HARMONIA: A Swift and Accurate Approximate Data Structure for Real-Time Heavy Flow Detection in High-Speed Networks

Weihe Li[1], Tianyue Chu[2], Christos-Savvas Bouganis[3], and Paul Patras[1]
[1] University of Edinburgh, Edinburgh EH8 9AB, United Kingdom
[2] Telefonica Research, Barcelona 08019, Spain
[3] Imperial College London, London SW7 2AZ, United Kingdom
weihe.li@ed.ac.uk, tianyue.chu@telefonica.com,
christos-savvas.bouganis@imperial.ac.uk, paul.patras@ed.ac.uk

Abstract. In high-speed networks, where numerous applications continuously generate high-volume data flows, real-time traffic monitoring of high-frequency flows (heavy hitters) is critical for maintaining reliable communication and enabling rapid anomaly detection (like DDoS attacks). In such settings, identifying heavy hitters is particularly onerous due to the limited fast memory available in practice. To address this, approximate data structures, or sketches, are widely employed. However, existing sketches often present low detection accuracy owing to highly-skewed traffic distributions and deployment challenges on existing hardware, e.g. programmable switches, due to stringent resource and arithmetic limitations. This paper introduces HARMONIA, a new sketch designed for efficient and accurate heavy flow detection. Unlike state-of-the-art methods that rely on multi-dimensional features to protect heavy flows (resulting in memory inefficiency), HARMONIA embraces traffic skewness and builds on the insight that when a flow's frequency exceeds a certain threshold, it is highly likely to be heavy. HARMONIA thus refrains from replacing such tracked flows upon hash collisions with arriving non-heavy flows. Experimental results across multiple implementations (CPU, programmable switch, and FPGA) and evaluations on real-world network traces show that HARMONIA achieves up to 27.83% higher detection accuracy compared to existing sketches. Additionally, its hardware resource footprints are no more than 18% on a programmable switch and 10% on a FPGA platform.

Keywords: approximate data structures · sketch · high-speed data streams · heavy hitter detection · anomaly detection · programmable switch · FPGA.

1 Introduction

The exponential growth in network traffic from various high-speed applications, such as video streaming, high-frequency trading, and cloud computing, has led to high-volume data flows across modern networks [25]. Real-time traffic monitoring is essential for identifying high-frequency flows, or "heavy hitters", which may conceal anomalies such as (distributed) denial-of-service (D)DoS attacks that need to be detected promptly to maintain service reliability, or simply handled appropriately to ensure quality-of-service (QoS) guarantees are satisfied.

Real-time heavy flow detection is challenging because it must keep up with rapidly increasing traffic speeds while maintaining high accuracy [22]. Furthermore, fast processing requires the use of high-speed and low-latency memory, such as the L1 cache of CPUs or programmable switches [16]. However, due to memory size limitations, such as the commonly available 64KB L1 cache [17,18], it is impractical to track every flow and then detect those heavy ones in high-volume traffic environments. To address these limitations, approximate data structures, or sketches that utilize hash techniques, typically organized as a two-dimensional table with multiple rows, have been introduced to perform various detection tasks while conserving memory [9, 15, 23].

Limitations of Existing Methods: Despite a series of advances in heavy flow detection, current solutions face notable limitations. (i) State-of-the-art methods, like Stable-Sketch [18] and Tight-Sketch [17], leverage multidimensional features, such as combining flow frequency with bucket stability or arrival continuity, to enhance protection for heavy flows. Although this approach enhances accuracy when the number of heavy flows is small, the additional memory required for tracking these features reduces the total number of available buckets, ultimately degrading memory efficiency and accuracy as the number of heavy flows increases. (ii) Methods like MV-Sketch [22] and Count-Min Sketch [9] demand large memory budgets to achieve high detection accuracy. Their replacement strategies often result in heavy flows being inadvertently evicted by a significantly larger number of non-heavy ones, especially under limited memory conditions where hash collisions are more severe. (iii) In high-speed networks, e.g. in data centers and IoT networks, programmable switches [3] are increasingly deployed due to their customizability and flexibility. However, these powerful devices have inherent limitations. For instance, programmable switches typically offer a limited number of processing stages (e.g., 12) and support only a narrow range of operations, generally restricted to basic arithmetic like addition and subtraction [8]. These constraints render many existing sketches impractical for such platforms, particularly those requiring complex update operations like counter merging [4].

