión y bajo uso de memoria. Este algoritmo puede ser implementado eficientemente sobre un FPGA para medir el tráfico de red con una alta tasa de procesamiento.

3.2. Objetivo general

Diseñar un algoritmo y la arquitectura de un acelerador hardware para la estimación de cuantiles de frecuencia en flujos de red. El algoritmo debe estimar cuantiles dentro de una traza con un error absoluto promedio inferior al 0.5 %. El acelerador debe soportar tasas de transmisión superiores a los 100 Gbps y tener una latencia de respuesta (el tiempo máximo que puede demorarse en responder a una consulta) menor a 70 μs.

3.3. Objetivos específicos

- Diseñar un algoritmo para la estimación de cuantiles de frecuencia de flujos, combinando un sketch de estimación de frecuencia con otras estructuras probabilísticas y con una distribución estadística.
- Diseñar la arquitectura del acelerador hardware adaptando el algoritmo anterior y explotando su paralelismo.
- Implementar y validar el acelerador sobre un dispositivo FPGA.
- Evaluar experimentalmente el desempeño del acelerador al estimar cuantiles de número de paquetes por flujo.

.

Capítulo 4. Algoritmos

A lo largo de este capítulo describiremos los distintos algoritmos que se utilizan para la estimación de cuantiles de frecuencia en tráfico de redes. En primer lugar, definiremos matemáticamente los conceptos de cuantil y rank. Luego, caracterizaremos el método empleado para estimar la distribución del tráfico usando la frecuencia de los K elementos más frecuentes. Posteriormente, presentamos el algoritmo general de cálculo de cuantiles. La entrada al método corresponde al número de flujos (cardinalidad) y la frecuencia de los top-K flujos, por lo que a continuación se presentarán tanto el algoritmo para la estimación de la cardinalidad como el algoritmo para la estimación de dichas frecuencias. Este último se divide en dos partes: la estimación de la frecuencia de los flujos mediante un TowerSketch y la acumulación de los flujos con mayor frecuencia en un arreglo de cola de prioridades denominado "PQA".

4.1. Definiciones relevantes

- El rank R_x de un elemento x en un conjunto de n elementos está dado por:

$$R_x = \Big|\{x_1, \ldots, x_n\} \mid x_i \leq x\Big|, \tag{4.1}$$

  es decir, el número de elementos del conjunto cuyo valor es menor o igual a x.

- El cuantil q_x en el que se encuentra un elemento x en un conjunto de n está dado por:

$$q_x = \frac{R_x}{n} \tag{4.2}$$

- El valor x_q que se encuentra en el cuantil q de un conjunto de n elementos está definido como el menor valor x tal que el rank de x sea mayor o igual a $q_x n$. Matemáticamente, esto equivale a:

$$x_q = \min\{x \in \{x_1, \ldots, x_n\} \mid R_x \geq q \cdot n\} \tag{4.3}$$

4.2. Estimación de la distribución

Para estimar los cuantiles de frecuencia en flujos de red, buscamos estimar la frecuencia de los flujos que no pertenecen a los top-K mediante una distribución estadística. Fernández et al. [1] mostraron que la frecuencia de una parte de los flujos menos frecuentes puede ser aproximada mediante una distribución de tipo power law. Se asume una frecuencia unitaria para el resto de los flujos. La distribución de tipo power law está dada por:

$$\hat{p}_i = Ci^{\alpha}, \tag{4.4}$$

donde $\hat{p}_i$ es la probabilidad estimada de que un paquete pertenezca al flujo i y C y α son constantes de la distribución, con $\alpha < 0$. Considerando que la frecuencia estimada de un flujo i está dada por $\hat{m}_i = M\hat{p}_i$, donde M es el número de paquetes entrantes, podemos expresar la frecuencia estimada de un flujo i de la siguiente manera:

$$\hat{m}_i = MCi^{\alpha} = C_m i^{\alpha}, \tag{4.5}$$

la cual corresponde igualmente a una distribución de tipo power law.

Sabiendo que la distribución power law se comporta como una recta en la escala log-log, Fernández et al. [1] utilizan el método de mínimos cuadrados para calcular la pendiente estimada $\hat{\alpha}$, usando las frecuencias ordenadas de mayor a menor de los K flujos más frecuentes:

$$\hat{\alpha} = \frac{K \sum_{i=1}^{K} (\log \hat{m}_i \log i) - (\sum_{i=1}^{K} \log i)(\sum_{i=1}^{K} \log \hat{m}_i)}{K \sum_{i=1}^{K} (\log i)^2 - (\sum_{i=1}^{K} \log i)^2} \tag{4.6}$$

Sin embargo, observamos que, por lo general, los flujos más frecuentes tienen un comportamiento menos predecible, lo que afecta las estimaciones de la pendiente $\hat{\alpha}$. Teniendo en cuenta esto, proponemos usar la mitad inferior de los top-K, es decir, los $K_b = K/2$ flujos con menor frecuencia dentro de los top-K. De esta manera, la pendiente estimada quedará expresada por:

$$\hat{\alpha_b} = \frac{K_b \sum_{i=K_b+1}^{K} (\log \hat{m}_i \log i) - (\sum_{i=K_b+1}^{K} \log i)(\sum_{i=K_b+1}^{K} \log \hat{m}_i)}{K_b \sum_{i=K_b+1}^{K} (\log i)^2 - (\sum_{i=K_b+1}^{K} \log i)^2} \tag{4.7}$$

Ahora, debemos determinar la constante C_m tal que:

$$\log \hat{m}_i = \hat{\alpha_b} \log i + \log C_m \tag{4.8}$$

Para esto, usaremos la fórmula del intercepto de la aproximación por mínimos cuadrados:

$$\log C_m = \frac{\sum_{i=K_b+1}^{K} \log \hat{m}_i - \hat{\alpha_b} \sum_{i=K_b+1}^{K} \log i}{K_b} \tag{4.9}$$

Luego, calculamos la constante C_m elevando 2 al intercepto calculado, es decir:

$$C_m = 2^{\log C_m} \tag{4.10}$$

Finalmente, determinamos el punto L donde la frecuencia comienza a ser unitaria mediante la ecuación de la recta. Buscamos L tal que:

$$C_m L^{\hat{\alpha_b}} = 1 \Leftrightarrow \log C_m + \hat{\alpha_b} \log L = 0 \Leftrightarrow \log L = \frac{-\log C_m}{\hat{\alpha_b}} \Leftrightarrow L = 2^{\frac{-\log C_m}{\hat{\alpha_b}}} \tag{4.11}$$

4.3. Algoritmo general

El Algoritmo 1 muestra las distintas etapas que involucra el algoritmo. En primer lugar, se inicializan las estructuras TowerSketch, HLL y el PQA según los parámetros de diseño definidos. El algoritmo está dividido en una etapa de procesamiento en línea y una fuera de línea. Por cada paquete, se actualiza en línea el sketch de frecuencia TowerSketch y se inserta la estimación de frecuencia de este sketch en el PQA. En paralelo, se inserta el paquete en el sketch HyperLogLog. Una vez terminado el procesamiento de los paquetes, comienza la etapa fuera de línea del procesamiento. En primer lugar, se estima la cardinalidad de los flujos y se ordenan los

elementos del PQA. Luego se calculan los parámetros de la distribución power-law. Finalmente, se inicia la etapa de recepción de consultas, las cuales se resuelven usando los parámetros calculados anteriormente y las frecuencias estimadas de los top-K.

Algoritmo 1: Algoritmo General

Input: Stream de paquetes de la traza, número de top-K, tamaño de una fila del Tow
precisión p del HyperLogLog

Init:
  `TowerSketch.init(`w`)`
  `HLL.init(`p`)`
  `PQA.init(`K`)`
foreach *paquete con identificador* $flow\_id$ *en el stream* **do**
  `TowerSketch.update(`$flow\_id$`)`
  `// Algoritmo 3`
  `PQA.update(`$flow\_id$`, TowerSketch.estimate(`$flow\_id$`)`
  `// Algoritmos 4 y 5`
  `HLL.update(`$flow\_id$`)`
  `// Algoritmo 8`
$\hat{N} \leftarrow$ `HLL.estimate()`
`SortedPQA` $\leftarrow$ `Sort(PQA)`
`// Algoritmos 6 y 7`
Estimate $\hat{\alpha_b}$
`// Ecuación 4.7`
Estimate C_m
`// Ecuaciones 4.9 y 4.10`
Estimate L
`//Ecuación 4.11`
Responder consultas de cuantiles usando α_b, C_m, L y $\hat{N}$

4.4. MurmurHash3

MurmurHash3 es una función hash no criptográfica que produce resultados de 32 y 128 bits. Esta función es frecuentemente utilizada en aplicaciones de análisis de tráfico de redes que usan sketches debido a su adecuada dispersión de valores y a su reducido tiempo de ejecución [53]. Además, MurmurHash3 se basa en su totalidad en operaciones sencillas de implementar en dispositivos FPGA, tales como multiplicaciones, desplazamientos y operaciones XOR. El Algoritmo 2 detalla cada paso del cálculo del valor hash para una entrada x de *len* bytes de largo, donde *len* debe ser un múltiplo de 4.

4.5. Estimación de los top-K

Para poder calcular los parámetros de la distribución descrita en la sección anterior, debemos estimar en línea el número de paquetes (frecuencia) de los top-K. Como mencionamos anteriormente, cada flujo está identificado por una tupla de 5 valores: direcciones IP de origen y destino, puertos de origen y destino, y protocolo.

Algoritmo 2: MurmurHash3

Input: Elemento de entrada x, número de bytes de elemento de entrada len, semilla $seed$
Output: Hash h

Let
  $c1 = 0xcc9e2d51$
  $c2 = 0x1b873593$
  $r1 = 15$
  $r2 = 13$
  $m = 5$
  $n = 0xe6546b64$
  $h = seed$
for k ***in*** $four\_bytes\_chunk(x)$ **do**
  $k_1 \leftarrow k \times c1$
  $k_2 \leftarrow (k_1 \ll r1) | (k_1 \gg (32 - r1))$
  $k_3 \leftarrow k_2 \times c2$
  $h_1 \leftarrow h \oplus k$
  $h_2 \leftarrow (h_1 \ll r2) | (h_1 \gg (32 - r2))$
  $h_3 \leftarrow h_2 \times m + n$
$h_4 \leftarrow h_3 \oplus len$
$h_5 \leftarrow h_4 \oplus (h_4 \gg 16)$
$h_6 \leftarrow h_5 \times 0x85ebca6b$
$h_7 \leftarrow h_6 \oplus (h_6 \gg 13)$
$h_8 \leftarrow h_7 \times 0xc2b2ae35$
$h_9 \leftarrow h_8 \oplus (h_8 \gg 16)$
return h_9

Durante un intervalo de observación, cada paquete se inserta en un sketch de estimación de frecuencia. Este trabajo usa una modificación del TowerSketch [16] para contabilizar el número de paquetes por flujo. Por cada paquete, se actualiza el sketch y se obtiene una estimación de frecuencia, la cual se inserta en una cola de prioridad aproximada (Priority Queue Array) de tamaño fijo que mantiene los flujos más grandes observados hasta el momento. Al finalizar el intervalo de observación, se ordena el contenido del PQA y se recuperan las K frecuencias más altas. En las siguientes secciones, se describe en detalle cada componente del algoritmo de estimación de top-K.

4.5.1. TowerSketch

TowerSketch [16] es una estructura de datos probabilística para la estimación de frecuencia. Al igual que otros sketches tradicionales, como CountMin y CountSketch, esta estructura se organiza en d filas de contadores. Sin embargo, TowerSketch está diseñado para manejar distribuciones de datos sesgadas, usando contadores con diferentes anchos de bits en cada fila, de modo que cada fila ocupa la misma cantidad de memoria. En consecuencia, el sketch contiene más contadores pequeños para registrar flujos menos frecuentes y una cantidad menor de contadores grandes para registrar los flujos más frecuentes, lo que es coherente con la distribución típica del tráfico de red.

Cada fila está compuesta por w_i contadores de δ_i bits, con $i \in [1, d]$. El máximo valor válido de un contador

es $2^{\delta_i}-2$. Cuando un contador alcanza el valor $2^{\delta_i}-1$ se considera que está desbordado (hay overflow) y su valor se interpreta como $+\infty$. Cada nuevo paquete se mapea a un contador en cada fila mediante los resultados de d funciones hash aplicadas a su identificador de flujo. Al usar la técnica de Conservative Updates [16], todos los contadores seleccionados que tienen el valor mínimo se incrementan en uno. Este nuevo valor mínimo es la estimación actual de la frecuencia del flujo. En la formulación original del sketch, el número de bits en cada fila consecutiva se duplica con respecto a la anterior.

Queremos capturar el conjunto de los top-K flujos responsables de al menos la mitad del tráfico total, lo que requiere un tamaño mínimo de contador de 8 bits. Por lo tanto, un TowerSketch en su formulación original utiliza 3 filas con contadores de 8, 16 y 32 bits. Sin embargo, debido a la distribución altamente sesgada del tráfico, es preferible aumentar aún más la cantidad de contadores pequeños. Por ello, modificamos el sketch para utilizar un total de 6 filas: 3 filas de contadores de 8 bits, 2 filas de contadores de 16 bits y 1 fila de 32 bits, como se muestra en la Fig. 4.1, pero usando la misma cantidad de memoria que la formulación original.

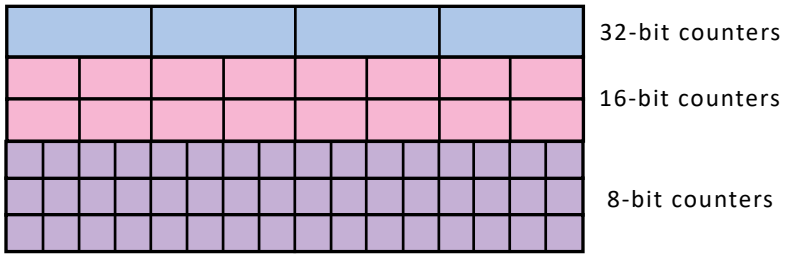


Figura 4.1: Estructura del TowerSketch

El Algoritmo 3 muestra el proceso de inserción del TowerSketch con Conservative Updates (TowerCU), el cual ha demostrado obtener los mejores resultados [16]. Una función hash de 32 bits (MurmurHash3), con semillas distintas para cada fila, mapea el identificador de flujo a un contador por cada fila, utilizando el número de bits necesario para direccionar cada fila según su número de contadores w_i. A continuación, se determina el valor mínimo almacenado en los contadores. Dado que los buckets que han desbordado se consideran como $+\infty$, se verifica que los buckets no estén desbordados antes de actualizar el valor mínimo. Posteriormente, se suma 1 a cada bucket que tenga el valor mínimo y que no esté desbordado, y se escriben los buckets actualizados en el sketch. La estimación de la frecuencia del flujo se muestra en el Algoritmo 4, donde se determina el valor mínimo de los buckets correspondientes al flujo que no estén desbordados.

4.5.2. Priority Queue Array

Para determinar los top-K flujos, debemos insertar cada nueva estimación de frecuencia del sketch en una cola de prioridades de tamaño fijo que mantiene los K flujos con mayor frecuencia. Sin embargo, implementar una cola de prioridades requiere ordenar los elementos en cada nueva inserción, lo que implica un mínimo de $\log_2 K$ accesos a memoria por inserción. Esta operación es costosa y limita el rendimiento del acelerador, especialmente para valores grandes de K.

Como alternativa, usamos un arreglo de colas de prioridad (PQA) propuesto por Soto et al. [41] que aproxima el comportamiento de una cola de prioridad usando R colas de prioridad de S elementos, tal que $K = R \times S$ y $S \ll R$. Una función hash mapea cada flujo a una cola en específico. Debido a las propiedades de la función hash, que busca distribuir los flujos de manera uniforme entre las colas, esta estructura permite capturar

Algoritmo 3: Inserción del TowerSketch

```
Input : Un paquete perteneciente al flujo con identificador f
1  i ← 0
2  while i < d do
       // MurmurHash3 es una función hash de 32 bits
3      hash ← MurmurHash3(f, seeds[i])
4      idx[i] ← hash & (w_i − 1)
       // tower es la matriz del TowerSketch
5      bucket[i] ← tower[i][idx[i]]
       // δ_i es el ancho del bucket en bits
6      if bucket[i] < minval and
7      bucket[i] ≠ 2^δi − 1 then
8          minval ← bucket[i]
9      i ← i + 1
10 i ← 0
11 while i < d do
12     if bucket[i] = minval then
13         if bucket[i] ≠ 2^δi − 1 then
14             bucket[i] ← bucket[i] +1;
15             tower[i][idx[i]] ← bucket[i]
```

Algoritmo 4: Estimación del TowerSketch

```
Input : Un paquete perteneciente al flujo con identificador f
Output: est, estimación de frecuencia de f
1  i ← 0
2  while i < d do
3      hash ← MurmurHash3(f, seeds[i])
4      idx[i] ← hash & (w_i − 1) bucket[i] ← tower[i][idx[i]]
       // δ_i es el ancho del bucket en bits
5      if bucket[i] < minval and
6      bucket[i] ≠ 2^δi − 1 then
7          minval ← bucket[i]
8      i ← i + 1
9  est ← minval
10 return est
```

gran parte de los elementos que capturaría una verdadera cola de prioridades. Además, dado que S es muy pequeño, cada cola puede ser ordenada en un ciclo de reloj, permitiendo, en una implementación hardware, insertar elementos a la cola en tiempo aproximadamente constante.

El Algoritmo 5 muestra el proceso de inserción al PQA. Por cada identificador de flujo, el PQA calcula una función hash de 32 bits. De estos 32, los $log_2 R$ bits menos significativos se usan para seleccionar una de las colas de S elementos. El resto de los bits se usan como una etiqueta que distingue el flujo de los otros que se insertan en la misma cola. Así, cada flujo en la cola es representado por una tupla compuesta por su frecuencia

Algoritmo 5: Inserción en el PQA

Input : Un identificador de flujo f y *est*, la frecuencia estimada de f

```
1  hash ← MurmurHash3(f, seed)
2  idx ← hash & (R − 1)
3  tag ← hash >> log2(R)
4  pqa[idx].find_tag(tag, i, found)
5  if found then
6      if pqa[idx][i].count<est then
7          pqa[idx][i].count ← est
8  else
9      if est > pqa[idx][S-1].count then
10         pqa[idx][S-1].count ← est
11         pqa[idx][S-1].tag ← tag
12 sort( pqa[idx])
```

estimada y la etiqueta del flujo. La etiqueta del flujo entrante se compara con los flujos ya contenidos en la cola. La bandera *found* señala que el flujo entrante ya está presente en la cola y se determina su posición i. Si el flujo se encontró, el PQA verifica que la frecuencia entrante *est* sea mayor que la almacenada (el caso contrario puede producirse por colisiones en la función hash) y de ser así, reemplaza la frecuencia almacenada por la frecuencia entrante. Si el flujo no está presente, el PQA compara *est* con la frecuencia más pequeña almacenada en la cola y mantiene el flujo más frecuente. Finalmente, se ordena la cola de prioridades.

El PQA descrito anteriormente guarda solamente S elementos en cada cola. Sin embargo, debido a colisiones en la función hash, más de S de los top-K flujos pueden ser mapeados a la misma cola. En consecuencia, el PQA pierde los menos frecuentes de estos elementos, disminuyendo su precisión al capturar los top-K. Para mitigar este efecto, proponemos agregar elementos extra a cada cola (e.g. 2 elementos adicionales cuando $S = 4$) para capturar más flujos. Una vez que todos los paquetes fueron procesados, ordenamos el contenido del PQA y extraemos los K flujos más grandes.

4.6. Ordenamiento

Para poder determinar los K flujos con mayor frecuencia en el PQA, es necesario ordenar su contenido. Con este fin, se ordenan en paralelo las S columnas del PQA usando un algoritmo de Radix Sort y luego aplicamos un algoritmo de fusión de S vías (S-way merge) para realizar un ordenamiento global de dichas columnas.