Contributions: To address these challenges, we propose a new sketch, called Harmonia, designed for highly accurate heavy flow detection. To this end, when an incoming packet triggers hash collisions across all rows, we employ a probabilistic replacement strategy to decide whether the new flow should be inserted into the sketch. This approach builds on the key observation that network traffic distributions are typically skewed, whereby only a small fraction of flows are heavy in practice [13]. Additionally, a data analysis we conduct reveals that when a flow's frequency exceeds a certain threshold, it is highly likely to be a heavy flow. In such cases, erroneously evicting a tracked flow due to collisions with non-heavy flows can increase estimation errors. To prevent this, we disable replacement operations when a tracked flow's frequency crosses the threshold, enhancing detection accuracy even under constrained memory conditions. Besides, our design avoids intricate computations, like matrix multiplication and bit-level counter merging, making it well-suited for programmable switch deployment.

We implement HARMONIA on multiple platforms to demonstrate its versatility and adaptability, including a CPU-based deployment in C++, an implementation on a programmable switch platform using the P4 language [6], and an FPGA-based implementation with Verilog. Extensive evaluations on real-world network traces [1] demonstrate that HARMONIA outperforms the advanced Stable-Sketch [18] by achieving up to 15.4% higher detection accuracy and reducing estimation errors by an average of 32.27%. Moreover, in hardware deployments, HARMONIA exhibits modest resource demands, using no more than 18% of those available on a programmable switch and 10% on a FPGA platform.

2 Background and Related Work

2.1 Sketches

In memory-constrained scenarios, recording information for every flow is impractical. To address this challenge, hashing-based sketches are commonly used for high-speed network measurements [22]. A well-known instance is the Count-Min Sketch [9], which organizes data in a table with r rows, each containing multiple buckets paired with a unique hash function. Each bucket has a counter to track flow occurrences. When a packet arrives, its flow key (e.g., source IP address) is hashed into r buckets across the rows, incrementing the corresponding counters by 1. For querying, a flow hashes into buckets in each row, and the smallest counter value among them serves as its estimated frequency.

2.2 Heavy Hitter Detection

A heavy hitter is a flow e in a network traffic set D whose frequency $f(e)$ meets or exceeds a specified threshold $\phi \cdot |D|$, where ϕ is a predefined fraction (e.g., $\phi = 0.001$) and $|D|$ is the total number of packets in D. Formally, e is a heavy hitter if $f(e) \geq \phi \cdot |D|$.

For detecting heavy hitters, various sketch-based methods have been introduced [12, 19]. Count-Min Heap [9] incorporates an additional heap to track flows exceeding the threshold, but the extra data structure reduces memory efficiency, making it less suitable for memory-constrained environments such as programmable switches. MV-Sketch [22] uses a majority vote algorithm to retain heavy flows but involves a larger memory footprint to achieve high detection accuracy. UA-Sketch [29] utilizes a probabilistic eviction strategy that considers flow arrival patterns to evict non-heavy flows. However, its design is based on the assumption that all heavy flows arrive continuously without interruptions, resulting in reduced detection accuracy when this assumption does not hold. Tight-Sketch [17] and Stable-Sketch [18] introduce additional features to enhance the protection of potential heavy flows. However, the additional features result in memory inefficiency and deployment complexity. Other methods, such as SALSA [4], utilize flexible counters that adjust based on traffic distribution. However, these methods rely on complex hierarchical data structures and computationally intensive operations, which result in lower processing speeds and complicate deployment on practical hardware.

2.3 Hardware Platforms

Programmable Switches offer high-speed packet processing along with significant flexibility and customization. Deployed at the network edge or within data

centers, they enable tailored traffic monitoring functions capable of efficient, line-rate processing of high-volume flows. As a result, programmable switches have become a preferred solution for improving the reliability of IoT and data center networks [10, 14, 26].

However, they faces several limitations: (i) each stage of the processing pipeline has a restricted amount of available memory (e.g., 1.4MB shared between forwarding and monitoring functions) [21]; (ii) only a limited set of arithmetic operations is supported, including basic functions like addition and subtraction; (iii) the pipeline is restricted to a fixed number of physical stages (e.g., 12); and (iv) each stage is unable to access memory from preceding stages. As a result, any sketch update algorithm is desired to follow a unidirectional workflow, meaning data flows from the first stage to the last [30].

FPGAs are reconfigurable digital chips that can be programmed to implement a diverse array of digital hardware circuits [7]. They are built from programmable logic blocks, I/O modules, and configurable interconnects, all managed by SRAM-based cells. Designers typically employ hardware description languages, such as Verilog, to specify the desired functionality. As noted in [23, 24], because FPGAs can deliver high throughput for high-speed network processing, they have become a popular platform for implementing sketches. However, FPGAs also have constraints, including limited support for arithmetic operations, such as division, which necessitates careful design of sketch update strategies.

3 HARMONIA

3.1 Design Philosophy

To maintain high accuracy under limited memory budgets, the sketch design needs to effectively adapt to the skewed traffic distributions commonly observed in practice [17]. This adaptation helps prevent heavy flows from being mistakenly evicted by large volumes of non-heavy flows when hash collisions are frequent. Overall, the design of HARMONIA is guided by the following two principles.

Principle 1: Compared to methods that rely on update strategies such as the majority vote algorithm [22], which often require substantially memory to achieve high detection accuracy under skewed data distributions, we employ a probabilistic replacement strategy in which the update operation is guided by flow frequency – flows with higher frequencies should be less likely to be evicted by incoming items. The simplicity of this strategy further facilitates implementation on hardware platforms, including programmable switches and FPGAs [30]. Our design stems from an analysis of the distribution of flow frequencies in real-world scenarios that we conduct, specifically of two traces from each of the CAIDA [1] and ToN_IoT [5] datsets. As we reveal in Table 1 the majority of flows are non-heavy, with over 95% in these traces having a frequency of 50 or less. In contrast, the number of flows with a frequency exceeding 300 is minimal; e.g., only around 1.22% in the CAIDA 2018 trace and 0.05% in the IoT_Scanning trace. This leads us to conclude that under limited memory where hash collisions are frequent, heavy flows are likely to be incorrectly evicted from buckets by the overwhelming number of small flows [29], which HARMONIA seeks to address.

Table 1: Flow Frequency Distribution in Practical Traces.

Frequency	CAIDA2015	CAIDA2018	IoT_DDoS	IoT_Scanning
[1, 50]	95.63%	96.37%	98.79%	99.57%
(50, 100]	1.50%	1.30%	0.37%	0.25%
(100, 300]	1.44%	1.11%	0.51%	0.12%
>300	1.43%	1.22%	0.33%	0.05%

Principle 2: Unlike state-of-the-art approaches such as Stable-Sketch [18] and Tight-Sketch [17], which leverage dual-feature mechanisms to protect heavy flows and consequently require additional memory, resulting in fewer buckets and increased hash collisions that elevate the risk of erroneously evicting heavy flows, our method protects heavy flows using only a single parameter, namely the actual frequency tracked in the value counter, ensuring efficient memory usage.

3.2 Data Structure

Existing sketch data structures can be broadly categorized into two types: flat [9, 22, 29] and hierarchical [19]. While hierarchical structures are generally more memory efficient due to their carefully designed counters of varying lengths, their complexity, such as bit-level counter management, renders them impractical for deployment on programmable switches. Therefore, we adopt a flat-based structure for its simplicity and ease of hardware implementation.

Fig.1 illustrates the data structure of HARMONIA, which consists of r rows, each associated with a pairwise-independent hash function h_i $(1 \leq i \leq r)$. Each row contains b columns (*buckets*), where each bucket $B(i, j)$ $(1 \leq j \leq b)$ has two fields: the flow key K and its frequency counter V.

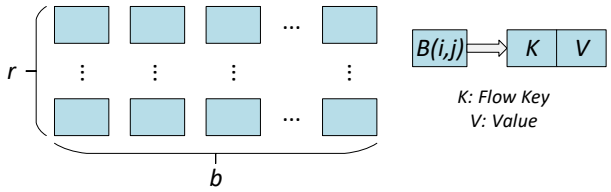


Fig. 1: HARMONIA's data structure.

3.3 Main Operations

Update: When an incoming packet belonging to a flow arrives, the update process falls into two possible stages, as outlined in Algorithm 1.