4.6.1. Radix Sort

Radix Sort es un algoritmo de ordenamiento no comparativo que procesa iterativamente cada posición (dígito, bit, o grupo de bits) de los elementos hasta obtener una lista ordenada. Su complejidad es de $O(mn)$, donde n es el número de elementos y m es el número máximo de posiciones en los elementos, por lo que, para un número fijo de bits, su complejidad es lineal al número de elementos. Como veremos más adelante,

nuestras frecuencias son números de 20 bits y usaremos conjuntos de 4 bits en cada posición, por lo que cada columna requerirá 5 iteraciones para quedar ordenada. Partimos procesando el conjunto menos significativo y terminamos con el más significativo.

El Algoritmo 6 muestra el proceso de ordenamiento de menor a mayor para una lista de n enteros de 20 bits. Por cada posición, se realiza el siguiente procedimiento:

- Contamos el número de ocurrencias para cada combinación de 4 bits posible.
- Determinamos el índice final para cada combinación de 4 bits
- Por cada elemento de lista, lo posicionamos según el índice correspondiente a esa combinación. El índice se decrementa cada vez que insertamos un elemento con dicha combinación.

Algoritmo 6: Radix Sort para números de 20 bits con bloques de 4 bits

```
Input: Arreglo lista de n enteros de 20 bits
Output: lista ordenada de menor a mayor
for i ← 0 to 4 do
    Inicializar arreglo contar[0..15] en cero
    Inicializar arreglo indice[0..15] en cero
    for j ← 0 to n − 1 do
        digito ← ( lista[j] >> (4 × i)) & 15
        contar[digito]++
    indice[0] ← contar[0]
    for j ← 1 to 15 do
        indice[j] ← contar[j] + indice[j-1]
    for j ← n − 1 to 0 do
        digito ← (lista[j] > > (4 × i)) & 15
        lista2[indice[digito]-1] ← lista[j]
        indice[digito] ← indice[digito]−1
    lista ← lista2
return lista
```

4.6.2. Fusión de las S columnas

Una vez que ya tenemos las S columnas del PQA ordenadas, usamos un algoritmo de fusión (k-way merge) para obtener una lista ordenada que contenga los elementos de todas las columnas. El Algoritmo recibe las columnas del PQA ordenadas de mayor a menor. El Algoritmo 7 muestra el proceso de fusión para estas columnas. Al principio comenzamos leyendo el primer elemento de cada lista. Por cada posición de la lista final, leemos el elemento de cada lista señalado por el puntero correspondiente y encontramos el máximo válido. Un elemento se considera no válido si el puntero de su lista ha sobrepasado la posición máxima $n-1$. Recorremos los elementos leídos de cada lista para encontrar el mayor valor y su lista de origen (`max_val` y `max_idx`). Finalmente, se escribe en la lista ordenada el valor máximo y se incrementa el puntero correspondiente. La complejidad de este algoritmo es de $O(Sn)$, donde n es el número de elementos de cada lista y S es el número de listas.

Algoritmo 7: Fusión de S listas ordenadas (S-way merge)

Input: `listas[0..S-1]`: arreglo de S listas ordenadas de mayor a menor, cada una con n elementos
Output: `merged`: lista ordenada de mayor a menor con $S \times n$ elementos

```
1  Inicializar punteros[0..S-1] ← 0
2  Inicializar merged ← lista vacía
3  for i ← 0 to S × n − 1 do
4      max_val ← 0
5      max_idx ← −1
6      for j ← 0 to S − 1 do
7          if punteros[j] < n then
8              val ← listas[j][punteros[j]]
9              if val > max_val then
10                 max_val ← val
11                 max_idx ← j
12     merged.push(max_val)
13     punteros[max_idx] ← punteros[max_idx] + 1
14 return merged
```

4.7. Estimación de cardinalidad

Para realizar la estimación de cuantiles, debemos además conocer el número de flujos presentes en el tráfico (cardinalidad). Podemos determinar esta cardinalidad de manera exacta utilizando una cantidad de memoria proporcional al número de flujos. Sin embargo, esto es inviable debido al espacio limitado de los dispositivos de red y la alta tasa de llegada de paquetes. Por ello, estimaremos la cardinalidad usando HyperLogLog, un algoritmo probabilístico propuesto por Flajolet et al. [17]. Este sketch ha sido usado en varias aplicaciones de monitoreo de tráfico de red [54, 1, 15], debido a su alta precisión y bajo uso de memoria.

La estructura de datos del sketch consiste en un arreglo A de tamaño $m = 2^p$ buckets. El algoritmo 8 muestra el proceso de inserción en el sketch. Primeramente, se calcula una función hash de 32 bits para el identificador de flujo f. Los p bits menos significativos se usan para seleccionar un bucket del arreglo A. Luego, se cuenta el número de ceros a la izquierda (leading zeros) del resto de la función hash (v) y se le suma 1. Si el resultado es mayor que el valor actualmente almacenado en el bucket correspondiente, éste se actualiza con el nuevo valor.

Algoritmo 8: Inserción en sketch HyperLogLog

Input : Identificador de flujo f

```
1 hash ← MurmurHash3(f, seed)
2 bucket ← hash[p − 1:0]
3 v ← hash[32:p]
4 if ldz(v)+1 > A[bucket] then
5     A[bucket] ← ldz(v)+1
```

Una vez insertados los flujos al sketch, podemos estimar la cardinalidad recorriendo el arreglo. Para esto,

se calcula la suma armónica de los buckets como:

$$Z = \sum_{j=0}^{m-1} 2^{-A[j]} \tag{4.12}$$

Luego, la cardinalidad estimada está dada por:

$$\hat{N} = \alpha_m \frac{m^2}{Z}, \tag{4.13}$$

donde α_m es una constante del sketch que puede ser aproximada por $\frac{0{,}7213}{1+\frac{1{,}079}{m}}$ para $p \geq 7$.

4.8. Algoritmo de consulta

Este trabajo considera dos tipos de consultas:

- Dada una frecuencia x, determinar en qué cuantil q se encuentra.
- Dado un cuantil q, determinar la frecuencia x que corresponde a ese cuantil.

4.8.1. Consulta de cuantil dado un valor de frecuencia

Finalmente, para estimar el cuantil en el que se encuentra una frecuencia x, considerando $x \geq 1$, deberemos primero estimar cuántos flujos tienen una frecuencia menor o igual a x,es decir, el rank de x, $R(x)$.

Si $x \geq 1$, entonces todos los flujos con frecuencia unitaria tienen una frecuencia menor o igual a x. Esta cantidad de flujos se calcula como:

$$R_1(x) = (\hat{N} - \hat{L}) + 1. \tag{4.14}$$

Luego, debemos analizar los flujos cuya frecuencia sigue una distribución de tipo power law. Para esto, debemos encontrar i_{min}, el menor valor entero de i para el cual $\hat{m}_i \leq x$. Resolvemos la siguiente ecuación:

$$C_m i_{min}^{\hat{\alpha_b}} \leq x \iff i_{min} = \left\lceil \left(\frac{x}{C_m} \right)^{\frac{1}{\hat{\alpha_b}}} \right\rceil \tag{4.15}$$

Una vez encontrado el valor de i_{min}, el número de flujos cuya frecuencia sigue una power law y es menor o igual a x es igual a:

$$\text{Si } \hat{L} \leq \hat{N} : R_2(x) = \begin{cases} 0 & \text{si } i_{min} \geq \hat{L} \\ \hat{L} - i_{min} & \text{si } K < i_{min} < \hat{L} \\ \hat{L} - K - 1 & i_{min} \leq K \end{cases} \tag{4.16}$$

$$\text{Si } \hat{L} > \hat{N} : R_2(x) = \begin{cases} 0 & \text{si } i_{min} \geq \hat{N} \\ \hat{N} - i_{min} + 1 & \text{si } K < i_{min} < \hat{N} \\ \hat{N} - K & i_{min} \leq K \end{cases} \tag{4.17}$$

Por último, para estimar el número de flujos que pertenecen a los top-K y cuya frecuencia es menor o igual a x, se busca el primer elemento de los top-K tal que $m_i \leq x$ con $1 \leq i \leq K$. Si este elemento no se encuentra, entonces $R_3(x) = 0$. De lo contrario, $R_3(x) = K - i + 1$

Con todo lo anterior, el cuantil en el que se encuentra la frecuencia x está dado por:

$$q = \begin{cases} \frac{R_1(x)+R_2(x)+R_3(x)}{\hat{N}} & \text{si } \hat{L} \leq \hat{N} \\ \frac{R_2(x)+R_3(x)}{\hat{N}} & \text{si } \hat{L} > \hat{N} \end{cases} \tag{4.18}$$

4.8.2. Consulta de frecuencia dado un valor de cuantil

Dado un cuantil q con $q \in [0, 1]$, el rank estimado que corresponde a ese cuantil se expresa como :

$$\hat{R} = \lceil q\hat{N} \rceil \tag{4.19}$$

Para encontrar el indice i del flujo que corresponde a ese rank, asumiendo que los flujos están ordenados de mayor a menor frecuencia, debemos encontrar el flujo que tiene por lo menos $\hat{R}$ flujos con frecuencia menor o igual. Podemos expresar este índice de la siguiente manera:

$$i = (\hat{N} - \hat{R}) + 1 \tag{4.20}$$

Una vez encontrado i, se pueden dar los siguientes casos:

- Si $i \leq K$, consultamos las frecuencias de los top-K para obtener la frecuencia del flujo i
- Si $K < i \leq \hat{L}$, estimamos la frecuencia mediante la ecuación $x = C_m i^{\alpha_b}$
- Si $i \geq \hat{L}$ entonces su frecuencia es unitaria

Capítulo 5. Arquitectura del Acelerador Hardware

En este capítulo, describimos en detalle la arquitectura del acelerador hardware que implementa los algoritmos presentados anteriormente. Especificamos el funcionamiento y arquitectura interna de cada módulo que compone el sistema, así como también las interconexiones que permiten la comunicación y sincronización entre ellos.

La Figura 5.1 muestra la topología de red considerada para el diseño de este acelerador. El acelerador está pensado como un punto de monitoreo pasivo. Los paquetes llegan al dispositivo FPGA a través de un Test Access Point ubicado en el enlace entre el router de borde y el switch de la red. Un TAP (Test Access Point) es un dispositivo de hardware pasivo que se inserta físicamente en un enlace de red con el fin de obtener una copia íntegra y continua del tráfico que circula por dicho enlace. Aunque el dispositivo FPGA no permite realizar acciones correctivas, puede exportar alertas e información a elementos existentes de la red como switches, routers y firewalls.

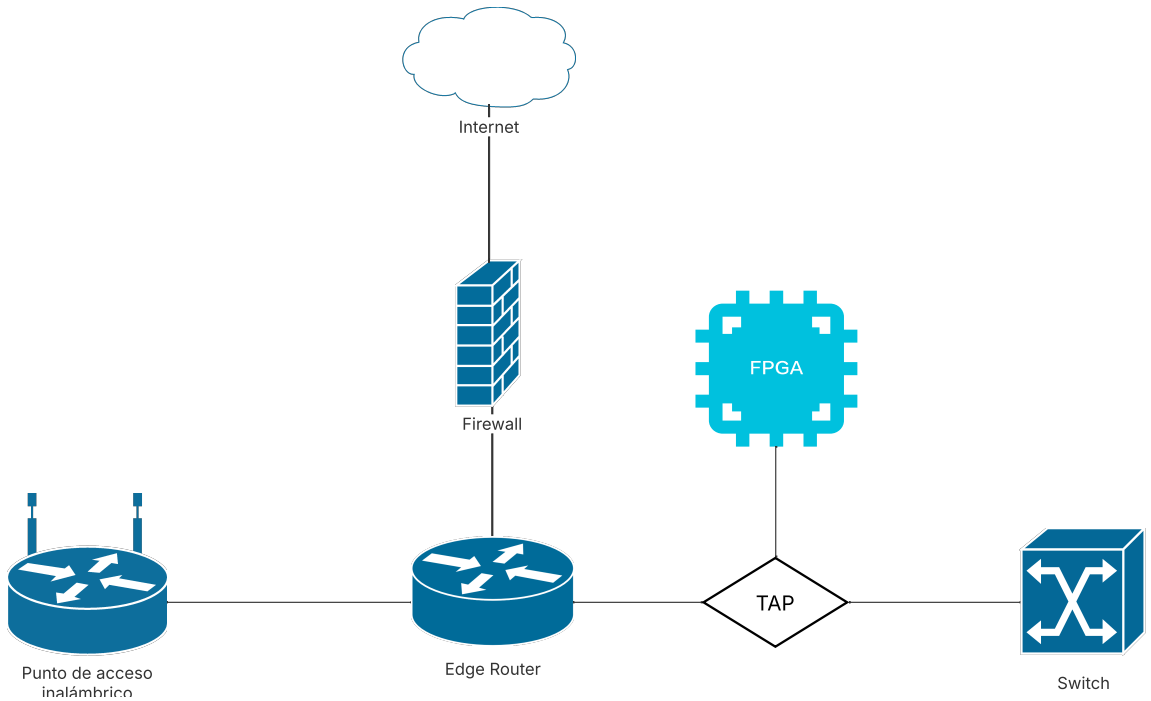


Figura 5.1: Topología de red

5.1. Arquitectura general

La Figura 5.2 presenta la arquitectura general del acelerador. En primer lugar, cada paquete entrante se inserta en paralelo en las estructuras TowerSketch y HyperLogLog. Para esto, ambas estructuras reciben un identificador de flujo `flow_id` de 104 bits, el cual se construye concatenando cinco campos del encabezado del paquete: los campos de 32 bits correspondientes a las direcciones IP de origen y destino, los campos de 16 bits para los puertos de origen y destino, y el campo de 8 bits para el protocolo. Además, reciben una

señal in_valid que indica si flow_id corresponde al identificador de flujo de un paquete válido que debe ser insertado.

Por cada paquete, el TowerSketch entrega una estimación de frecuencia est acompañada de una señal ready que indica cuando dicha estimación es válida. Además, genera out_id, que corresponde a una hash de 32 bits de flow_id e identifica el flujo cuya frecuencia fue estimada por el TowerSketch. Cuando ready está en alto, se inserta la tupla {est,out_id} en el arreglo de colas de prioridad PQA.

Una vez que fueron procesados todos los paquetes, el PQA recibe una señal output indicando que comienza el proceso de lectura de su contenido. El PQA entrega S tuplas {count,out_id} que corresponden a los valores almacenados en una fila de la estructura. Cada uno de estos valores se inserta en uno de los S módulos RadixSort que ordenan las columnas del PQA en paralelo. Estos módulos se sincronizan con el PQA usando las señales rd_ready y rd_valid. Una vez ordenadas las columnas, cada módulo entrega los elementos de la columna ordenados de mayor a menor, uno en cada ciclo de reloj, y los guarda en r_sorted.

Cuando el módulo Merge detecta que la señal r_valid es igual a 1, inicia la lectura de los datos proporcionados por los módulos RadixSort y los guarda en S memorias independientes. Luego, estas memorias son fusionadas de manera ordenada para generar una memoria que contenga los elementos de todas las columnas ordenados de mayor a menor. Los K primeros elementos de esta memoria representan las frecuencias de los top-K flujos, las cuales serán enviadas a un procesador soft-core. Un procesador soft-core es un procesador implementado con la lógica reconfigurable del FPGA. El conjunto de instrucciones soportado y los módulos integrados son configurables. En particular, usamos el procesador NEORV32 [55] y enviamos los top-K mediante una interfaz SLINK.

En paralelo a lo anterior, una vez procesados todos los paquetes (simbolizado por la señal output), el sketch HyperLogLog entrega a través de GPIO la suma armónica del contenido del sketch harm_sum y una señal sum_valid que indica que la estimación está disponible.

El procesador embebido estima la cardinalidad usando la suma armónica y usa las frecuencias entregadas para calcular los parámetros de la distribución *power-law* y realizar las consultas de cuantiles.

El resto de este capítulo describe los detalles de operación de cada módulo y sus componentes principales.

5.2. MurmurHash3

En esta sección describiremos la arquitectura y funcionamiento del módulo que implementa el Algoritmo 2, correspondiente al cálculo del valor hash de un elemento usando la función MurmurHash3.

La Figura 5.3 muestra el procedimiento de cálculo de un valor hash para el identificador de flujo. Dicho identificador de flujo tiene 104 bits, los cuales se completan con ceros por la izquierda hasta llegar a 128 bits, dado que la función hash trabaja con bloques de 32 bits.

En primer lugar, para cada bloque de 32 bits de la entrada, se ejecuta la etapa “Prologue”, correspondiente a las líneas 10-12 del algoritmo:

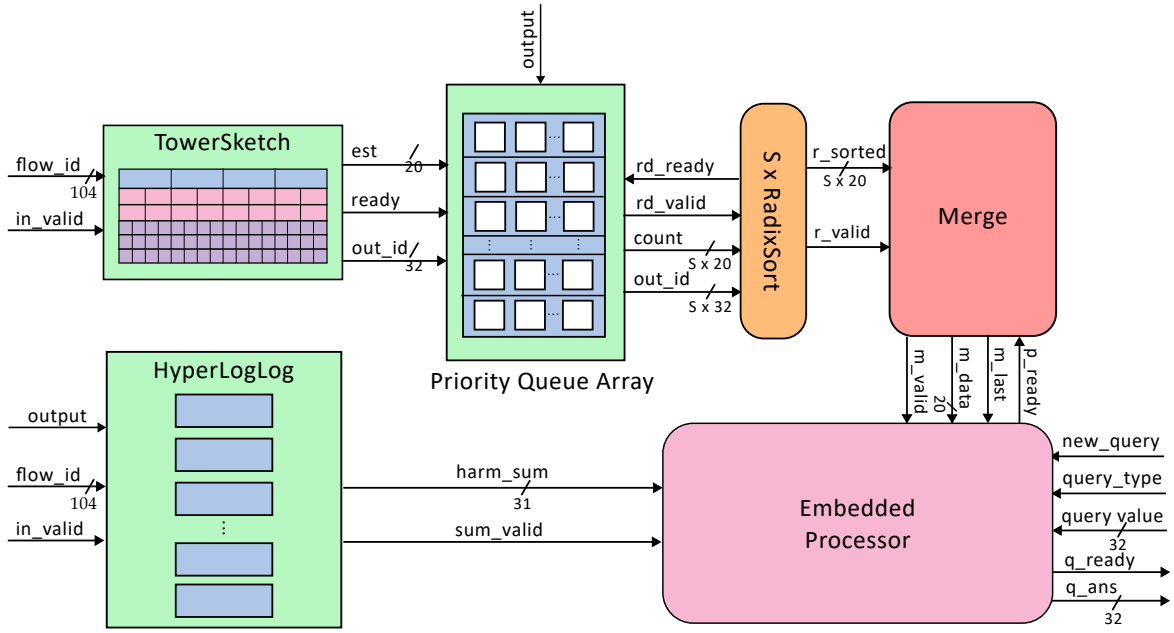


Figura 5.2: Arquitectura general del acelerador

Los ID de flujo de cada paquete entrante son insertados en paralelo en el TowerSketch y en HyperLogLog. Luego, las estimaciones de frecuencia del Tower son insertadas al PQA junto con un valor hash que identifica el flujo. La señal output le indica al sketch HLL y al PQA que ya se insertaron todos los paquetes. Cuando esto sucede, el HyperLogLog envía la suma harmonica de su contenido al procesador embebido y el PQA entrega el contenido de cada una de sus columnas a 6 módulos de RadixSort, los cuales ordenan las columnas en paralelo. El módulo Merge ordena las columnas globalmente y envía las top-32768 frecuencias al procesador a través de SLINK. El procesador calcula los parámetros de la power-law y comienza a recibir consultas de cuantiles. Una consulta está compuesta por 3 señales, una señal que indica que hay una consulta nueva, una señal que indica el tipo de consulta a realizar (cuantil o frecuencia) y el valor por el que se está consultando. El procesador entrega una señal ready cuando la respuesta a la consulta está lista y la guarda en la señal q_ans.

$$
\begin{aligned}
k_1 &\leftarrow k \times c1 \\
k_2 &\leftarrow (k_1 \ll r1) | (k_1 \gg (32 - r1)) \\
k_3 &\leftarrow k_2 \times c2
\end{aligned}
$$

Dado que no existe dependencia entre estas operaciones para los distintos bloques, pueden ejecutarse en paralelo para los distintos bloques de 32 bits de la entrada. Para reducir la latencia del camino crítico, se usan dos ciclos de reloj para cada multiplicación. En cambio, la rotación se puede realizar en un solo ciclo de reloj. Considerando esto, la etapa “Prologue” se ejecuta en 5 ciclos de reloj.