HARMONIA begins by hashing the key of the incoming flow into a bucket in the first row using the hash function h_1 (Line 3). If the corresponding bucket $B(1, h_1(e_m.key))$ is empty (i.e., $key = \emptyset$ and $count = 0$), the flow key is inserted into the bucket, and the count is initialized to 1, terminating the update process (Lines 4–8). If the bucket already contains the same flow key, the count is incremented by 1, and the process ends (Lines 9–11). Otherwise, if the bucket contains a different flow key, the algorithm proceeds to the next row. This process is repeated row by row using the corresponding hash functions $h_2, h_3, ..., h_r$. In each row, if the bucket is empty or contains the same flow key, the algorithm

Algorithm 1: HARMONIA Update Operation

```
Input: Flow key key, Hash functions h_1, h_2, ..., h_r, Value val, Sketch
       parameters r (number of rows), b (buckets per row), Threshold Ω
1  Initialization: All buckets in the sketch are initialized with key set to ∅ and
     count set to 0.
2  Stage I: Attempt to Find or Update a Bucket
3  for i = 1 to r do
4      bucket ← B(i, h_i(key));
5      if bucket.key == ∅ and bucket.count == 0 then
6          bucket.key ← key;
7          bucket.count ← 1;
8          return;
9      else if bucket.key == key then
10         bucket.count ← bucket.count + 1;
11         return;
12     else if bucket.count < min then
13         min ← bucket.count;
14         loc ← (i, h_i(key));
15 bucket ← B(loc);
16 Stage II: Probability-Based Replacement
17 if bucket.count >= Ω then
18     return;
19 k ← rand(0, 1);
20 threshold ← 1/(bucket.count + 1);
21 if k < threshold then
22     bucket.key ← key;
23     bucket.count ← bucket.count + 1;
24     return;
25 else
26     return;
       // Discard the item if the condition is not met
```

updates the bucket and terminates. If no matching or empty bucket is found after traversing all rows, HARMONIA identifies the bucket with the smallest count across all rows and proceeds to the next stage (Lines 12–15).

In the second stage, HARMONIA decides whether to replace the bucket with the smallest count. If the smallest count $count \geq \Omega$, where Ω is a predefined threshold (*analyzed in detail in Section 6.2*), the bucket retains its current key, and the incoming flow is discarded (Lines 16–18). Otherwise, HARMONIA calculates the replacement probability. A random number k is generated within the range 0 and 1, and the replacement threshold is computed as $\frac{1}{count+1}$ (Lines 19–20). If the replacement succeeds, the bucket's key is updated with the new flow key, and the count is incremented by 1 (Lines 21–24). If the replacement fails, the incoming flow is discarded, and the process terminates (Lines 25–26).

Query: To retrieve the heavy flows, HARMONIA simply scans all the buckets. If the frequency value of a tracked flow exceeds the predefined threshold for heavy hitters, the flow is identified as a heavy hitter.

4 Formal Analysis

In this section, we prove that HARMONIA achieves (ϵ, δ)-counting [18], thus demonstrating its low error rate in estimating heavy flows.

Theorem 1. *In Algorithm 1, given a small positive number ϵ, for any heavy flow e_m with frequency $f(e_m)$, the estimated frequency $\hat{f}(e_m)$ satisfies:*

$$\Pr\left(\hat{f}(e_m) \leq f(e_m) - \epsilon|D|\right) \leq \delta$$

where $\delta = \frac{1}{\epsilon b}\left[1 - \frac{\Omega}{2|D|^2}\right]$. Here, b represents the number of buckets in each row, $|D|$ is the total number of packets, and Ω is the predefined threshold.

Proof. Consider a heavy flow e_m. Whenever e_m arrives and is mapped to the same bucket $B(i,j)$ of a different flow $e \neq e_m$, the counter of that bucket either increases by 1 or remains unchanged. We adopt the following assumption, which is generally valid in heavy-flow detection [17, 18, 28]: once a heavy flow e_m is mapped into a bucket, it remains in that bucket until the detection process is complete. Let t^* be the (random) time slot at which the flow e_m *first* enters a bucket. Define α_m to be the number of arrivals of e_m *before* time t^* that did not cause e_m to be placed in any bucket; this quantity represents the *underestimation error*. When e_m finally enters a bucket at time t^*, the *overestimation error* β_m is the current count of that bucket. By construction, the estimated frequency $\hat{f}(e_m)$ can be written as $\hat{f}(e_m) = f(e_m) - \alpha_m + \beta_m$,, where $f(e_m)$ is the true frequency of flow e_m. Using Markov's inequality, for any $\epsilon > 0$, we have:

$$\Pr\left(\hat{f}(e_m) \leq f(e_m) - \epsilon|D|\right) = \Pr\left(\alpha_m - \beta_m \geq \epsilon|D|\right) \leq \frac{\mathbb{E}\left(\alpha_m - \beta_m\right)}{\epsilon|D|}. \quad (1)$$