A continuación, ejecutamos la etapa “Body” correspondiente a las líneas 13-15 del algoritmo para el primer bloque de 32 bits:

$$
\begin{aligned}
h_1 &\leftarrow h \oplus k \\
h_2 &\leftarrow (h_1 \ll r2) | (h_1 \gg (32 - r2)) \\
h_3 &\leftarrow h_2 \times m + n
\end{aligned}
$$

Dado que el valor de h cambia luego de cada iteración, la ejecución de la etapa “Body” del segundo bloque debe esperar a que termine la ejecución de esta etapa para el primer bloque y así sucesivamente. Esta etapa

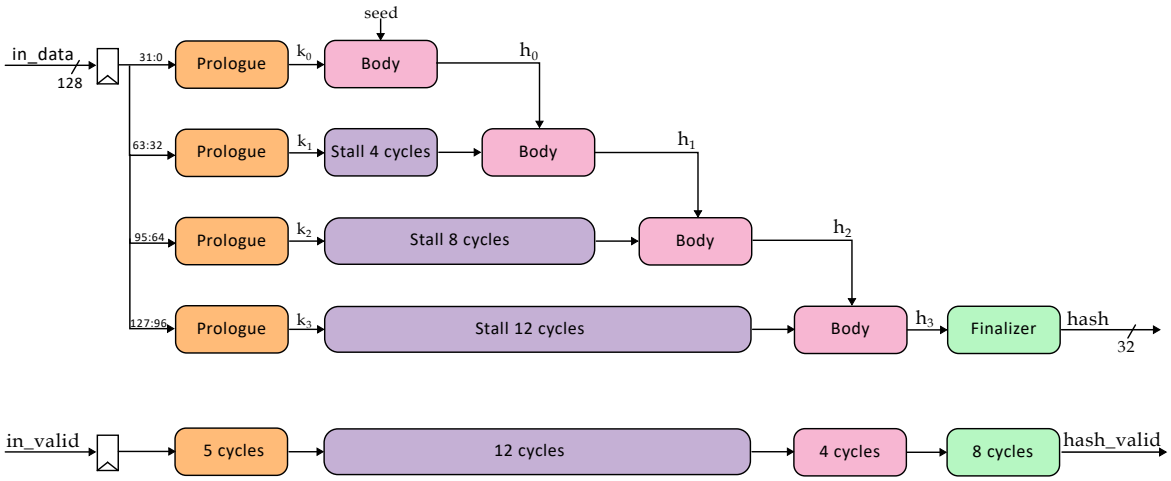


Figura 5.3: Cálculo de MurmurHash3

tarda 4 ciclos de reloj en ejecutarse (la última instrucción se ejecuta en dos ciclos), por lo que atrasamos la salida de "Prologue" en múltiplos de 4 según corresponda para cada bloque. La etapa "Body" del último bloque de 32 bits termina de ejecutarse 16 ciclos después de "Prologue", 12 de retraso y 4 de ejecución.

Finalmente, a la salida de la etapa "Body" del último bloque se le aplica la etapa "Finalizer", que corresponde a las líneas finales del algoritmo:

$$
\begin{aligned}
h_4 &\leftarrow h_3 \oplus len \\
h_5 &\leftarrow h_4 \oplus (h_4 \gg 16) \\
h_6 &\leftarrow h_5 \times \texttt{0x85ebca6b} \\
h_7 &\leftarrow h_6 \oplus (h_6 \gg 13) \\
h_8 &\leftarrow h_7 \times \texttt{0xc2b2ae35} \\
h_9 &\leftarrow h_8 \oplus (h_8 \gg 16)
\end{aligned}
$$

Esta etapa se ejecuta en 8 ciclos y entrega como resultado en `hash` el valor hash del elemento entrante.

Considerando todos los retardos anteriores, la entrada `in_valid` se retrasa $5 + 12 + 4 + 8 = 29$ ciclos para producir la señal `hash_valid` que indica que el cálculo de la hash terminó y el valor guardado en `hash` es válido. El módulo está implementado en forma de pipeline, lo que le permite recibir un identificador de flujo en cada ciclo de reloj, generando su valor hash después de 29 ciclos.

5.3. TowerSketch

La Figura 5.4 muestra la arquitectura del módulo TowerSketch, el cual implementa el Algoritmo 3. Como se discute en la sección de Algoritmos, la estructura está compuesta por 3 filas de contadores de 8 bits, 2 filas de contadores de 16 bits, y 1 fila de contadores de 32 bits.

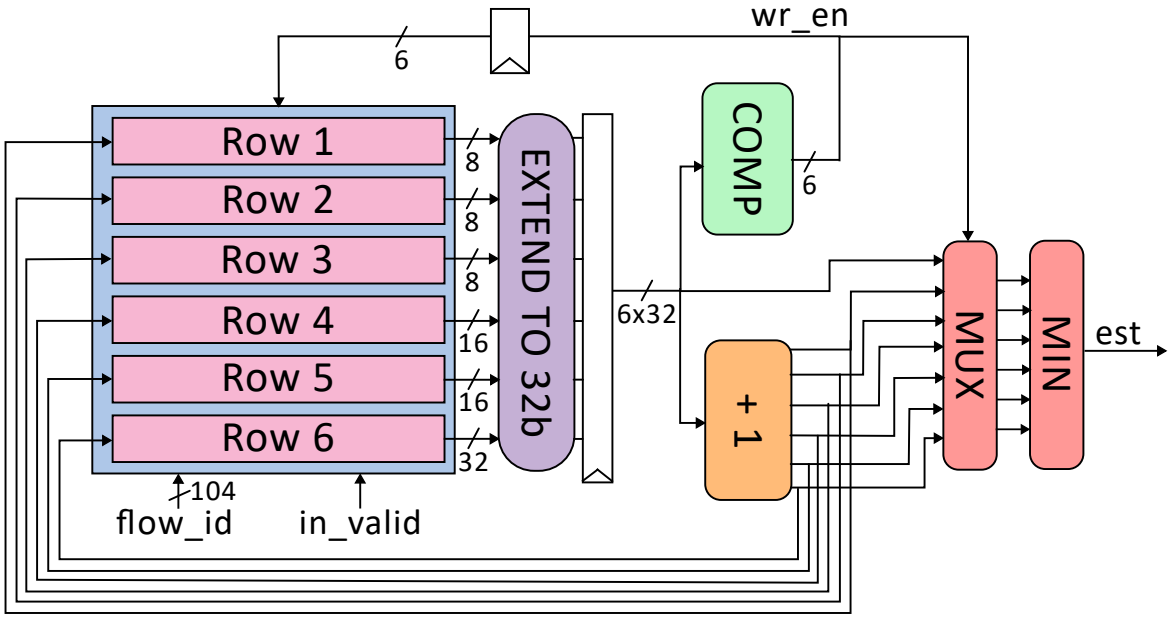


Figura 5.4: Arquitectura del módulo TowerSketch

Al identificador de flujo `flow_id` se le aplica una MurmurHash3 por cada fila, que se utiliza para leer el valor almacenado en un bucket de la fila.

Para comparar los valores leídos, éstos se extienden a 32 bits, de modo que si un bucket se encuentra desbordado, su valor extendido se establece en $2^{32} - 1$. Cada valor se compara con los demás, generando la señal de 6 bits `wr_en`, donde cada bit se activa solo si el bucket de esa fila contiene el valor mínimo y, por lo tanto, debe ser actualizado. El bloque `+1` suma 1 a todos los buckets que no están desbordados. Los valores incrementados se redirigen nuevamente al sketch, pero solo se escriben en cada fila si el bit correspondiente de la señal `wr_en` está activado.

Luego, un multiplexor selecciona entre el valor incrementado y el valor anterior de cada bucket. Finalmente, un árbol de comparadores calcula el mínimo de estos valores para entregar la estimación actual de la frecuencia del flujo.

La sección de memoria del sketch, en azul en la Figura 5.4, está compuesta por seis filas. En nuestra implementación sobre un FPGA AMD UltraScale+, cada fila se implementa utilizando UltraRAMs de 4K×64 bits. El número de UltraRAMs empleado por cada fila depende del ancho total de la fila $m = w_i \times \delta_i$, donde w_i es el número de contadores en la fila i y δ_i es el ancho en bits de dichos contadores. Para esta implementación, escogimos $m = 2^{21}$, por lo que se requieren 8 UltraRAMs por fila.

A modo de ejemplo, la Figura 5.5 muestra el circuito de lectura de una fila con bucket de 8 bits. Cuando se inserta un paquete, el sketch calcula un valor hash de 32 bits del identificador de flujo. Los 18 bits menos significativos del valor hash resultante se utilizan para seleccionar un contador de 8 bits de la fila del sketch: los 8 bloques UltraRAM se direccionan usando los 12 bits intermedios, seleccionando así una palabra de 64 bits de cada bloque; los 3 bits superiores seleccionan una de estas 8 palabras; y los 3 bits menos significativos seleccionan uno de los ocho contadores de 8 bits que conforman la palabra de 64 bits. Esta última selección se realiza desplazando la palabra de 64 bits seleccionada en múltiplos de 8 bits. El número de bits en el que debemos desplazar la palabra seleccionada para seleccionar el bucket j está dado por $8j$, por lo que desplazamos

a la izquierda en entre 0 y 56 bits. Finalmente, conservamos los ocho bits menos significativos del resultado.

Después de actualizar los contadores, los buckets modificados se actualizan en la memoria. El sketch inserta el valor modificado del bucket en la palabra previamente leída, actualizando solamente los bits correspondientes. Luego, escribe esta palabra en la memoria seleccionada, utilizando los 12 bits intermedios del hash como dirección de memoria y los 3 bits superiores para seleccionar la UltraRAM que se escribe.

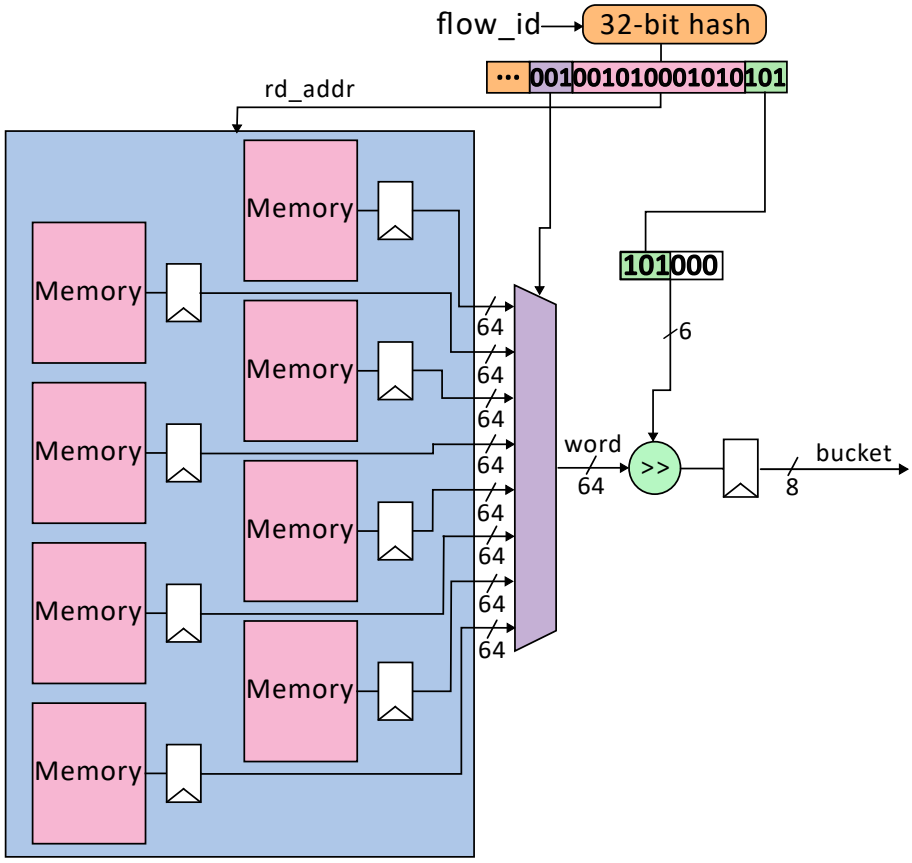


Figura 5.5: Circuito de lectura para una fila del TowerSketch con contadores de 8 bits

Para una fila de contadores de 16 bits, se utilizan 17 bits del valor hash, de modo que los 2 bits menos significativos seleccionan uno de los cuatro contadores de 16 bits. Para las filas de contadores de 32 bits, se emplean 16 bits del valor hash, y el bit menos significativo selecciona uno de los dos contadores de 32 bits. En general, para una fila de 2^{21} bits compuesta por contadores de δ_i bits, el sketch utiliza $21 - \log_2 \delta_i$ bits del valor hash para direccionar la memoria, de los cuales 12 bits se emplean para direccionar los ocho bloques UltraRAM, 3 bits para seleccionar una palabra de 64 bits y $6 - \log_2(\delta_i)$ bits para desplazar la palabra (en múltiplos de δ_i bits) y obtener el valor del contador.

El módulo del sketch está implementado usando pipeline, lo que le permite recibir un nuevo paquete en cada ciclo de reloj. Entre la lectura de la palabra en memoria y la escritura del valor actualizado de la palabra transcurren 4 ciclos de reloj. Esto puede provocar conflictos, denominados hazards, cuando dos flujos que se mapean a la misma palabra ingresan al sketch con una diferencia de cuatro ciclos de reloj o menos. Para resolver este problema, se detectan los casos en los que puede haber conflictos y se aplican técnicas de forwarding para resolverlos.

5.4. Priority Queue Array

El módulo PQA utiliza S bloques de memoria en paralelo, cada uno con R elementos almacenados. En cada inserción se implementa el Algoritmo 5. En la implementación hardware, reutilizamos el valor hash calculado por la primera fila de sketch, y tomamos los $log_2 R$ bits menos significativos para seleccionar indexar los S bloques de memoria. Los elementos leídos constituyen una de las colas de prioridad de S elementos. El resto de los bits del valor hash se almacenan en `tag` y se usan para identificar el flujo. Dicho flujo se inserta con una frecuencia `est`. Un circuito combinacional compara la tupla `{tag,est}` entrante con todas los elementos de la cola y actualiza su contenido según los resultados de esta comparación.

La Figura 5.6 muestra el circuito de actualización del PQA. Cada bloque de memoria puede recibir como entrada una de las siguientes posibilidades: la tupla entrante, el elemento contenido en la memoria de la derecha (frecuencia mayor) o el mismo elemento contenido anteriormente en la memoria.

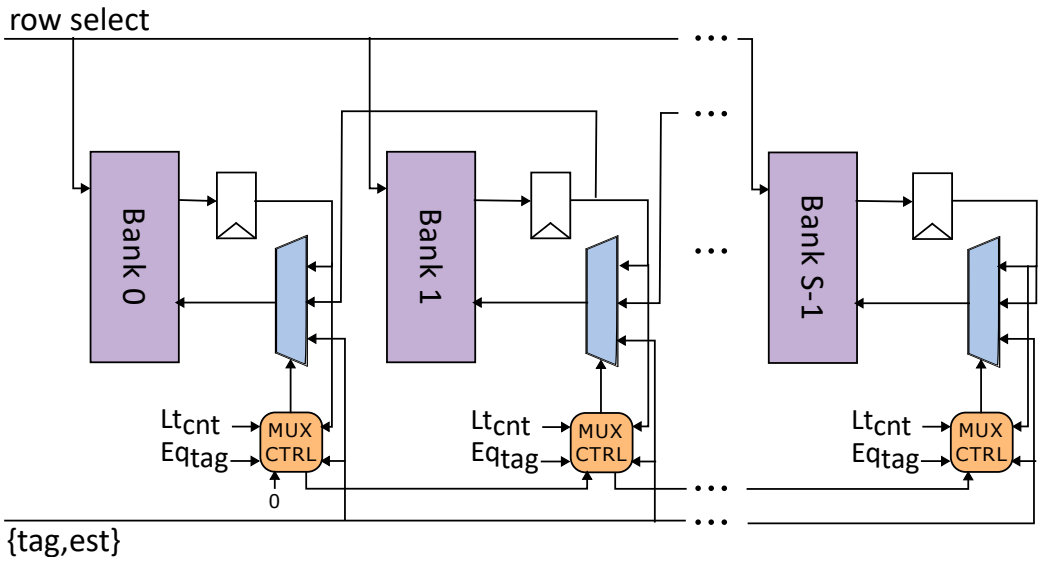


Figura 5.6: Circuito de actualización del PQA

Para determinar la entrada de cada memoria, se consideran los siguientes casos:

- **El tag entrante no se encuentra en la cola y la frecuencia entrante es mayor que el menor elemento en la cola:** El elemento entrante debe ser insertado. Para esto, computamos una señal $\texttt{Lt}_{cnt}$ que representa un código termómetro, donde un 1 indica que el elemento en esa posición de la cola tiene una frecuencia menor a la frecuencia entrante. La transición de 1 a 0 indica la posición donde debe ser insertado el elemento. Los elementos con menor frecuencia son desplazados hacia el bloque de la izquierda, descartando el elemento menos frecuente cuando la memoria está llena.
- **El tag entrante está en la cola y la frecuencia entrante es mayor a la registrada anteriormente:** La frecuencia del elemento debe ser actualizada en la cola. Se deben desplazar los elementos que deben cambiar su posición debido a esta actualización. Esto se determina mediante otro código termómetro llamado $\texttt{Eq}_{tag}$, cuya transición de 1 a 0 se produce en el elemento siguiente al que tiene el mismo tag que el elemento entrante. Una operación XOR con el código $\texttt{Lt}_{cnt}$ determina qué elementos deben ser desplazados hacia la izquierda.

- **El tag entrante no se encuentra en la cola y la frecuencia entrante es menor o igual que el menor elemento en la cola:** No se realizan cambios en la cola.

Una máquina de estado maneja la transición entre el estado de recepción de elementos y el estado de lectura del contenido del PQA. En el estado de lectura, se entregan S tuplas de elementos correspondientes al contenido de una cola del PQA.

Por otro lado, el módulo del PQA está implementado usando pipeline, para poder insertar un elemento por ciclo de reloj. Considerando que, para $K = 32768$, el arreglo de colas contiene 8192 colas, cada columna se implementa con dos UltraRAMs que contienen 4096 elementos cada una, por lo que el módulo usa dos ciclos de reloj para la lectura de memoria, uno para leer ambas memorias y otro para seleccionar una de las dos. Además, se usa un ciclo para determinar las entradas de cada memoria después de la inserción del nuevo elemento, y otro para escribir estas entradas en la memoria. Por lo tanto, entre la lectura y la escritura de la memoria transcurren 3 ciclos de reloj. Para resolver posibles conflictos debidos a esta latencia, se implementa un forwarding de 3 ciclos, permitiéndonos operar sin hazards (por ejemplo, un hazard ocurre cuando se intenta leer una dirección en memoria antes de que se haya escrito el nuevo valor).

5.5. Ordenamiento

Como discutimos anteriormente, el ordenamiento de las frecuencias contenidas en el PQA está dividido en dos etapas. En primer lugar, se ordenan en paralelo las S columnas del PQA. Luego, se ordenan los elementos de estas columnas usando un algoritmo de fusión de S vías. De esta manera, obtenemos los elementos más frecuentes ordenados de mayor a menor.