1. Bounding α_m. Define Δ_m as the probability that flow e_m hashes into the same bucket with a distinct flow but does not replace it. For each flow e_m, the probability that in each of the r arrays there are hash collisions in the bucket to which flow e_m is mapped is:

$$\Pr\left(\bigcap_{\substack{n\in\mathcal{M}\setminus\{m\}\\ v\in\{1,\ldots,r\}}} \{h_v(e_m) = h_v(e_n)\}\right) = \left[1 - \left(1 - \frac{1}{b}\right)^{M-1}\right]^r$$

Where M is the total number of distinct flows. Moreover, the probability that a given flow is mapped to a particular bucket is $\frac{1}{b}$. Consequently, for flow e_m, the probability that it maps to the same bucket as a distinct flow but does not replace it is bounded by:

$$\Delta_m \leq \frac{1}{b}\left[1-\left(1-\frac{1}{b}\right)^{M-1}\right]^r \frac{B(i.j).count}{B(i.j).count+1} \leq \frac{1}{b}\left[1-\left(1-\frac{1}{b}\right)^{M-1}\right]^r \quad (2)$$

Let $X_m(t)$ be the total number of arrivals of flow e_m in the time interval $[0,t]$. At time t^*, the expected number of arrivals of e_m is $\mathbb{E}(X_m(t^*)) \leq |D|$. Thus we have,

$$\mathbb{E}(\alpha_m) \leq \mathbb{E}(\Delta_m X_m(t^*)) \leq \frac{|D|}{b}\left[1-\left(1-\frac{1}{b}\right)^{M-1}\right]^r$$

2. Bounding β_m. Let the original flow in the bucket $B(i,j)$, where the flow e_m is hashed into, be denoted as flow e_n. The count associated with this flow in the bucket is V_n, which satisfies $V_n \in [1, X_n(t^*)]$. When the flow e_m replaces the flow e_n, the overestimation in count is given by V_n. A necessary condition for replacement is that e_m collides with a distinct flow e_n. As above, the probability of such a collision across all r arrays is at most $\frac{1}{b}\Pr\left(\bigcap_{n\in\mathcal{M}\setminus\{m\}}\{h_v(e_m)=h_v(e_n)\}\right)$. In addition, the replacement needs the conditions that $(X_n(t^*)<\Omega)$ with the replacement rate $\frac{1}{V_n+1}$. Thus,

$$\beta_m = \frac{1}{b}\Pr\left(\bigcap_{n\in\mathcal{M}\setminus\{m\}}\{h_v(e_m)=h_v(e_n)\}\right)\Pr\left(X_n(t^*)<\Omega\right)\frac{V_n}{V_n+1}. \quad (3)$$

We next lower-bound the probability that $(X_n(t^*)<\Omega)$. We have

$$\mathbb{E}\left(\Pr\left(X_n(t^*)<\Omega\right)\right) = 1-\mathbb{E}\left(\Pr\left(X_n(t^*)\geq\Omega\right)\right) \geq 1-\frac{|D|-\Omega}{|D|} = \frac{\Omega}{|D|}. \quad (4)$$

$$\mathbb{E}(\beta_m) \geq \frac{1}{b}\left[1-\left(1-\frac{1}{b}\right)^{M-1}\right]^r \frac{\Omega}{|D|}\min_{1\leq V_n\leq\Omega}\frac{V_n}{V_n+1} \geq \frac{1}{2b}\left[1-\left(1-\frac{1}{b}\right)^{M-1}\right]^r \frac{\Omega}{|D|} \quad (5)$$

3. Combining the bounds. Putting the above estimates together:

$$\begin{aligned}\mathbb{E}(\alpha_m-\beta_m) &\leq \frac{|D|}{b}\left[1-\left(1-\frac{1}{b}\right)^{M-1}\right]^r - \frac{1}{2b}\left[1-\left(1-\frac{1}{b}\right)^{M-1}\right]^r \frac{\Omega}{|D|} \\ &= \frac{1}{b}\left[1-\left(1-\frac{1}{b}\right)^{M-1}\right]^r \left[|D|-\frac{\Omega}{2|D|}\right] \quad (6)\end{aligned}$$

Hence, recalling Equation (1),

$$\begin{aligned}\Pr\left(\hat{f}(e_m)\leq f(e_m)-\epsilon|D|\right) &\leq \frac{\mathbb{E}\left(\alpha_m-\beta_m\right)}{\epsilon|D|} \leq \frac{1}{\epsilon b}\left[1-\left(1-\frac{1}{b}\right)^{M-1}\right]^r \left[1-\frac{\Omega}{2|D|^2}\right] \\ &\leq \frac{1}{\epsilon b}\left[1-\frac{\Omega}{2|D|^2}=\right] = \delta \quad (7)\end{aligned}$$