5.5.1. RadixSort

El módulo de RadixSort implementa el Algoritmo 6. El módulo es controlado por una máquina de estado que divide el algoritmo en cuatro etapas principales:

- **Etapa de recepción:** Se reciben los datos a ordenar y se guardan en una memoria que llamaremos buffer de entrada.
- **Etapa de conteo:** Se recorre el buffer de entrada y se divide el elemento leído en grupos de 4 bits. Por cada grupo, se cuenta cuantas repeticiones de cada combinación posible de 4 bits existen en el buffer.

  La Figura 5.7 muestra cómo se implementa la etapa de conteo para un ejemplo de 10 números de 8 bits. Para cada elemento contenido en el buffer, se lee el contenido de la memoria y se divide en grupos de 4 bits. En este caso, dividimos cada elemento en dos partes: del bit 0 al bit 3 y del bit 4 al bit 7. Cada uno de estos grupos se utiliza como índice para acceder a dos arreglos de conteo: Count[0] y Count[1], cada uno de 16 posiciones (una por cada posible combinación de 4 bits) inicializadas en 0. El contenido de cada arreglo en su índice correspondiente es incrementado en 1. Luego, se escribe cada valor incrementado en el arreglo Count correspondiente. Así, Count$[i][j]$ almacena el número de ocurrencias del patrón de 4 bits igual a j en la posición i del número. El contenido del bloque Buffer de entrada representa los 10 datos a ordenar y el contenido de los bloques Count muestra el estado final de estos arreglos, luego de haber leído todos los elementos del buffer.

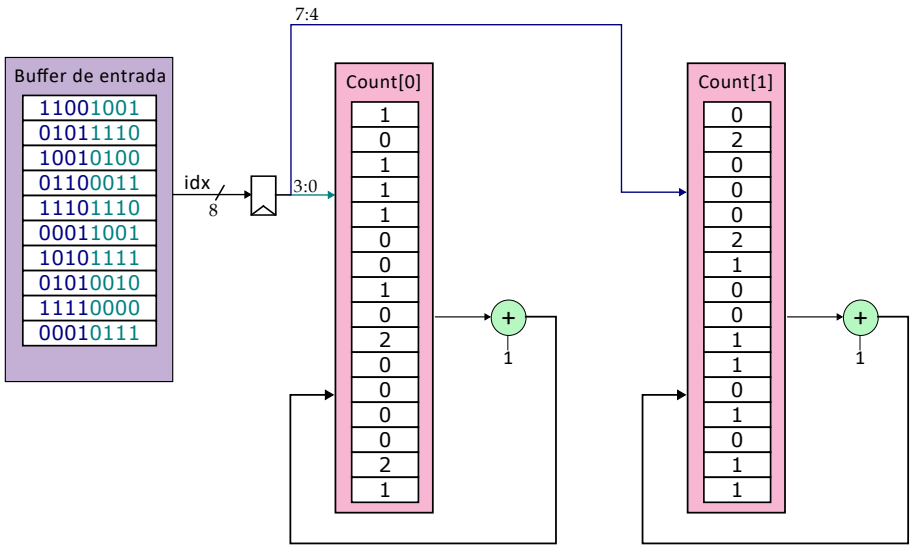


Figura 5.7: Etapa de conteo del módulo RadixSort

- **Etapa de acumulación:** En esta etapa se usan los arreglos Count construidos en la etapa anterior, para determinar el último índice en el que se escribirá un cierto patrón de 4 bits. Para un patrón j en la posición i, $Accum[i][j]$ está dado por:

$$Accum[i][j] = \begin{cases} Count[i][j] - 1 \text{ si } j = 0 \\ Count[i][j] + Accum[i][j-1] \text{ si } j \neq 0 \end{cases} \tag{5.1}$$

  La Figura 5.8 muestra la ejecución de la etapa de acumulación correspondiente al ejemplo ilustrado en la Figura 5.7. Para cada posición i, se recorre secuencialmente el arreglo Count$[i]$, utilizando el índice `r_idx_count`. A partir de la segunda posición (es decir, excluyendo el primer elemento), el valor leído en Count$[i]$[`r_idx_count`] se suma con el valor almacenado en Accum$[i]$[`r_idx_acc`], donde `r_idx_acc` = `r_idx_count` −1. El resultado de esta suma se escribe en Accum$[i]$[`w_idx_accum`], siendo `w_idx_accum` = `r_idx_acc`. De esta manera, Accum$[i]$ contendrá por cada patrón el último índice donde se debe escribir. El contenido de los bloques Accum muestra el estado final de estos arreglos después de haber recorrido Count por completo.

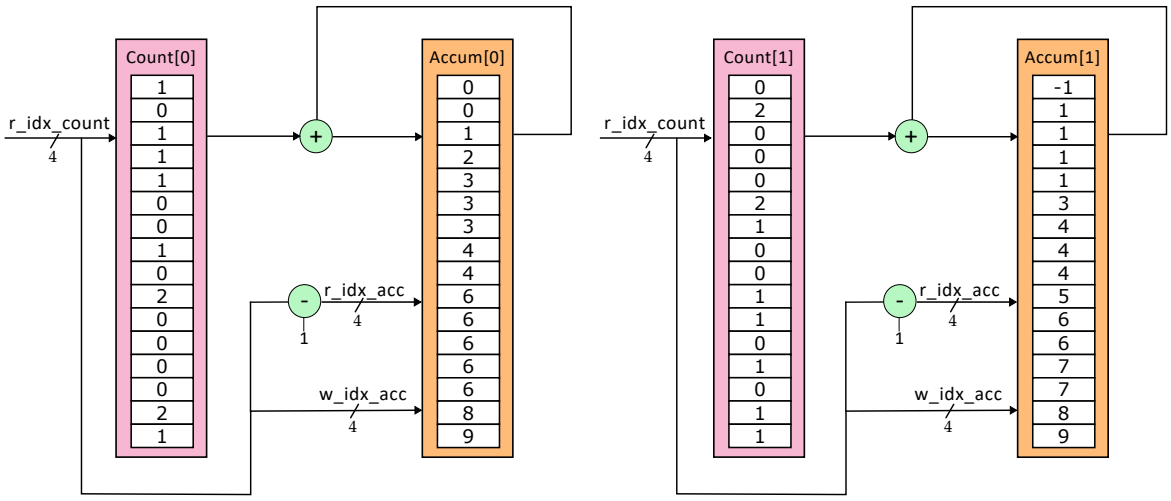


Figura 5.8: Etapa de acumulación del módulo RadixSort

- **Etapa de escritura:** En la etapa de escritura, se recorre el buffer de entrada de manera inversa (desde el último elemento al primero). Como muestra la Figura 5.9, en la primera etapa se escriben los elementos en el buffer de salida ordenados de menor a mayor según sus 4 bits menos significativos. Para hacer esto, se usan estos bits como índice de lectura para el arreglo Accum. El resultado de esta lectura corresponde al índice del buffer de salida donde se escribe el elemento leído del buffer de entrada. Luego, se decrementa en uno el valor leído de Accum para que, si existe otro elemento con el mismo patrón de bits, éste se escriba en la posición anterior.

  En la segunda etapa, el buffer de salida de la etapa 1 pasa a ser el buffer de entrada. Esto se implementa en hardware usando dos buffers, que alternan sus funciones: cuando uno se lee, el otro se escribe y luego se intercambian. Se aplica el mismo procedimiento de la etapa 1 pero esta vez con los bits más significativos, escribiendo los elementos ordenados de menor a mayor según estos 4 bits. Una vez terminada esta etapa, el buffer de salida contiene los elementos ordenados de menor a mayor. Cabe mencionar que la posibilidad de ordenar los elementos por etapas y obtener un resultado globalmente ordenado se debe a la estabilidad del algoritmo, que preserva el orden relativo de elementos iguales.

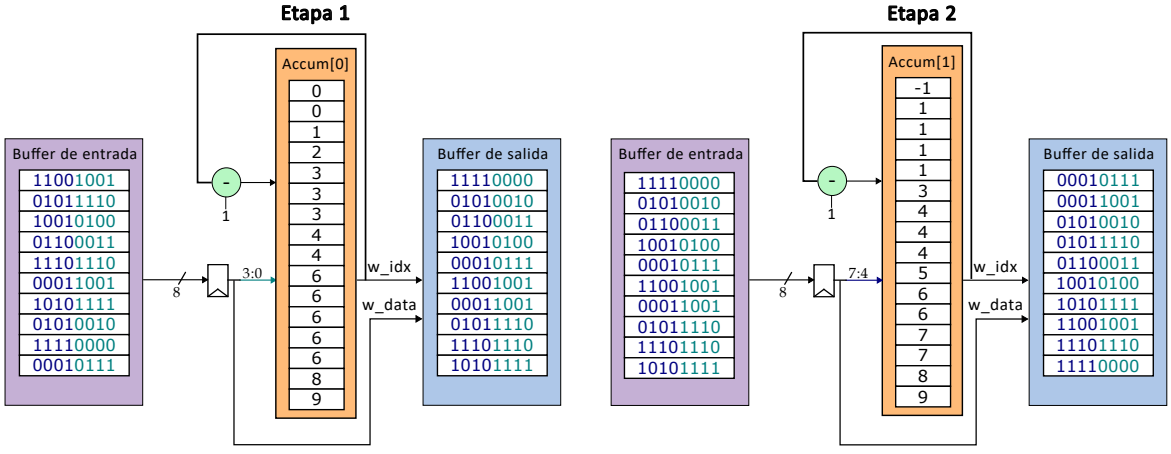


Figura 5.9: Etapa de escritura del módulo RadixSort

Una vez ejecutadas estas etapas, basta con leer el buffer de salida desde el último elemento hasta el primero, para entregar al módulo de fusión los elementos ordenados de mayor a menor.

Para ordenar una columna de elementos de 20 bits, se necesitan 6 pasadas por los buffers: Una para contar las ocurrencias de cada patrón, y una por cada grupo de 4 bits para ordenar los elementos según los 4 bits correspondientes. Además, en la etapa de acumulación se deben recorrer los arreglos Count de 16 elementos. Considerando esto, una vez leída la entrada, para una columna de 8192 elementos, el algoritmo genera un arreglo ordenado en $6 \times 8913 + 16 = 49174$ ciclos de reloj.

5.5.2. Fusión de S vías

Luego de que las columnas del PQA hayan sido ordenadas en paralelo en los módulos RadixSort, la salida de estos módulos se guardan en S memorias distintas, las cuales contienen las columnas ordenadas de mayor a menor. Este módulo implementa el Algoritmo 14.

La Figura 5.10 muestra el proceso de fusión de 6 columnas. La estructura Pointers almacena los punteros de lectura para cada una de las memorias de entrada, los cuales están inicializados en 0. Usando estos punteros, se lee el elemento correspondiente de cada memoria. Estos elementos leídos se comparan usando un árbol, el cual después de 3 ciclos de reloj determina cuál es el elemento más grande de los leídos y de qué memoria fue leído este elemento. El elemento más grande se escribe en la memoria de salida. Además, se incrementa el puntero de lectura de la memoria a la que corresponde el elemento seleccionado. En cada comparación se verifica que el puntero de lectura apunte a una posición válida de la memoria, para que el elemento leído sea válido. El proceso continúa hasta que se hayan escrito K elementos en la memoria.

Estos K valores representan las frecuencias de los K flujos más frecuentes, las cuales se envían al procesador Neorv32 mediante la interfaz SLINK. Dado que el procesador (PS) y la lógica programable (PL) usan distintas frecuencias de reloj, se maneja el cruce de dominios de reloj usando sincronizadores y se coordina la transmisión de datos mediante las señales `m_valid`, `p_ready` y `m_last`.

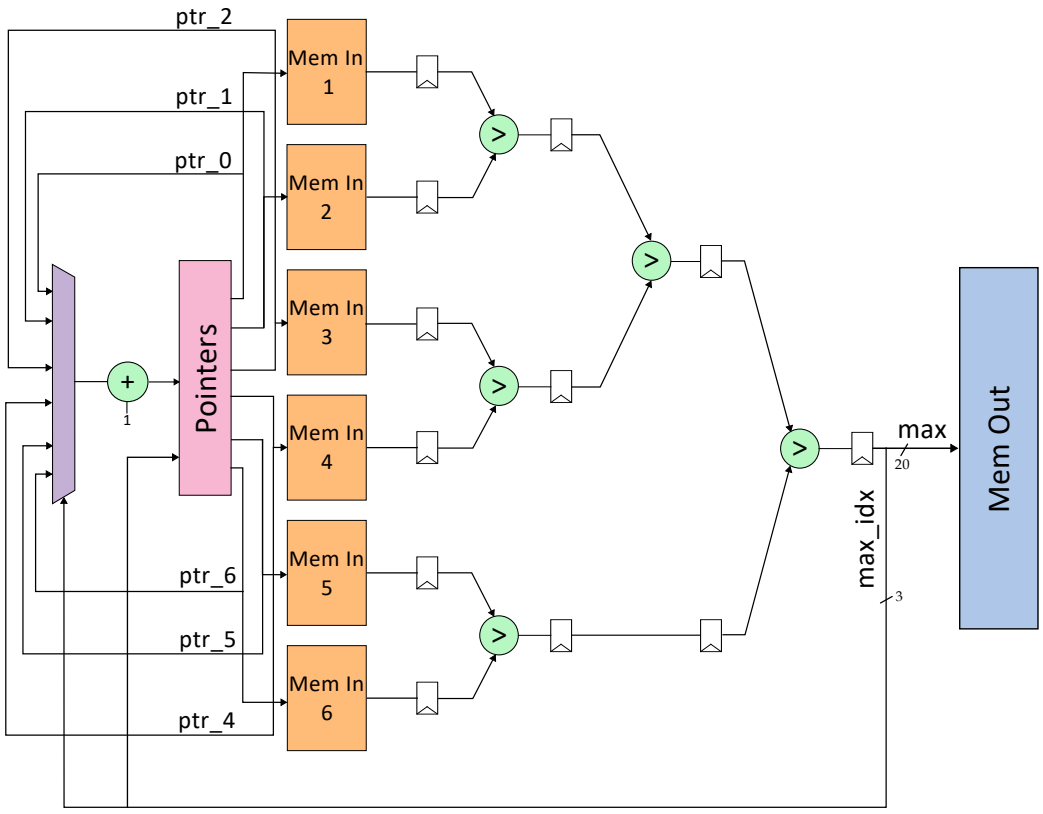


Figura 5.10: Proceso de fusión de 6 vías

5.6. Estimación de cardinalidad

La estimación de la cardinalidad de los flujos se hace usando un sketch HyperLogLog. La Figura 5.11 muestra el proceso de inserción de un flujo en el sketch, descrito anteriormente en el Algoritmo 8. A cada identificador de flujo se le aplica la función MurmurHash3, obteniendo un valor hash de 32 bits. Los p bits menos significativos se usan para indexar la memoria del sketch. El resto de los bits entran al módulo "LDZ" que determina el número de ceros por la izquierda de esta entrada y le suma uno.

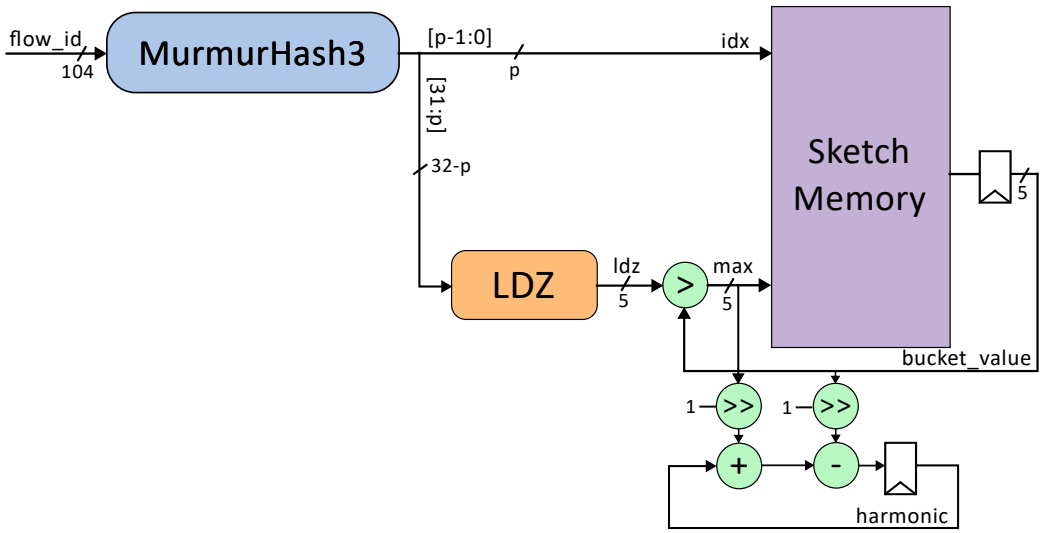


Figura 5.11: Proceso de inserción y acumulación en el sketch HyperLogLog

La Figura 5.12 muestra el proceso de determinación del número de ceros por la izquierda para una entrada de 18 bits. En primer lugar, se divide la entrada en 3 slices de 6 bits cada uno. Se le aplica una función OR a estos slices para producir una señal de 3 bits, que indica si el slice correspondiente tiene al menos un bit en 1. En paralelo, el módulo "Leftmost Index" busca el índice (entre 1 y 18) del 1 más significativo dentro de cada slice. Un multiplexor con prioridad selecciona el índice del slice más significativo que contenga un 1, éste es el índice del 1 más significativo de la entrada. A continuación, se le resta este índice a 19 (el largo de la entrada más uno) para encontrar la cantidad de ceros por la izquierda.

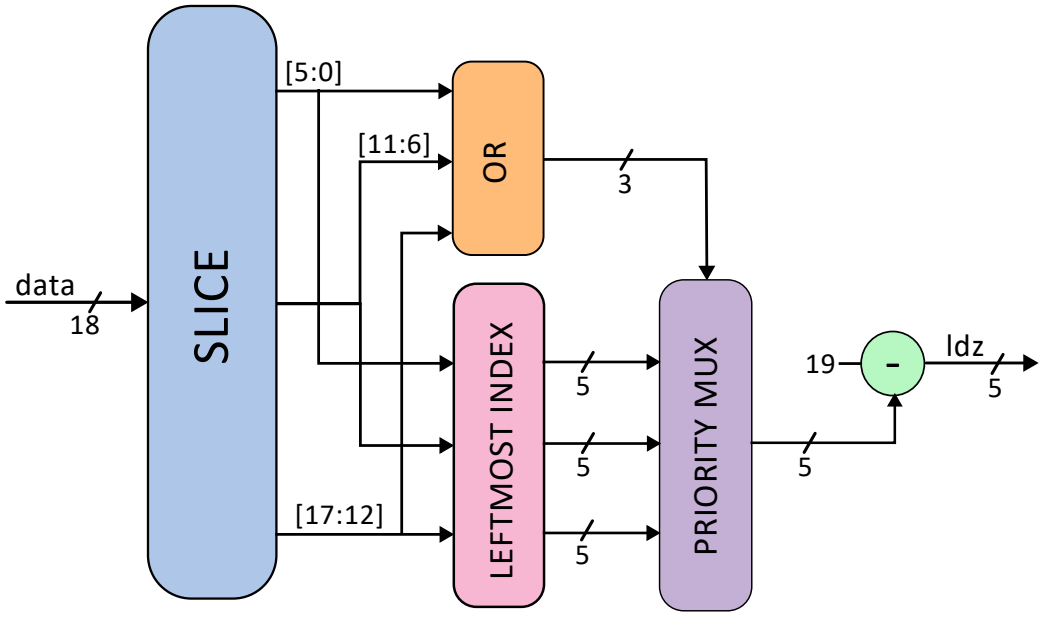


Figura 5.12: Cálculo de ceros por la izquierda

Luego, se comparan los leading zeros con el valor contenido anteriormente en esa posición del sketch y

se mantiene el máximo.

Además, cada vez que entra un flujo, calculamos la suma armónica en paralelo con la inserción. Para esto, le sumamos al registro de acumulación un 1 desplazado hacia la derecha en `max` bits (2^{-max}) y le restamos un 1 desplazado hacia la derecha en `bucket_value` bits ($2^{-bucket\_value}$). Una vez que se activa la señal `output`, el módulo activa la señal `sum_valid` y guarda el resultado de la suma armónica en `harm_sum`.