5 Implementation on Hardware Platforms

5.1 Programmable Switches

The implementation of HARMONIA is optimized for programmable switches, enabling real-time heavy flow detection. It employs a two-row sketch, where each bucket contains two 32-bit registers storing a flow key (e.g., source IP) and a frequency counter. These registers are stateful arrays, facilitating efficient in-pipeline lookups and updates. Flow keys are mapped to buckets using built-in `crc-16` and `crc_16_dds_110` hash functions, reducing collisions and enhancing accuracy compared to single-row sketches [22, 30]. HARMONIA defines custom headers and metadata to manage stateful operations. The `hp_hdr` header stores intermediate states, such as the smallest count and corresponding bucket location, crucial during recirculation. Metadata fields track operational data like flow keys, bucket locations, states, and probabilistic replacement conditions.

When a packet arrives, it is hashed into buckets via CRC-16 functions and accessed with the `modify_field_with_hash_based_offset` primitive. Stateful ALUs then determine bucket states (empty, matching, or occupied by a different key). An empty bucket is initialized with the new flow key and a count of 1, while matching keys increment the counter. If neither condition is met, the bucket with the smallest count is marked for replacement. Then, probabilistic replacement occurs based on the predefined threshold Ω. If the smallest count exceeds Ω, no replacement occurs, and the incoming flow is discarded. Otherwise, due to the difficulty of performing floating-point calculations on programmable switches, the replacement threshold $\frac{2^{16}}{\text{count}+1}$ is computed using integer arithmetic within a blackbox ALU [13]. This threshold is then compared against a random 16-bit number generated by the `modify_field_rng_uniform` primitive. If replacement conditions are met, the bucket's key is updated, and the count incremented. This probabilistic replacement requires recirculation to manage intermediate bucket states, tracked by the `hp_hdr` header.

The HARMONIA implementation uses around 650 lines of P4 code, leveraging hardware primitives and adhering to the programmable pipeline constraints.

5.2 FPGA

HARMONIA's FPGA implementation fully pipelines the update process into four distinct stages, with two rows processed concurrently (employing a slightly modified update strategy). In stage 1, two independent CRC32 hash modules, implemented as synchronous logic blocks, compute two distinct 32-bit hash values in parallel from the incoming 32-bit flow key to generate bucket addresses. In stage two, these addresses are applied to dual-port Block Random Access Memory (BRAMs), where each synchronous read takes two clock cycles. In stage three, the update logic evaluates each accessed bucket: if the bucket is empty or already contains the same flow key, the counter is simply incremented and the key updated; otherwise, if the current counter value is below the threshold (Ω), a 32-bit random number is generated and–only if the product of this random number and the current counter value is less than 2^{32}–is the stored key replaced

while the counter is incremented. Finally, in stage four, each row independently writes back the updated counter and key values to their respective memories. This pipelined key/value memory access effectively increases the clock frequency. Overall, HARMONIA is implemented in around 370 lines of Verilog code.

6 Evaluation

6.1 Setup

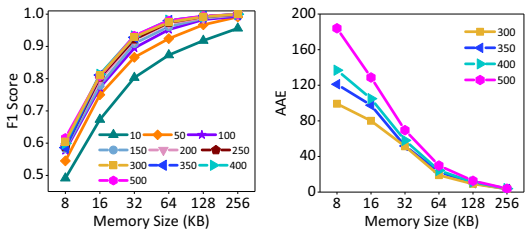
Platform (1) CPU Platform. We implement HARMONIA in C++ and evaluate it on a laptop with an Intel® Core™ i5-1135G7 @ 2.40GHz processor, 16GB of DRAM, running Ubuntu 20.04 LTS.
(2) Programmable Switch. We prototype HARMONIA in the P4 language [6] and compile it using Intel® P4 Studio [2], an industrial-grade development environment that facilitates deployment on Intel® Tofino™ ASICs [3].
(3) FPGA. We implement HARMONIA in Verilog and compile it using Vivado 2024.2 [11]. We pick the UltraScale+ XCU250-L2FIGD2104E FPGA as our target device.

Datasets To thoroughly evaluate HARMONIA, we use real-world traces from the CAIDA dataset [1]. Two of these traces were previously introduced in Section 3.1, while the third is from the CAIDA 2019 dataset, containing 29.5M packets across 1.53M flows. The heavy hitter parameter ϕ is set to 10^{-4} to identify heavy hitters in these traces, using the source IP address as the flow key.

Benchmarks: We compare HARMONIA with four state-of-the-art heavy hitter detection methods: MV-Sketch [22], UA-Sketch [29], Tight-Sketch [17], and Stable-Sketch [18]. For these methods, the number of rows is set to 2. We omit comparisons with other methods, such as CocoSketch [30] and Elastic [27] because Stable-Sketch already outperforms them.

Memory Allocations: For HARMONIA and the considered baselines, we evaluate their detection accuracy under memory sizes of 8KB, 16KB, 32KB, 64KB, 128KB, and 256KB [18]. For each method, the size of each bucket is fixed. Based on the total memory size, the number of allocated buckets can be easily calculated. For example, in HARMONIA, both the key field and the counter field occupy 4 bytes each. Thus, with a memory size of 8KB, the total number of buckets allocated is 1024 $\left(\frac{8\times1024}{4+4}\right)$. Operating under limited memory offers several advantages, such as reserving more memory resources for other applications and accelerating the query process for retrieving heavy hitters.

Metrics: The detection accuracy of the algorithm is evaluated using three key metrics. (1) F1 Score: calculated as $\frac{2\times\text{recall}\times\text{precision}}{\text{recall}+\text{precision}}$, where recall represents the proportion of actual heavy hitters (ground truth) correctly identified by the algorithm, and precision denotes the proportion of correctly identified heavy hitters among all the detected ones. (2) Average Absolute Error (AAE): the mean absolute difference between the true frequency and the estimated frequency of heavy items, reflecting the accuracy of frequency estimation. (3) Update Speed: the processing speed measured in millions of packets per second (Mpps).



(a) F1 score. (b) AAE.

Fig. 2: Detection accuracy (F1 score) for varying parameter Ω. F1 score improves as Ω increases but plateaus after 300, beyond which estimation error grows.

6.2 Parameter Setting

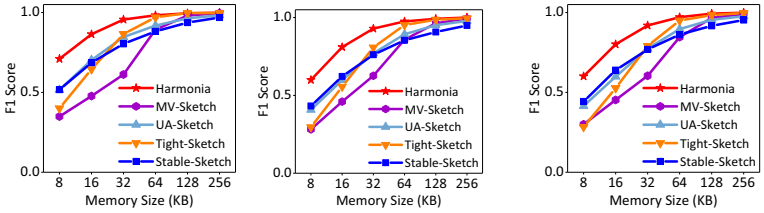
Recall that HARMONIA uses a threshold parameter Ω to enhance the protection of heavy hitters. To determine the value, following the approach of prior works [16, 19, 27], we conduct empirical evaluations on the CPU platform to determine suitable parameter configurations for our method. The CAIDA 2018 trace is used as the test trace, with similar trends observed across other traces.

As shown in Fig. 2(a), the F1 score increases significantly as Ω grows from 10 to 50. This improvement is attributed to smaller Ω values (e.g., 10), which allow many non-heavy flows to occupy the buckets, thereby reducing detection accuracy. As Ω continues to increase, the F1 score improves further, though the gains diminish once Ω reaches 300. In contrast, Fig. 2(b) shows that increasing Ω beyond 300 results in higher estimation errors. This is because excessively large Ω values cause heavy flows to be incorrectly evicted by non-heavy flows. Thus, we select $\Omega = 300$ as the preferred threshold, a configuration that demonstrates its effectiveness in subsequent extensive evaluations.

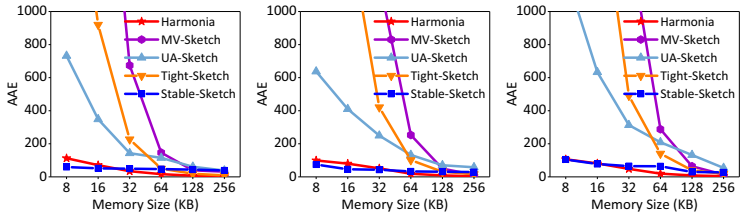
Note: Ideally, the parameter Ω could be dynamically adjusted based on real-time observations of network traffic characteristics, such as skewness. However, existing approaches are often computationally intensive [20], rendering them impractical for deployment on hardware platforms like programmable switches. As a next step, we aim to develop a dynamic adjustment mechanism for Ω that remains feasible for implementation on practical hardware.