5.7. Cálculo de parámetros y resolución de consultas

La estimación de parámetros de la distribución power-law y la resolución de los dos tipos de consultas de cuantiles pueden ser implementadas en hardware. Sin embargo, estas requieren operaciones de división y logaritmo, las cuales son complejas y costosas de implementar en hardware. Además, estos procesamientos no se hacen sobre datos en línea con una alta velocidad de llegada, sino sobre datos almacenados en memoria, por lo que no se requiere una velocidad de procesamiento tan alta. Considerando esto, decidimos trasladar este cómputo a un procesador soft-core NEORV32 configurable, el cual puede ser implementado directamente en el FPGA y programado usando el lenguaje C. Esta decisión permite, además, una separación funcional entre el plano de datos y el plano de control, similar al enfoque utilizado en arquitecturas SmartNIC.

La Figura 5.13 presenta un diagrama lógico de la operación del procesador. Al iniciar, el procesador espera a que la señal `sum_valid` se active, lo que indica que puede leer la suma armónica a través de los registros GPIO. Esta suma se usa para calcular la cardinalidad de los flujos usando las fórmulas descritas en la sección Algoritmos. Luego, el procesador espera a que haya datos disponibles para lectura en la interfaz SLINK, simbolizado por `m_valid`. A continuación, se leen y guardan en memoria las frecuencias de los top-K transmitidas a través de SLINK. Estas frecuencias se utilizan para calcular los parámetros C_m y α_b de la distribución power-law. Usando estos parámetros, precomputamos tablas que nos permitan resolver las consultas más rápidamente. En particular, precomputamos dos tablas: Una tabla con los valores de $(x/C_m)^{1/\alpha_b}$ para 511 valores de x y $C_m i^{\alpha_b}$ para 2048 valores de i. Finalmente, entramos a un estado de resolución de consultas, donde se aplican los algoritmos vistos para resolver los dos tipos de consultas de cuantiles, usando las tablas y en el caso de la consulta de frecuencia dado un valor de cuantil, se usa además interpolación. En este contexto, se contemplan dos modalidades:

- Resolver consultas predefinidas (p. ej., frecuencias correspondientes a 10,000 cuantiles equidistantes o consultas de cuantiles para las frecuencias entre 1 y la del flujo más frecuente). Estas consultas se codifican en el código en C del procesador.
- Recibir consultas desde un host, que indica el tipo de consulta y el valor por el cual se desea consultar.

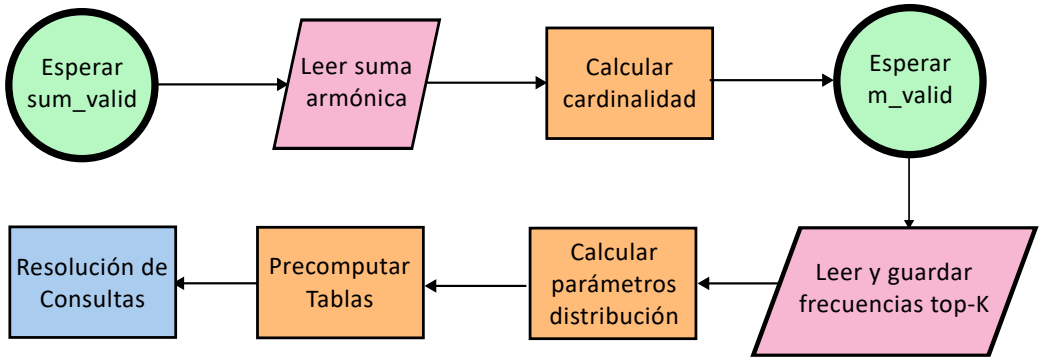


Figura 5.13: Diagrama lógico de la operación del procesador

El procesador NEORV32 utilizado está configurado de la siguiente manera:

- Tiene una frecuencia de reloj de operación de 200 MHz
- Tiene una memoria de instrucciones interna de 192 KB y una memoria de datos interna de 192 KB.
- Soporta los conjuntos de instrucciones **I** (manejo de enteros), **C** (instrucciones comprimidas), **M** (multiplicación de enteros, **Zicntr** (contadores de ciclos), y **Zfinx** (operaciones de punto flotante usando registros enteros).
- Utiliza las interfaces SLINK, GPIO y UART para comunicarse con la lógica programable y el host.

Capítulo 6. Resultados

En este capítulo se presenta la evaluación del desempeño del algoritmo y el acelerador propuestos. En primer lugar, se analiza la precisión del algoritmo en la estimación de los datos de entrada al método, específicamente: la precisión en la estimación de la frecuencia de los top-K (Sección 6.1) y la precisión en la estimación de la cardinalidad de los flujos (Sección 6.2). En la Sección 6.3 se evalúa la precisión del método al estimar la frecuencia del total de los flujos, asumiendo una distribución de tipo power-law. Finalmente, se presentan en la Sección 6.4 los errores obtenidos en las consultas de cuantiles, con el objetivo de evaluar la calidad de dichas estimaciones. Por otro lado, en la Sección 6.5 evaluamos el desempeño del acelerador hardware en cuanto a uso de recursos, frecuencia de operación y latencia.

Para evaluar el desempeño de cada parte del algoritmo, usaremos 9 trazas de 1 minuto disponibles públicamente en el repositorio de CAIDA (Center for Applied Internet Data Analysis) [2]. Estas trazas fueron obtenidas de enlaces de alta velocidad que conectan Equinix con Chicago y New-York. La tabla 6.1 describe las trazas usadas en cuanto a número de flujos (cardinalidad) y su número de paquetes. La cardinalidad de las trazas varía entre 395 mil y 7.3 millones de flujos. Por otro lado, las trazas contienen entre 3 y 84 millones de paquetes.

6.1. Estimación de los top-K

En esta sección evaluaremos la calidad de la detección y estimación de frecuencia de los top-K usando distintas estructuras probabilísticas. Para esta evaluación, consideramos dos métricas. Comenzamos analizando el error relativo promedio (ARE) de la estimación de frecuencia, el cual se calcula como: $\sum_{i=1}^{K} \frac{|f_i - \hat{f}_i|}{f_i}$, donde f_i es la frecuencia real del flujo i y $\hat{f}_i$ es la frecuencia estimada del flujo i. En segundo lugar, consideramos la precisión en la detección de los top-K. La precisión representa cuántos de los flujos reportados por el algoritmo como top-K pertenecen a los top-K reales de la traza. Su fórmula es la siguiente: $P = \frac{TP}{TP+FP}$, donde TP es el número de flujos correctamente identificados como top-K y FP representa el número de flujos reportados como top-K que no pertenecen a la solución real. Para obtener esto, comparamos los valores hash de 32 bits almacenados en la cola de prioridad con los valores hash de los identificadores de flujo de los verdaderos top-K. Evaluamos las estructuras para distintos valores de K entre 1024 y 32768.

Primeramente, comparamos el desempeño de distintos sketches de frecuencia utilizados comúnmente en aceleradores hardware: CountMin-CU [56, 41], Count-Sketch [57, 1], Elastic Sketch [14], TowerCU [16] y nuestra versión modificada del TowerCU. Todos los sketches excepto ElasticSketch usan 1.54 MB de memoria: CountMin-CU y CountSketch usan 12 filas de 2^{15} contadores de 32 bits. TowerCU usa 3 filas de contadores de 8, 16, y 32 bits, donde cada fila usa 2^{22} bits. Nuestro TowerSketch modificado usa 6 filas, donde cada fila usa 2^{21} bits. Elastic Sketch usa una tabla hash de $2 \times K$ entradas de 85 bits y un sketch CountMin-CU de 13 filas con 2^{14} contadores de 32 bits, totalizando 1.51 MB de memoria.

Cada fila de la Tabla 6.2 presenta la desviación estándar y el promedio del ARE de estimación de frecuencia entre las 9 trazas de CAIDA, para distintos valores de K. Podemos observar que, en promedio, nuestro sketch modificado mejora significativamente el ARE con el mismo uso de memoria, comparado con el resto de los sketches. Esta mejora es aún más evidente para valores más altos de K. Cuando K aumenta, el ARE aumenta en todos los sketches. Sin embargo, incluso para $K = 32768$, nuestro sketch estima las frecuencias con un error relativo promedio de 1.22 % en las 9 trazas y menor a 2.05 % en la traza con peor desempeño.

Tabla 6.1: Trazas usadas para evaluar el sistema

Traza	**Flujos**	**Paquetes**
Chicago-20150219-125911	635,775	15,808,577
Chicago-20160121-140200	866,722	31,166,491
Chicago-20080319-200100	395,051	3,895,532
NYC-20181220-125909	2,339,880	29,633,594
NYC-20181220-130100	2,858,270	35,717,638
NYC-20190117-125910	2,426,848	29,492,979
NYC-20190117-130400	2,971,112	37,921,931
NYC-20181220-140100	1,503,466	12,425,462
NYC-20190117-132100	7,338,987	83,104,571

Tabla 6.2: Promedio y desviación estándar del ARE en la estimación de frecuencia de los top-K para distintos sketches

K	**CMCU** [35]		**CS** [33]		**Elastic** [14]		**TowerCU** [16]		**Ours**	
	μ (%)	σ(%)	μ (%)	σ(%)	μ (%)	σ(%)	μ (%)	σ(%)	μ (%)	σ(%)
1K	0.11	0.07	0.25	0.11	1.27	1.23	0.05	0.08	0.01	0.01
2K	0.17	0.08	0.31	0.11	1.92	1.91	0.09	0.10	0.02	0.02
4K	0.39	0.16	0.34	0.12	3.01	2.36	0.22	0.21	0.05	0.03
8K	1.01	0.42	0.98	0.41	5.75	3.37	0.53	0.54	0.11	0.08
16K	3.23	1.51	6.45	1.89	12.82	5.80	1.13	1.04	0.32	0.20
32K	15.86	7.13	35.19	11.60	39.25	18.16	3.60	2.58	1.22	0.73

La Tabla 6.3 muestra la precisión promedio para las 9 trazas y su desviación estándar, para distintos valores de K y distintos sketches. Podemos observar que todos los sketches obtienen una alta precisión para valores de K menores a 8192. Sin embargo, la precisión de la mayoría se degrada rápidamente al aumentar el valor de K a 16384. Esta degradación es aún más importante para $K = 32768$, como podemos apreciar en la última fila de la tabla. TowerCU mantiene una precisión mayor a 0.95 para todos los valores de K. Sin embargo, al modificar este sketch como describimos anteriormente, podemos aumentar la precisión a más de 0.99 en promedio para todos los valores de K. Para la traza con peor desempeño, se identifican los top-32768 con una precisión de 0.98.

Considerando lo expuesto anteriormente, concluimos que la modificación al sketch propuesta mejora tanto la estimación de frecuencia de los top-K como la precisión en la identificación de estos. Esta mejora se consigue a través de una redistribución de la memoria disponible y no afecta la complejidad de la actualización ni la consulta del sketch más allá de requerir el ordenamiento de 6 elementos en lugar de 3.

Posteriormente, evaluamos el desempeño del PQA al capturar las frecuencias más grandes entregadas por nuestro TowerSketch modificado. Para distintos valores de K comparamos el desempeño de una cola de prioridad perfecta (PPQ) de K elementos con el desempeño de un arreglo de colas de prioridad de $K/4$ colas de 4 elementos (PQA4) y el desempeño del arreglo de colas de prioridad con $K/4$ colas de 6 elementos propuesto

Tabla 6.3: Promedio y desviación estándar de la precisión en la identificación de top-K para distintos sketches

K	**CMCU** [35]		**CS** [33]		**Elastic** [14]		**TowerCU** [16]		**Ours**	
	μ	σ	μ	σ	μ	σ	μ	σ	μ	σ
1K	1.00	0.004	0.99	0.009	0.98	0.018	1.00	0.003	1.00	0.000
2K	1.00	0.003	0.98	0.019	0.97	0.021	1.00	0.003	1.00	0.000
4K	0.99	0.005	0.98	0.015	0.97	0.022	1.00	0.006	1.00	0.000
8K	0.99	0.006	0.95	0.019	0.95	0.029	0.99	0.012	1.00	0.000
16K	0.97	0.013	0.85	0.035	0.91	0.037	0.98	0.019	1.00	0.004
32K	0.87	0.041	0.65	0.067	0.81	0.047	0.96	0.031	0.99	0.007

Tabla 6.4: Promedio y desviación estándar del ARE en la estimación de frecuencia de los top-K para distintas colas de prioridad

K	PPQ		PQA4 [1]		PQA6 (Ours)	
	μ (%)	σ (%)	μ (%)	σ (%)	μ (%)	σ (%)
1K	0.01	0.01	7.20	1.26	0.71	0.29
2K	0.02	0.02	8.32	1.56	0.78	0.17
4K	0.05	0.03	10.63	1.67	1.14	0.26
8K	0.11	0.08	11.55	1.69	1.28	0.26
16K	0.32	0.20	12.75	1.85	1.28	0.33
32K	1.22	0.73	12.06	0.81	0.88	0.23

anteriormente (PQA6), donde se ordena el contenido al final del procesamiento y se escogen los K elementos con mayor frecuencia.

La Tabla 6.4 muestra el ARE promedio de 9 trazas obtenido por las colas de prioridad para distintos valores de K. El ARE obtenido por el PQA4 aumenta significativamente con respecto a la PPQ para todos los valores de K. Dado que las estimaciones del sketch tienen un ARE bajo, concluimos que las colisiones de la función hash tienen un alto impacto en el ARE de las estimaciones. Al insertar 2 elementos extra por cola del arreglo, disminuimos el efecto de estas colisiones, mejorando el ARE considerablemente. El ARE obtenido por el PQA6 es de K y oscila entre 0.71 % y 1.28 %. En la traza con el peor desempeño, el ARE es menor a 1.96 %.

Por otro lado, la Tabla 6.5 muestra la precisión obtenida por las distintas colas de prioridad. Podemos observar que la cola de prioridad perfecta alcanza precisiones de 0.99 o más para todos los valores de K. Esta precisión disminuye a 0.81 al usar el PQA4 debido a las colisiones en la función hash que causan pérdida de elementos pertenecientes a los top-K. Al agregar dos elementos a cada cola en el PQA6, la precisión aumenta de manera notable, alcanzando una precisión de 0.95-0.96, donde la peor traza alcanza una precisión de 0.94.

Estos resultados nos permiten concluir que el PQA6 aproxima satisfactoriamente el comportamiento de una cola de prioridad perfecta, a diferencia del PQA4 que es más sensible a las colisiones de la función hash. El uso del PQA6 introduce un costo en memoria del 50 % extra en esa sección del acelerador, además de introducir la necesidad de ordenar su contenido al final del intervalo de información. Sin embargo, el PQA6 aproxima significativamente mejor el comportamiento de una PPQ que el PQA4 y soporta además la inserción de un

Tabla 6.5: Promedio y desviación estándar de la precisión en la identificación de top-K para distintas colas de prioridad

K	PPQ		PQA4 [1]		PQA6 (Ours)	
	μ	σ	μ	σ	μ	σ
1K	1.00	0.00	0.81	0.01	0.96	0.01
2K	1.00	0.00	0.81	0.01	0.95	0.00
4K	1.00	0.00	0.81	0.01	0.95	0.00
8K	1.00	0.00	0.81	0.00	0.95	0.00
16K	1.00	0.00	0.81	0.00	0.95	0.00
32K	0.99	0.01	0.81	0.00	0.96	0.01

paquete por ciclo de reloj a altas frecuencias, lo cual es imposible de conseguir con una cola de prioridad perfecta.

6.2. Estimación de cardinalidad

La cardinalidad juega un rol fundamental en ambas consultas de cuantiles, como se aprecia en las expresiones presentadas en la Sección 4.8. Debido a esto, es necesario evaluar la estimación de cardinalidad entregada por el sketch HyperLogLog.

La Tabla 6.6 muestra la cardinalidad real de cada traza. Además, presenta la cardinalidad estimada por el sketch y los errores absolutos y relativos de esta estimación para distintos valores de p. Podemos observar que todos los valores de p otorgan estimaciones de cardinalidad con un error relativo máximo de 2.55 %. Sin embargo, para obtener un error relativo máximo menor a 2 % y un error relativo promedio menor al 1 %, decidimos utilizar un HyperLogLog con $p = 13$, valor que además ha dado buenos resultados en estimaciones de entropía [1]. Aunque un sketch con $p = 14$ entrega mejores estimaciones de cardinalidad en promedio, la disminución es pequeña, con 0.21 % en el error relativo promedio y 0.44 % en el error relativo máximo. De esa manera, el sketch HyperLogLog usado contiene 2^{13} posiciones de 5 bits (el valor para el que se determinan los ceros a la izquierda es de $32 - 13 = 19$ bits), ocupando un total de memoria de 5.1 KB.

6.3. Estimación de frecuencia

Una vez que verificamos que la frecuencia de los top-K y la cardinalidad están siendo estimadas satisfactoriamente, es pertinente evaluar la calidad de la estimación de frecuencia de la totalidad de los flujos. Esta estimación se obtiene bajo la hipótesis de que las frecuencias de los flujos no pertenecientes al conjunto top-K siguen una distribución de tipo power-law hasta alcanzar una frecuencia unitaria. Para calcular los parámetros de esta distribución, usamos la mitad inferior de los $K = 32768$ flujos más frecuentes.

La Figura 6.1 presenta dos gráficos por cada traza evaluada. El primer gráfico compara la distribución real de frecuencias (en azul) con dos distribuciones estimadas por el método: una utilizando las frecuencias reales

Tabla 6.6: Cardinalidad real y estimada por el sketch HyperLogLog con distintos valores de p

Traza	Real	$p = 12$			$p = 13$			$p = 14$		
		Estimada	E.Abs	E.Rel	Estimada	E.Abs	E.Rel	Estimada	E.Abs	E.Rel
Chicago-20150219-125911	635.7K	642.6K	6829	1.07 %	634.3K	1502	0.24 %	634.4K	1436	0.23 %
Chicago-20160121-140200	866.7K	857.1K	9650	1.11 %	875.3K	8551	0.99 %	867.7K	932	0.11 %
Chicago-20080319-200100	395.1K	397.6K	2529	0.64 %	395.0K	92	0.02 %	394.9K	108	0.03 %
NYC-20181220-125909	2.340M	2.323M	16531	0.71 %	2.338M	1883	0.08 %	2.315M	24745	1.06 %
NYC-20181220-130100	2.858M	2.849M	8997	0.31 %	2.856M	2491	0.09 %	2.858M	515	0.02 %
NYC-20190117-125910	2.427M	2.369M	58338	2.40 %	2.389M	38280	1.58 %	2.414M	12572	0.52 %
NYC-20190117-130400	2.971M	2.928M	42888	1.44 %	2.931M	40211	1.35 %	2.949M	22370	0.75 %
NYC-20181220-140100	1.503M	1.501M	2475	0.16 %	1.488M	15582	1.04 %	1.486M	17120	1.14 %
NYC-20190117-132100	7.339M	7.152M	187067	2.55 %	7.253M	85551	1.17 %	7.275M	64075	0.87 %
Error Relativo Promedio				1.16 %			0.73 %			0.52 %

de los top-K y la cardinalidad real (en morado), y otra empleando las frecuencias estimadas por el TowerSketch y el PQA y la cardinalidad estimada mediante HyperLogLog (en verde). El segundo gráfico muestra el error absoluto por flujo en la estimación de frecuencia, junto al valor promedio de dicho error.

Al observar el primer gráfico de cada figura, podemos notar que la distribución de la frecuencia estimada por el método con entradas perfectas es muy similar a la obtenida usando los sketches y el PQA. Si observamos más detalladamente los gráficos de error absoluto, podemos observar un patrón que se repite en las distintas trazas. La frecuencia de los primeros diez top-K presenta un error casi nulo, mientras que para los siguientes elementos del top-K, el error aumenta debido a las limitaciones del sketch y del PQA. Esto es especialmente notorio en la última traza (Figura 6.1.i), que posee mayor cardinalidad y, por ende, más colisiones en las funciones hash. Por otra parte, la frecuencia de los primeros flujos no pertenecientes a los top-K está bien aproximada por la power-law, como lo demuestra el bajo error absoluto en esta sección. Para los flujos finales, el error absoluto de estimación aumenta ligeramente. Este comportamiento se explica por dos factores: (i) la distribución de frecuencia en ese rango no sigue estrictamente una power-law como habíamos supuesto, y (ii) en algunas trazas, el punto estimado L se encuentra significativamente a la izquierda del valor real, lo que provoca que la frecuencia estimada decrezca más rápidamente que en la distribución real.

A pesar de estas limitaciones, se obtienen errores absolutos promedio entre 0.4 y 1.8 cuentas y errores absolutos máximos de hasta 43. Además, en las primeras ocho trazas, los errores absolutos superiores a 10 cuentas corresponden a errores relativos inferiores al 0.7 %. En la última traza, cuya estimación se ve afectada por la gran cantidad de flujos, todos los errores absolutos mayores a 10 cuentas están asociados a errores relativos inferiores al 5.3 %.

6.4. Estimación de cuantiles

Finalmente, para terminar con la evaluación del algoritmo, debemos evaluar el desempeño de éste al responder las dos consultas de cuantiles descritas anteriormente. Esta evaluación permite determinar en qué medida los errores en la estimación de la frecuencia de los flujos (tanto de los top-K como del resto) y en la estimación de la cardinalidad impactan en la precisión de las respuestas entregadas por el método.

6.4.1. Consulta de cuantil dado un valor de frecuencia

Para evaluar la capacidad del algoritmo de estimar con precisión el cuantil en que se encuentra un cierto valor de frecuencia, consultamos por el cuantil correspondiente para cada frecuencia entre 1 y la frecuencia del top-1 de cada traza. De esta manera, obtenemos todas las posibles respuestas a las consultas de cuantiles dada una frecuencia para cada traza.

La Figura 6.2 presenta los resultados de esta evaluación con dos gráficos por cada traza de evaluación. Para cada frecuencia consultada (representada en escala logarítmica en el eje x), el primer gráfico muestra el valor real del cuantil en el que se encuentra esta frecuencia y el valor estimado por el algoritmo. La línea intermitente de este gráfico representa el cuantil a partir del cual las frecuencias se encuentran dentro de los top-K. Adicionalmente, el segundo gráfico presenta el error absoluto en la estimación (en porcentaje) y el valor promedio de este error. Se observa una diferencia notable entre el cuantil real y el estimado para frecuencias menores a 20. Esta disparidad se atribuye a los errores de estimación de frecuencia discutidos en la Sección 6.3. En particular, el error en la estimación del punto L provoca que el cuantil estimado para la frecuencia 1 presente un error absoluto superior al 10 %. A partir de la frecuencia 2, los errores decrecen consistentemente: se reduce a 1 % o menos a partir de la frecuencia 20 y a 0.1 % o menos a partir de la frecuencia 30 en la mayoría de las trazas. Para la traza Chicago-20160121-140200, el error baja de 1 % cerca de la frecuencia 30 y baja de 0.1 % a partir de la frecuencia 90. En términos generales, se obtienen errores absolutos promedio entre 0.0004 % y 0.0122 %, lo cual es considerablemente menor que el error promedio de 0.5 % mencionado en el objetivo general.

6.4.2. Consulta de frecuencia dado un valor de cuantil

Ahora evaluamos la operación inversa: dado un valor de cuantil, determinar la frecuencia correspondiente. Para esta evaluación, se realizaron consultas para 10,000 cuantiles equi-espaciados en el intervalo $[0, 1]$, es decir, con una separación de 0.0001 entre consultas consecutivas.

La Figura 6.3 muestra los valores reales (en azul) y estimados (en celeste) de frecuencia de cada cuantil para cada traza, así como también el valor absoluto del error y su valor promedio. A pesar de observarse discrepancias en las frecuencias estimadas para cuantiles entre 0.6 y 0.9 (atribuidas a las limitaciones del método discutidas previamente) los errores absolutos en este rango se mantienen por debajo de las 10 cuentas en todas las trazas.

Los errores máximos se encuentran en la sección estimada por los top-K almacenados. Este error elevado se explica por dos factores: (i) Primeramente, el TowerSketch sobreestima algunas de las frecuencias de los top-K. Sin embargo, esta sobreestimación es ligera, como mostramos en la Sección 6.3. (ii) El efecto más considerable sobre este error lo tiene la estimación de cardinalidad. Recordando, el rank estimado correspondiente a un cuantil q se calcula como $\hat{R} = \lceil q\hat{N} \rceil$, por lo que cualquier imprecisión en la estimación de $\hat{N}$ puede causar que el flujo consultado en los top-K difiera del flujo real que se encuentra en ese cuantil. Dado que en los flujos más frecuentes existe una diferencia de frecuencia grande entre elementos consecutivos, esta diferencia en la estimación de rank puede causar un error absoluto máximo elevado. Aún considerando esto, obtenemos un error absoluto máximo de estimación de 288 cuentas. Si bien el valor absoluto del error es elevado, este error corresponde a solo un 0.33 % de error relativo. Todos los errores absolutos mayores a 10 están relacionados con errores relativos de 3.22 % o menos.

Adicionalmente, considerando el panorama general de la estimación, podemos observar que los errores

absolutos promedio varían entre 0.43 y 2.09 cuentas, lo que nos permite afirmar que, a pesar de las limitaciones descritas anteriormente,logra aproximar las frecuencias correspondientes a cuantiles con un nivel de precisión satisfactorio.

6.5. Evaluación del acelerador

Completada la evaluación del algoritmo, corresponde ahora evaluar el desempeño del acelerador hardware que implementa este algoritmo. Diseñamos el acelerador a nivel RTL (Register-Transfer Level) usando SystemVerilog. El acelerador fue implementado usando Vivado 2022.2 en una tarjeta AMD Alveo U280, que contiene un FPGA XCU280 basado en la arquitectura UltraScale+. Esta tarjeta fue seleccionada debido a su compatibilidad con NetFPGA PLUS, un framework de código libre para dispositivos de red inteligentes.

La Tabla 6.7 presenta el uso de recursos de cada módulo del sistema, además de la utilización total de recursos y el porcentaje del total de recursos disponibles que representa. Podemos ver que el mayor uso de recursos se concentra en 3 módulos: (i) El TowerSketch, el cual usa 48 UltraRAMs para almacenar las 6 filas del sketch, 180 DSP usados en el cálculo de las funciones hash, 6770 LUTs y 9870 Flip-Flops usados para implementar lógica, memoria distribuida y registros de pipeline. (ii) El módulo PQA que usa 12 UltraRAMs para almacenar las 8192 colas de prioridad de 6 elementos y 1318 LUTs y 1407 Flip-Flops para implementar lógica, memoria distribuida y registros de pipeline. (iii) Los 6 módulos RadixSort que usan 24 UltraRAMs para almacenar el doble buffer de cada módulo. Además, los módulos de RadixSort usan 5454 LUTs para lógica y memoria distribuida y 1387 Flip-Flops para registros de pipeline. Por otro lado, el sketch HyperLogLog y el módulo MergeSort tienen usos de recursos bastante más bajos, usando 1.5 y 45 BlockRAMs respectivamente. El procesador soft-core NEORV32 utiliza 2998 LUTs, 2025 Flip-Flops, 151 BlockRAMs y 2 DSP. Finalmente, se usan FIFOs de sincronización para manejar el cruce de dominios de reloj, las cuales usan 54 BlockRAMs. Podemos observar que el sistema ocupa menos de un 9 % de las UltraRAMs disponibles y menos del 13 % de las BlockRAMs disponibles. Además, ocupa menos de un 2.4 % de las LUTs, los Flip-Flops y los DSP disponibles. Esto nos permite afirmar que este sistema se podría integrar fácilmente con hardware de procesamiento adicional en esta plataforma.

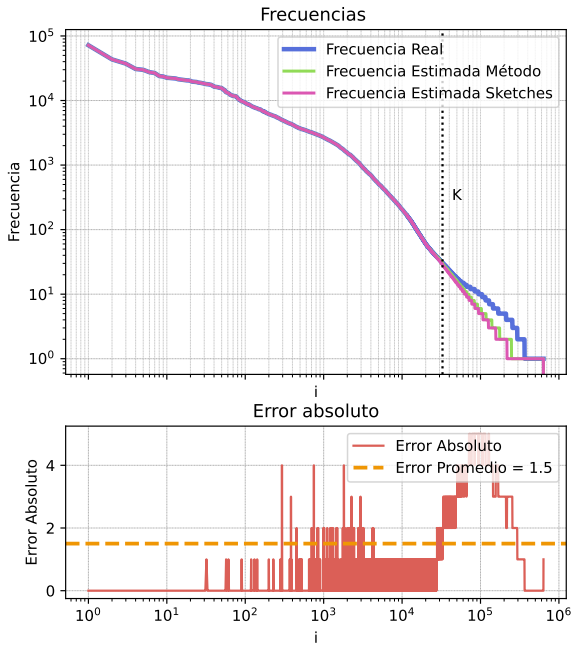
Tabla 6.7: Uso de recursos del acelerador

Recurso	LUT	FF	URAM	BRAM	DSP
TowerSketch	6770	9870	48	0	180
PQA	1318	1407	12	0	0
HyperLogLog	567	987	0	1.5	30
RadixSort	5454	1387	24	0	0
MergeSort	585	436	0	45	0
Procesador	2998	2025	0	151	2
FIFOs	4480	8443	0	54	0
Total	18402	17605	84	251.5	212
Porcentaje de uso (%)	1.41	0.68	8.75	12.48	2.35

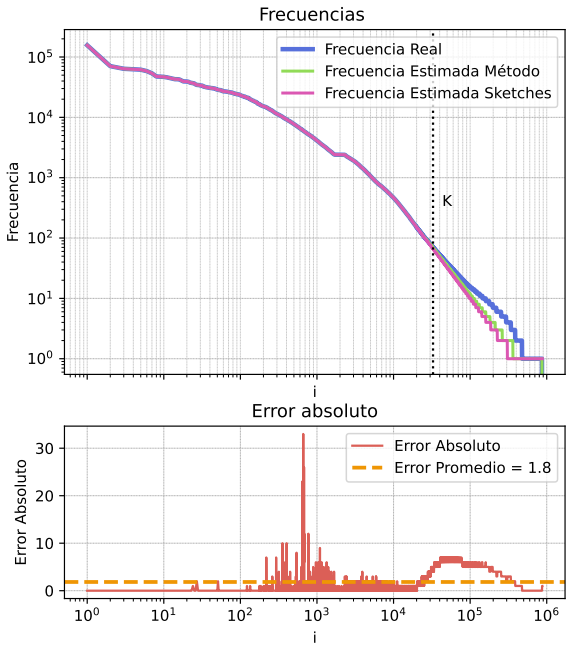
El acelerador trabaja con 3 dominios de reloj. El TowerSketch, el PQA y el HyperLogLog soportan una frecuencia máxima de reloj de 392 MHz y pueden recibir un paquete por ciclo de reloj. Esto nos permite alcanzar una velocidad de recepción de datos mínima de 200 Gbps, obtenida en el peor caso donde todos los paquetes son de tamaño mínimo (64 bytes). Sin embargo, esto no es un caso realista. En nuestras trazas, el tamaño de paquete promedio por traza es mayor a 435 bytes, lo que permitiría al acelerador operar a 1300 Gbps usando un buffer que compense las diferencias instantáneas.

Por otro lado, el ordenamiento (RadixSort y MergeSort) alcanza una frecuencia de reloj máxima de 250 MHz, lo cual nos entrega una latencia de ordenamiento de 1 ms. Sin embargo, como este ordenamiento se realiza solo una vez antes de la etapa de recepción de consultas, no afecta directamente la rapidez de estas. La sincronización entre los módulos con distintos dominios de reloj se realiza usando memorias FIFO (First In First Out).

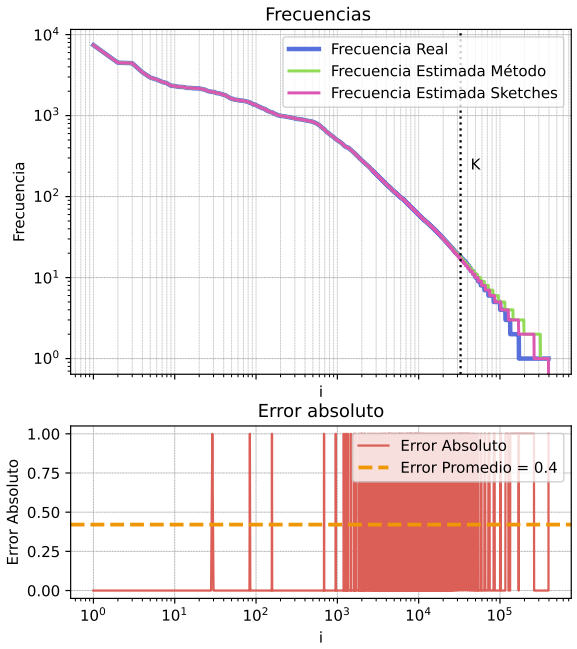
El procesador NEORV32 trabaja a 200 MHz. El tiempo de recepción de los top-K es de 6 ms, mientras que el cómputo de los parámetros de la distribución toma alrededor de 14 segundos. Este tiempo está dominado casi completamente por el cálculo de los logaritmos necesarios para calcular α_b. En precomputar las tablas, el procesador se demora alrededor de 200 ms. Una vez transcurridos estos tiempos iniciales, el procesador puede recibir consultas y responderlas con una latencia máxima de 38.5 μs para consultas de frecuencia dado un valor de cuantil y 4.2 μs para consultas de cuantil dado un valor de frecuencia.



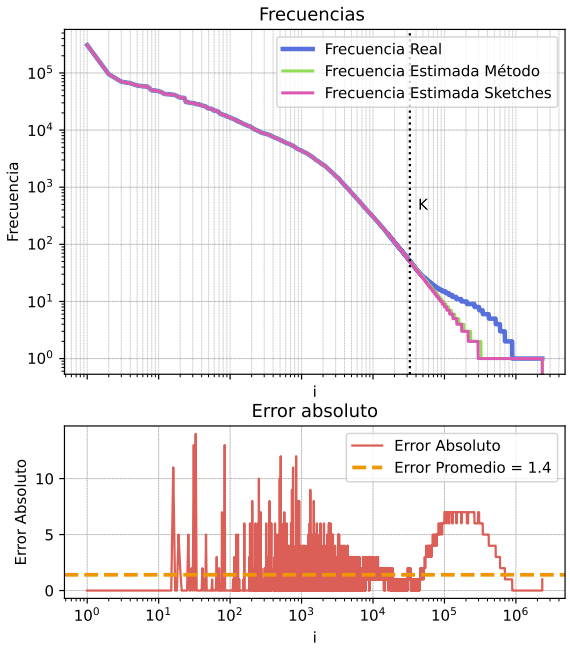
(a) Chicago-20150219-125911



(b) Chicago-20160121-140200

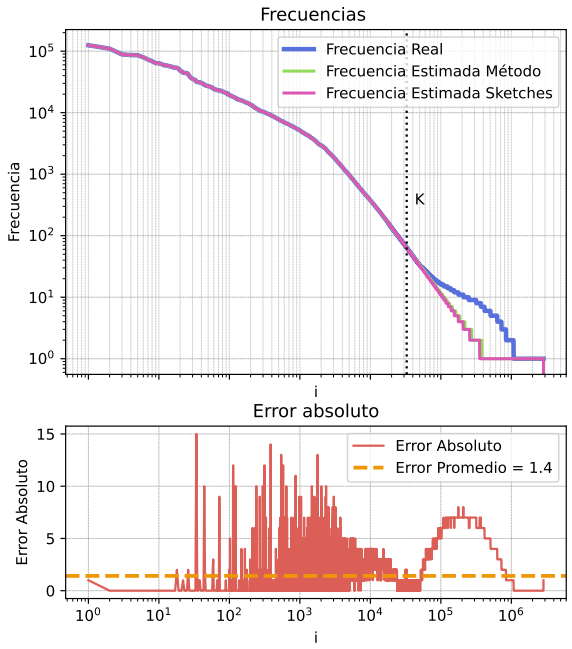


(c) Chicago-20080319-200100

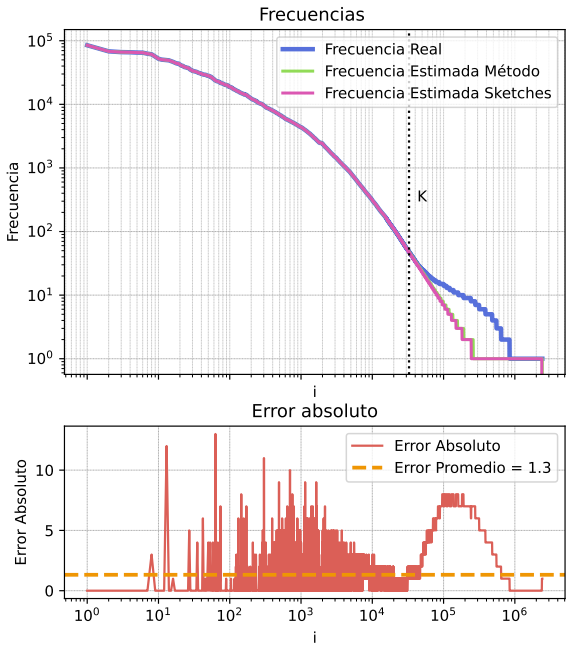


(d) NYC-20181220-125909

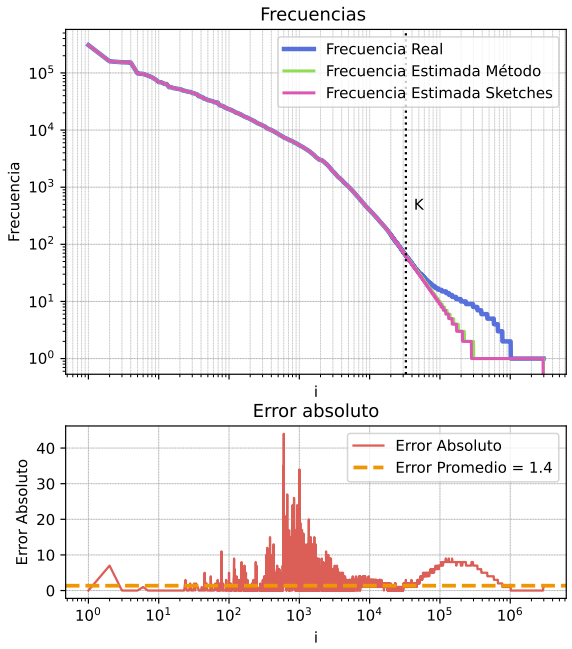
Figura 6.1: Resultados estimación de frecuencia (1/3)



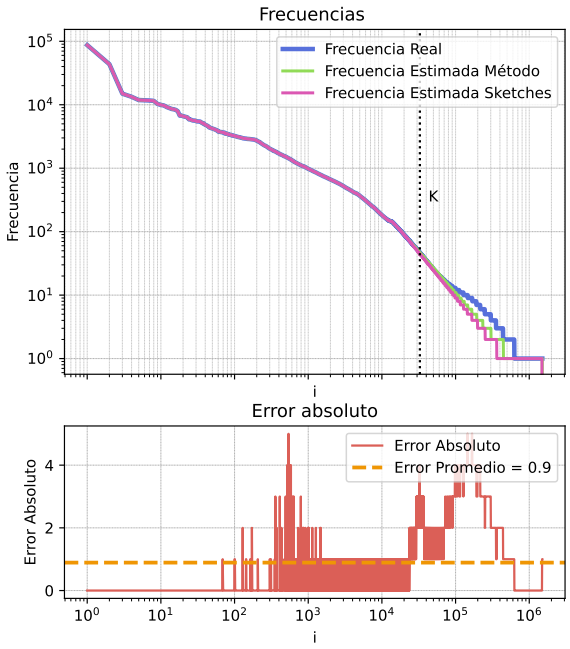
(e) NYC-20181220-130100



(f) NYC-20190117-125910

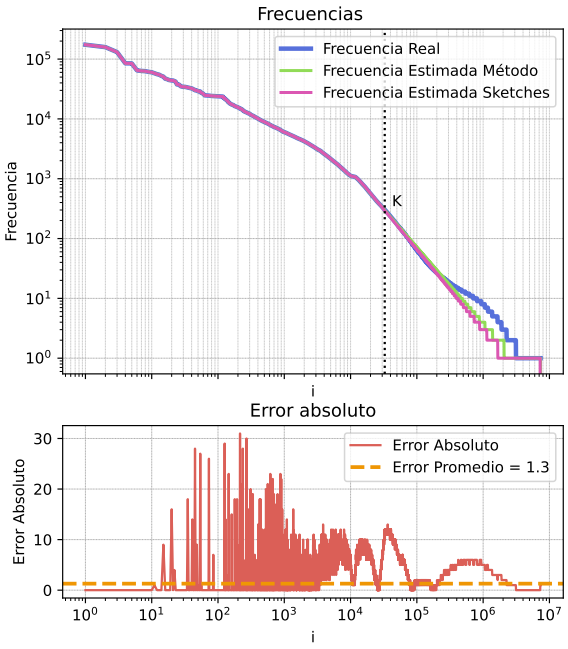


(g) NYC-20190117-130400



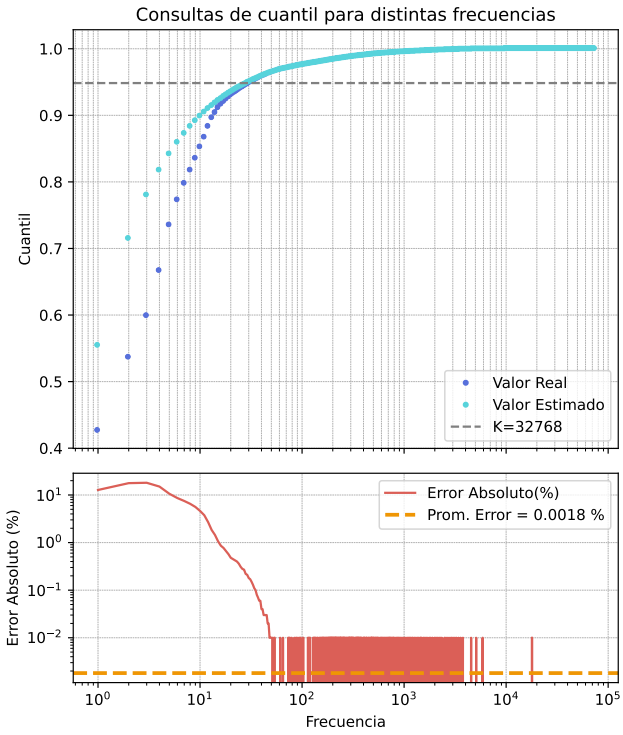
(h) NYC-20181220-140100

Figura 6.1: Resultados estimación de frecuencia (2/3)

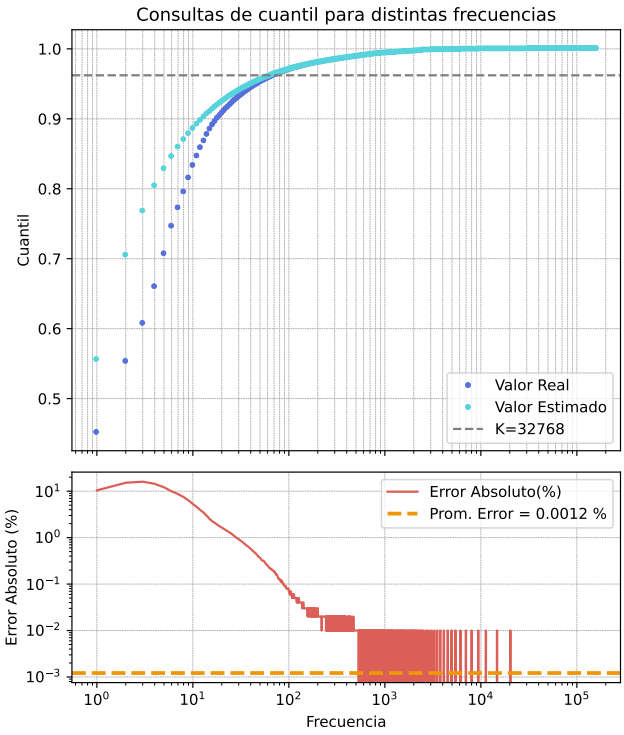


(i) NYC-20190117-132100

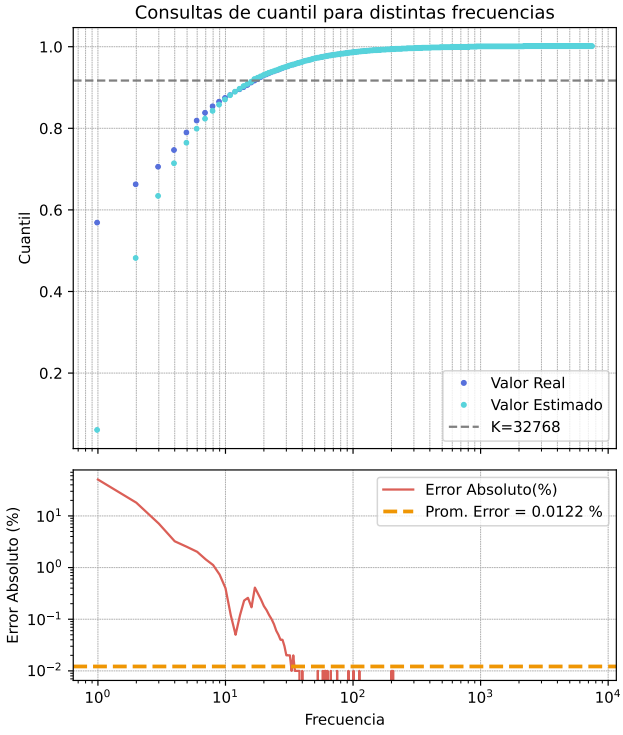
Figura 6.1: Resultados estimación de frecuencia (3/3).



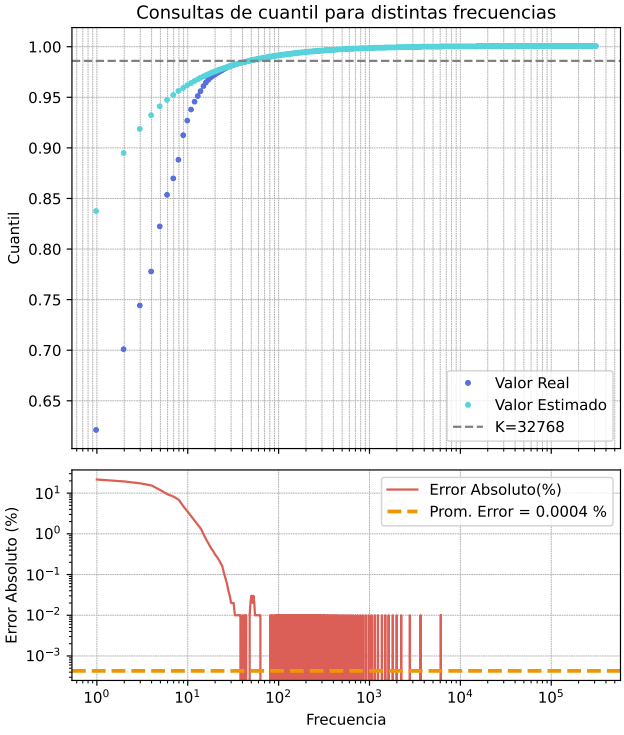
(a) Chicago-20150219-125911



(b) Chicago-20160121-140200

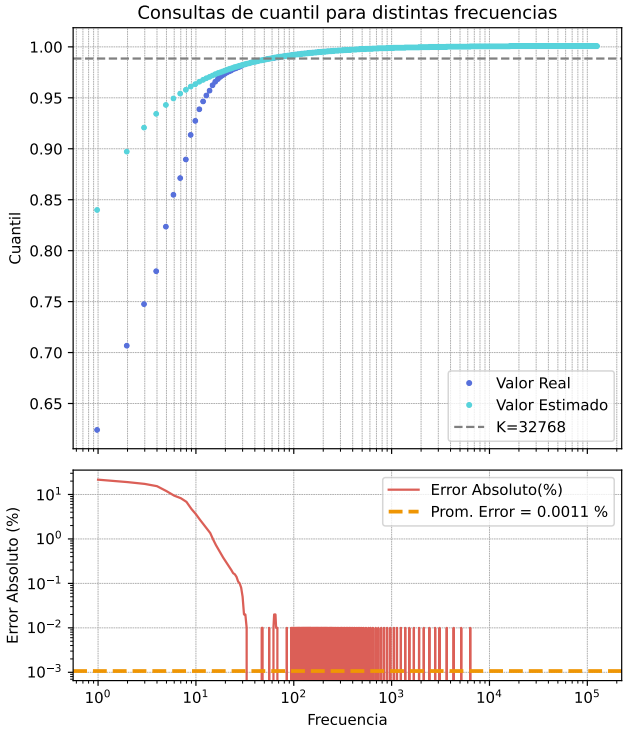


(c) Chicago-20080319-200100

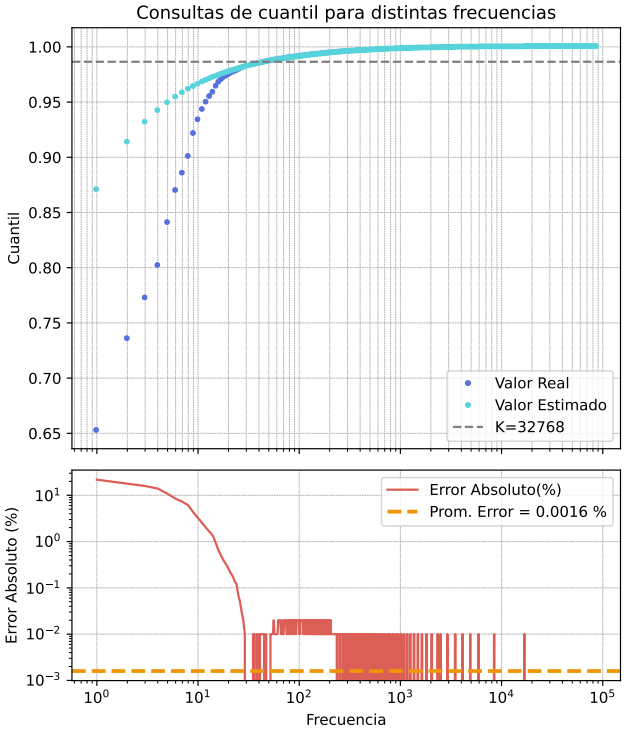


(d) NYC-20181220-125909

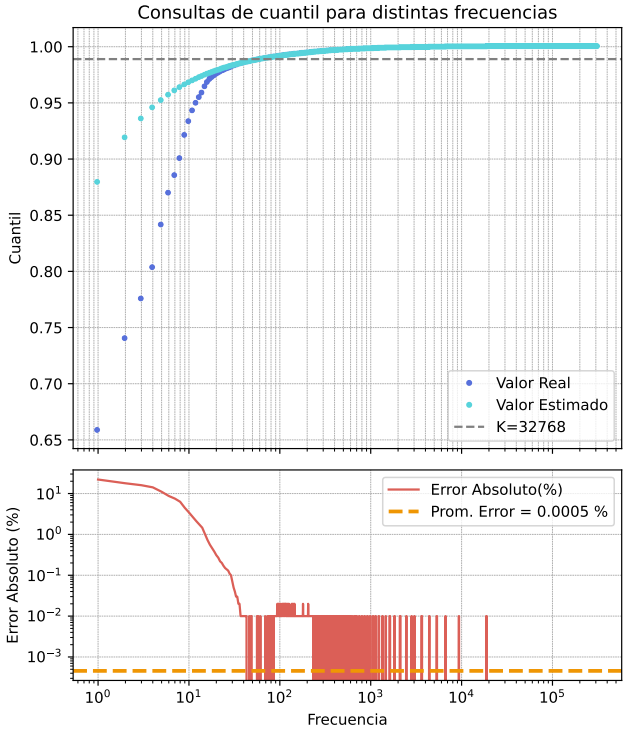
Figura 6.2: Resultados de consultas de cuantil de frecuencia (1/3)



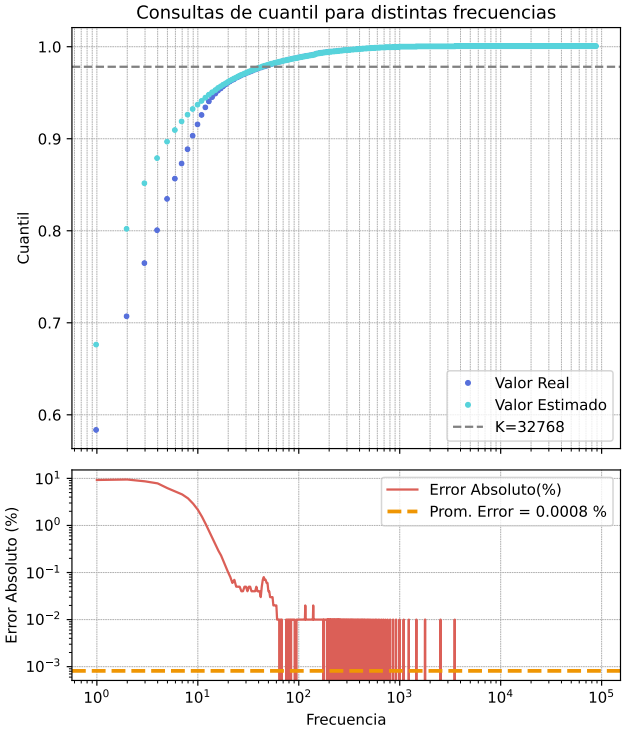
(e) NYC-20181220-130100



(f) NYC-20190117-125910

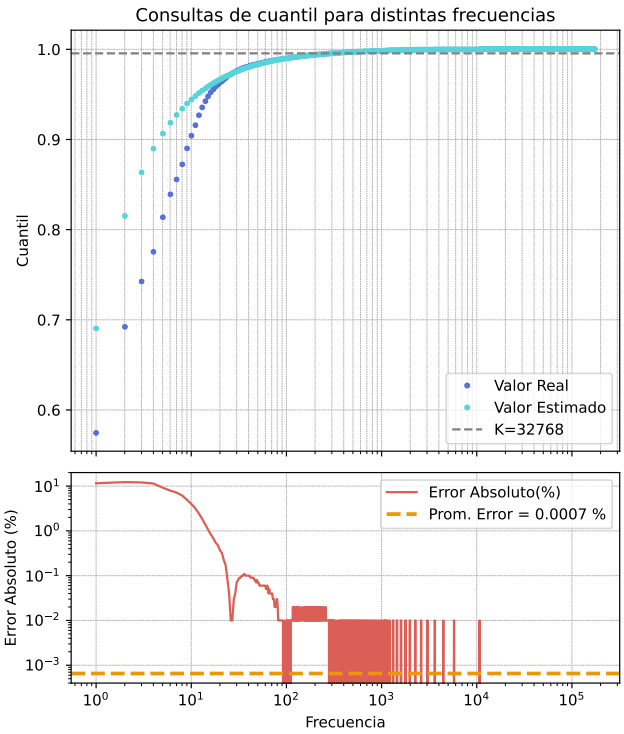


(g) NYC-20190117-130400



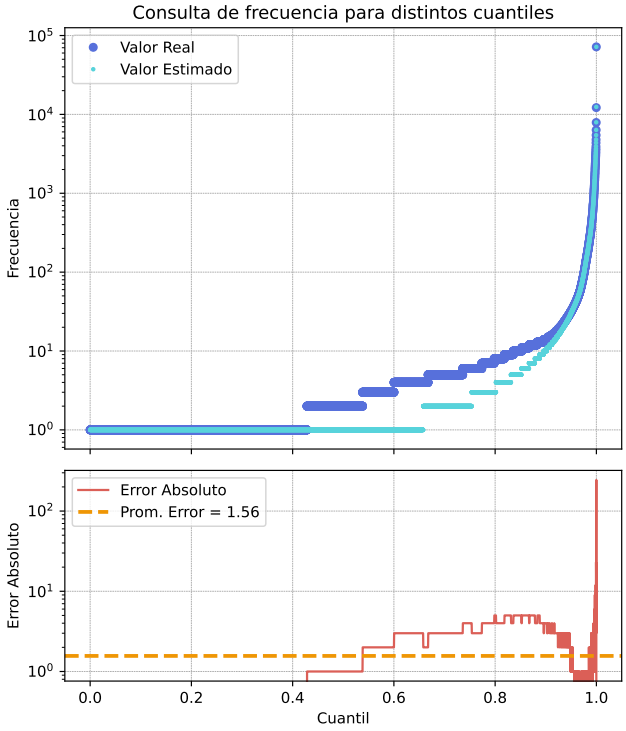
(h) NYC-20181220-140100

Figura 6.2: Resultados de consultas de cuantil de frecuencia (2/3)

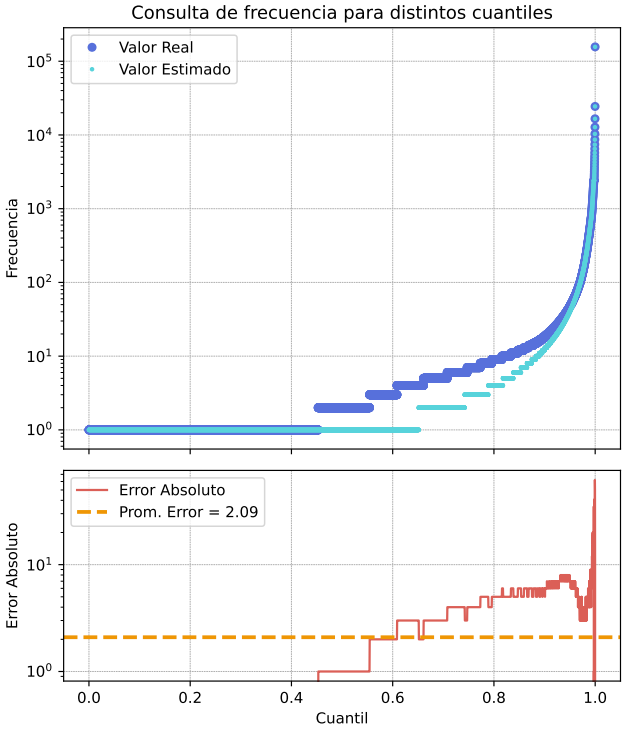


(i) NYC-20190117-132100

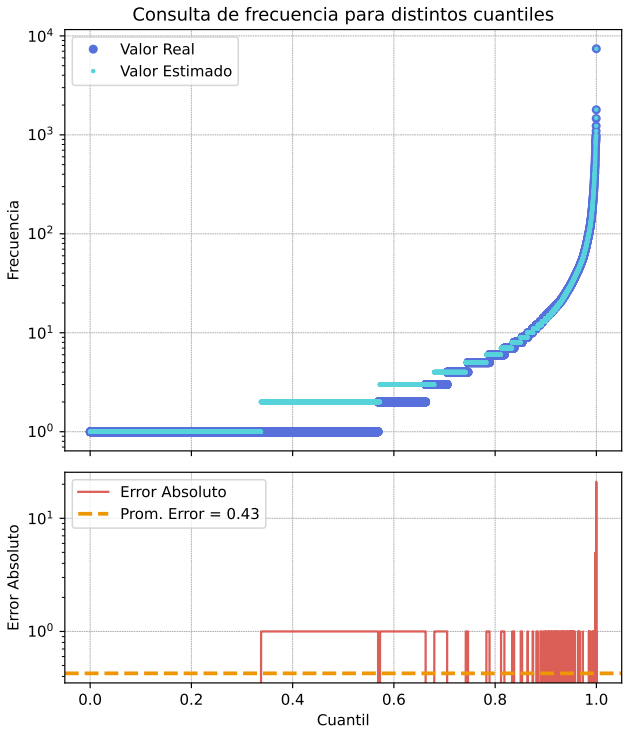
Figura 6.2: Resultados de consultas de cuantil de frecuencia (3/3).



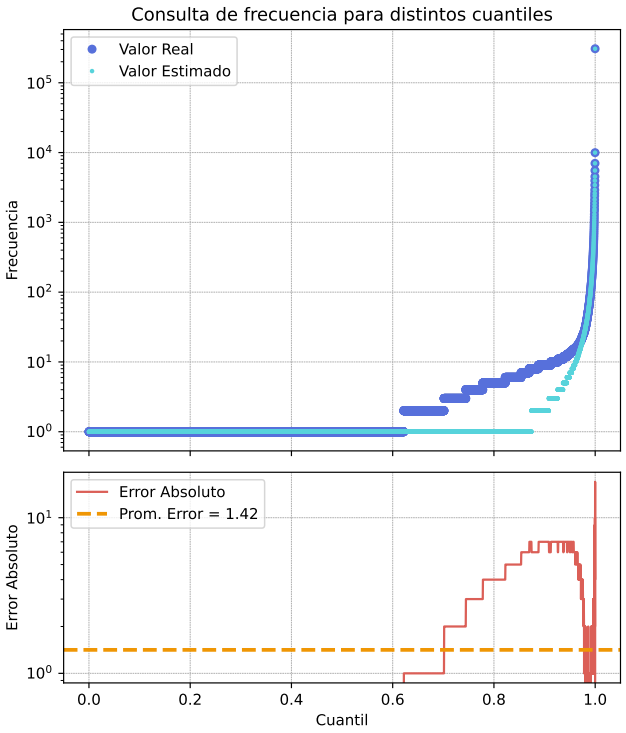
(a) Chicago-20150219-125911



(b) Chicago-20160121-140200

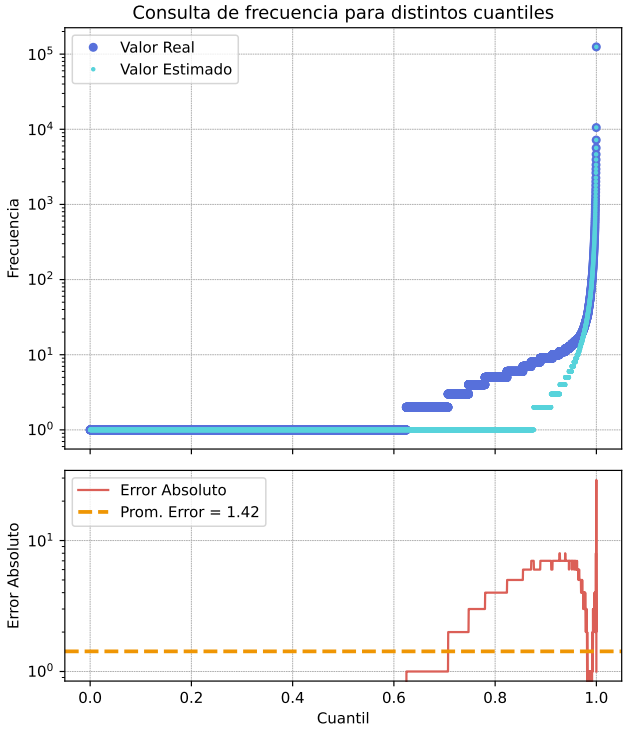


(c) Chicago-20080319-200100

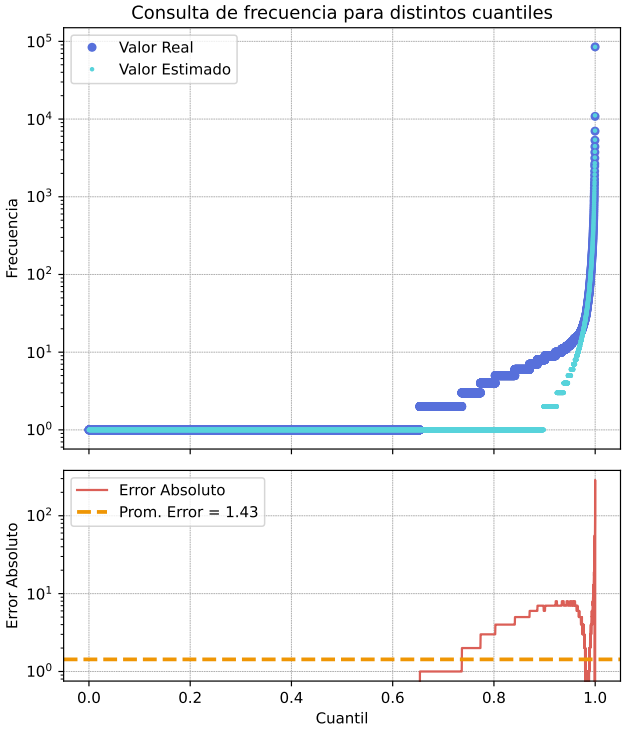


(d) NYC-20181220-125909

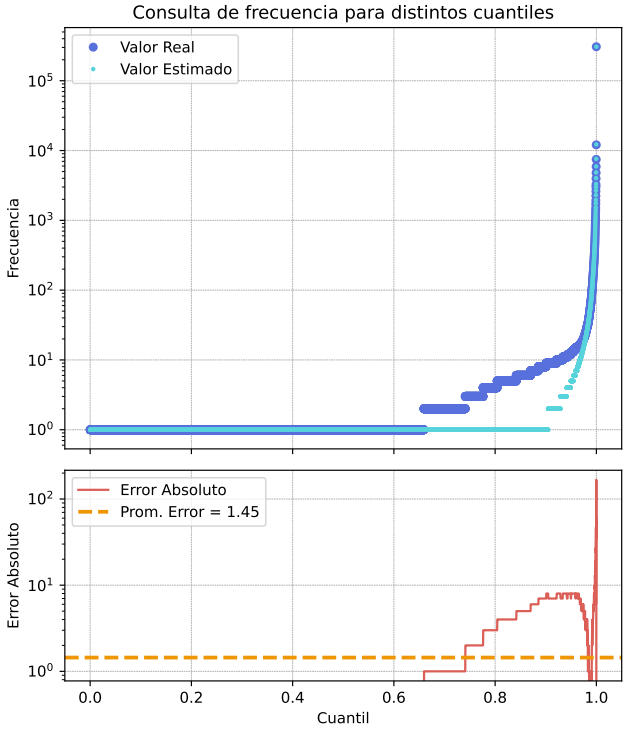
Figura 6.3: Resultados de consultas de frecuencia de cuantiles (1/3)



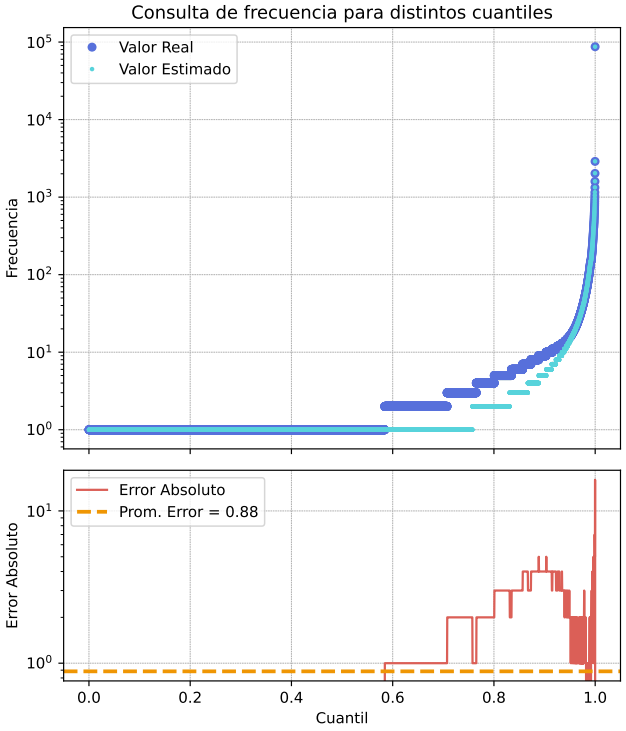
(e) NYC-20181220-130100



(f) NYC-20190117-125910

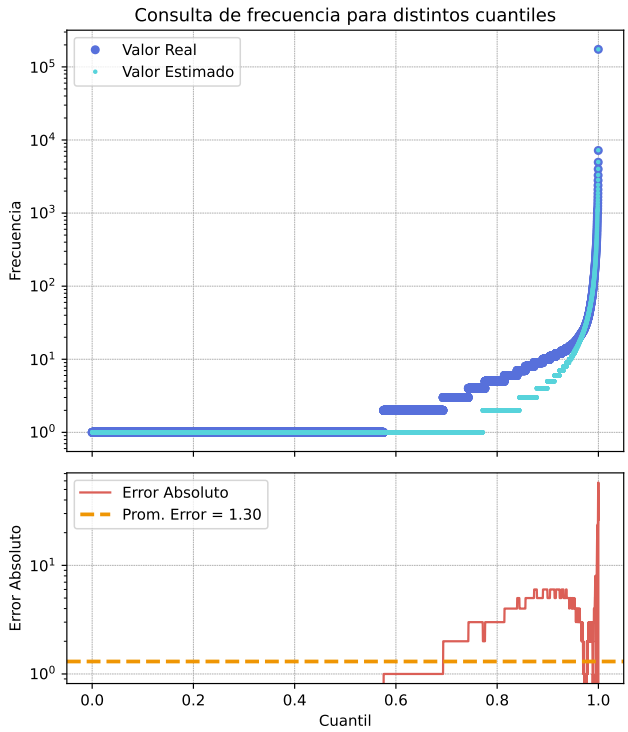


(g) NYC-20190117-130400



(h) NYC-20181220-140100

Figura 6.3: Resultados de consultas de frecuencia de cuantiles (2/3)



(i) NYC-20190117-132100

Figura 6.3: Resultados de consultas de frecuencia de cuantiles (3/3).

Conclusión

En este trabajo, presentamos un algoritmo para la estimación de cuantiles de frecuencia en redes de datos. El algoritmo propuesto usa un TowerSketch para estimar la frecuencia de los flujos y un arreglo de colas de prioridad para almacenar las frecuencias de los flujos top-K. Estas frecuencias se usan a su vez para calcular los parámetros de una distribución estadística de tipo power-law que aproxima la frecuencia del resto de los flujos. Usando esta aproximación y las frecuencias estimadas para los top-K, se determina el cuantil en el que se encuentra una frecuencia o la frecuencia a la que corresponde un cuantil. El algoritmo es capaz de responder de manera precisa ambos tipos de consulta, con un error absoluto promedio entre 0.0004 % y 0.0122 % para consultas de cuantil dado un valor de frecuencia y un error absoluto promedio entre 0.43 y 2.09 cuentas para consultas de frecuencia dado un valor de cuantil. Adicionalmente, presentamos una arquitectura para un acelerador hardware que implementa este algoritmo aprovechando el paralelismo de los dispositivos FPGA. La arquitectura fue implementada sobre una tarjeta Alveo U280 y presenta un uso de recursos menor al 13 % de los recursos disponibles en la tarjeta, lo que permitiría integrar hardware de procesamiento adicional en la misma plataforma. El acelerador recibe y procesa los paquetes con una frecuencia de reloj máxima de 392 MHz, soportando tasas de comunicación de al menos 200 Gbps. La latencia de consulta, es decir, el tiempo máximo que se demora el procesador en responder a una consulta después de haber recibido los top-K y calculado los parámetros de la distribución es de 38.5 μs para consultas de frecuencia dado un valor de cuantil y 4.2 μs para consultas de cuantil dado un valor de frecuencia. Por otro lado, el uso de un procesador soft-core nos entrega flexibilidad sobre las consultas que se pueden responder con el algoritmo. Como trabajo futuro, buscaremos mejorar la estimación del punto L, con el objetivo de incrementar la precisión del algoritmo en el rango de frecuencias bajas, donde actualmente se observan mayores desviaciones. Asimismo, se explorará el uso de técnicas de aproximación para el logaritmo a fin de reducir la latencia en el cálculo de parámetros de la distribución. Adicionalmente, exploraremos la posibilidad de usar este algoritmo para estimar otras propiedades de los flujos de red como, por ejemplo, el tamaño del payload (la porción de un paquete que transporta la información real del usuario o de la aplicación).

Publicaciones

A streaming algorithm and hardware accelerator for top-K flow detection in network traffic
Aceptado en 2025 en Euromicro Conference on Digital System Design. Presenta lo descrito en los Capítulos 5.2 y 5.3 y los resultados presentados en el Capítulo 6.1.

Sketch-based algorithm and hardware accelerator for quantile estimation of network flow frequencies
Enviado a la revista Computer Networks. Presenta todo el resto del algoritmo de estimación de cuantiles y la arquitectura del acelerador propuesto.

Bibliografía

[1] 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.

[2] "CAIDA Anonymized Internet Traces," https://catalog.caida.org/dataset/passive_metadata.

[3] 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.

[4] 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.

[5] 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.

[6] 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.

[7] 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.

[8] 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.

[9] 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.

[10] 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.

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

[12] 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.

[13] 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.

[14] T. Yang, J. Jiang, P. Liu, Q. Huang, J. Gong, Y. Zhou, R. Miao, X. Li, and S. Uhlig, "Elastic sketch: Adaptive and fast network-wide measurements," pp. 561–575, 2018.

[15] 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.

[16] 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.

[17] P. Flajolet, É. Fusy, O. Gandouet, and F. Meunier, "Hyperloglog: the analysis of a near-optimal cardinality estimation algorithm," *Discrete mathematics & theoretical computer science*, no. Proceedings, 2007.

[18] 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.

[19] ——, "Random sampling techniques for space efficient online computation of order statistics of large datasets," in *Proceedings of the 1999 ACM SIGMOD international conference on Management of data*, 1999, pp. 251–262.

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

[21] 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.

[22] 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.

[23] 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.

[24] 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.

[25] E. Gan, J. Ding, K. S. Tai, V. Sharan, and P. Bailis, "Moment-based quantile sketches for efficient high cardinality aggregation queries," *Proceedings of the VLDB Endowment*, vol. 11, no. 11, 2018.

[26] L. Wasserman, *All of statistics: a concise course in statistical inference*. Springer Science & Business Media, 2013.

[27] C. Masson, J. E. Rim, and H. K. Lee, "Ddsketch: A fast and fully-mergeable quantile sketch with relative-error guarantees," *Proceedings of the VLDB Endowment*, vol. 12, no. 12.

[28] 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.

[29] 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.

[30] A. C. Gilbert, Y. Kotidis, S. Muthukrishnan, and M. J. Strauss, "How to summarize the universe: Dynamic maintenance of quantiles," in *VLDB'02: Proceedings of the 28th International Conference on Very Large Databases*. Elsevier, 2002, pp. 454–465.

[31] G. Cormode and S. Muthukrishnan, "An improved data stream summary: The count-min sketch and its applications," in *LATIN 2004: Theoretical Informatics: 6th Latin American Symposium, Buenos Aires, Argentina, April 5-8, 2004. Proceedings 6*. Springer, 2004, pp. 29–38.

[32] L. Wang, G. Luo, K. Yi, and G. Cormode, "Quantiles over data streams: An experimental study," in *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data*, 2013, pp. 737–748.

[33] M. Charikar, K. Chen, and M. Farach-Colton, "Finding frequent items in data streams," in *International Colloquium on Automata, Languages, and Programming*. Springer, 2002, pp. 693–703.

[34] G. Cormode and S. Muthukrishnan, "An improved data stream summary: the count-min sketch and its applications," *Journal of Algorithms*, vol. 55, no. 1, pp. 58–75, 2005.

[35] C. Estan and G. Varghese, "New directions in traffic measurement and accounting," in *Proceedings of the 2002 conference on Applications, technologies, architectures, and protocols for computer communications*, 2002, pp. 323–336.

[36] T. Yang, Y. Zhou, H. Jin, S. Chen, and X. Li, "Pyramid sketch: A sketch framework for frequency estimation of data streams," *Proceedings of the VLDB Endowment*, vol. 10, no. 11, pp. 1442–1453, 2017.

[37] P. Roy, A. Khan, and G. Alonso, "Augmented sketch: Faster and more accurate stream processing," in *Proceedings of the 2016 International Conference on Management of Data*, 2016, pp. 1449–1463.

[38] J. Liu, Z. Li, H. Du, H. Zhou, L. Li, Y. An, Y. Zhang, K. Liu, and Q. Li, "An effective and accurate flow size measurement using funnel-shaped sketch," *Computer Networks*, vol. 247, p. 110467, 2024.

[39] T. Yang, H. Zhang, J. Li, J. Gong, S. Uhlig, S. Chen, and X. Li, "Heavykeeper: an accurate algorithm for finding top-k elephant flows," *IEEE/ACM Transactions on Networking*, vol. 27, no. 5, pp. 1845–1858, 2019.

[40] Q. Shi, Y. Xu, J. Qi, W. Li, T. Yang, Y. Xu, and Y. Wang, "Cuckoo counter: Adaptive structure of counters for accurate frequency and top-k estimation," *IEEE/ACM Transactions on Networking*, vol. 31, no. 4, pp. 1854–1869, 2023.

[41] 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.

[42] X. Yang, K. Yang, H. Zhang, G. Jin, and Y. Lu, "Cts sketch: A sketch scheme for precise identification of top-k flows combined with sdn," in *2024 Twelfth International Conference on Advanced Cloud and Big Data (CBD)*. IEEE, 2024, pp. 386–391.

[43] L. Cao, Q. Shi, Y. Liu, H. Zheng, Y. Xin, W. Li, T. Yang, Y. Wang, Y. Xu, W. Zhang *et al.*, "Bubble sketch: A high-performance and memory-efficient sketch for finding top-k items in data streams," in *Proceedings of the 33rd ACM International Conference on Information and Knowledge Management*, 2024, pp. 3653–3657.

[44] 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.

[45] B. Wang, R. Chen, and L. Tang, "Easyquantile: Efficient quantile tracking in the data plane," in *Proceedings of the 7th Asia-Pacific Workshop on Networking*, 2023, pp. 123–129.

[46] "Intel® Tofino™ Series," https://www.intel.com/content/www/us/en/products/details/network-io/intelligent-fabric-processors/tofino.html, [Accessed 29-07-2024].

[47] 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.

[48] 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.

[49] "Intel® Tofino™ 2," https://www.intel.com/content/www/us/en/products/details/network-io/intelligent-fabric-processors/tofino-2.html, [Accessed 29-07-2024].

[50] J. E. Soto, S. Vera, Y. Fernández, D. Yunge, C. Hernández, and M. Figueroa, "A sketch-based algorithm for network-flow entropy estimation on programmable switches using p4," in *2023 26th Euromicro Conference on Digital System Design (DSD)*. IEEE, 2023, pp. 79–86.

[51] 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.

[52] C. Gallardo-Pavesi, Y. Fernandez, J. E. Soto, C. Hernández, and M. Figueroa, "A hardware accelerator for quantile estimation of network packet attributes," in *2024 27th Euromicro Conference on Digital System Design (DSD)*. IEEE, 2024, pp. 114–121.

[53] D. Grochol and L. Sekanina, "Multi-objective evolution of hash functions for high speed networks," in *2017 IEEE Congress on Evolutionary Computation (CEC)*. IEEE, 2017, pp. 1533–1540.

[54] D. Ding, "Carbine: Exploring additional properties of hyperloglog for secure and robust flow cardinality estimation," in *IEEE INFOCOM 2024-IEEE Conference on Computer Communications*. IEEE, 2024, pp. 471–480.

[55] S. Nolting and A. T. A. Contributors, "The neorv32 risc-v processor," Jun. 2025. [Online]. Available: https://doi.org/10.5281/zenodo.15579069

[56] 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.

[57] J. F. Zazo, S. Lopez-Buedo, M. Ruiz, and G. Sutter, "A single-fpga architecture for detecting heavy hitters in 100 gbit/s ethernet links," in *2017 International Conference on ReConFigurable Computing and FPGAs (ReConFig)*. IEEE, 2017, pp. 1–6.