6.3 CPU Platform

Detection Accuracy: Figs. 3(a)-(c) show the F1 scores for different methods. As illustrated, HARMONIA exhibits robustness across various memory configurations and network traces. On average, HARMONIA improves the F1 score by 11.72%-27.83%, 15.4%-26.82%, and 14.49%-27.02% over the baseline methods on the 2015, 2018, and 2019 traces, respectively. Due to space limitations, we omit the results for other metrics such as recall and precision; however, our method consistently outperforms existing approaches in these measures as well. Figure 3(d)-(f) presents the AAE for various methods. As demonstrated, HARMONIA yields the lowest estimation errors. For instance, on the CAIDA 2019



(a) F1 score (2015). (b) F1 score (2018). (c) F1 score (2019).

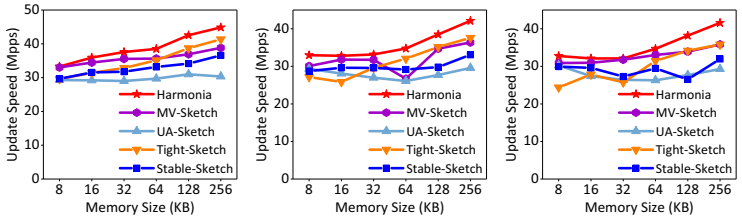


(d) AAE (2015). (e) AAE (2018). (f) AAE (2019).

Fig. 3: Detection accuracy of various methods on the CPU platform (with 2015, 2018, and 2019 referring to the CAIDA 2015, 2018, and 2019 traces, respectively).

trace, HARMONIA reduces the AAE by 37.27% compared to the state-of-the-art method, Stable-Sketch.

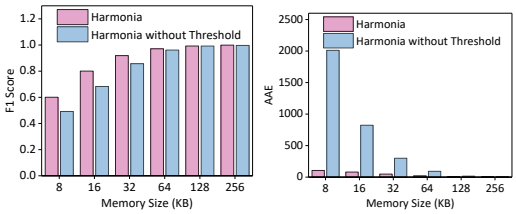
Update Speed: Fig. 4 shows the update speeds of various approaches. Observe that HARMONIA achieves the highest update speed thanks to its simple and effective update strategy that avoids extra processing of additional features. On average, it improves update speed by 8.53%-30.35%, 12.01%-27.74%, and 7.66%-26.37% compared to the baselines over the CAIDA 2015, 2018, and 2019 traces, respectively.



(a) Speed (2015). (b) Speed (2018). (c) Speed (2019).

Fig. 4: Update speed of different methods for heavy hitter detection.

Ablation Study: Fig. 5 compares the performance with and without utilizing the threshold Ω on the CAIDA 2019 trace. The results show that using Ω effectively protects heavy flows from being incorrectly replaced by a large number of non-heavy flows, particularly under limited memory budgets. For instance, compared to HARMONIA, the version without Ω experiences a 22.15% degradation in F1 score when the memory size is 8KB (1024 buckets). Additionally, the careful protection provided by Ω improves frequency estimation for heavy flows, significantly reducing the AAE across different memory allocations. These findings highlight the effectiveness of incorporating the threshold Ω.



(a) F1 score. (b) AAE.

Fig. 5: Performance comparison with/without the threshold Ω.

6.4 Resource Utilization on Hardware Platforms

Programmable Switch: Table 2 lists the resource usage of HARMONIA on the programmable switch. Observe that most of the resource consumption comes from the Logical Table ID, utilizing 17.19% of the total resources. Gateways, which contribute 15.62% of the resources, are used to implement conditional statements such as if/else logic. Additionally, Meter ALUs and the Exact Match Search Bus each utilize 10.42% of the resources, supporting efficient packet processing and ensuring accurate flow recording. HARMONIA uses SRAM to maintain registers, and consumes only 1.56% of the available SRAM resources. Similarly, other components such as Map RAM (1.74%) and Stash (0.52%) have limited usage, showcasing HARMONIA's efficiency. Overall, HARMONIA demonstrates a limited resource footprint, leaving sufficient capacity to deploy additional applications on the programmable switch.

Table 2: Resource Usage and Percentage Details on the Programmable Switch.

Resource	Exact Match Input xbar	Hash Bit	Hash Dist Unit	Gateway	SRAM	Map RAM
Usage	86	103	6	30	15	10
Percentage (%)	5.60	2.06	8.33	15.62	1.56	1.74
Resource	VLIW Instr	Meter ALU	Stash	Exact Match Search Bus	Logical Table ID	
Usage	19	5	1	20	33	
Percentage (%)	4.95	10.42	0.52	10.42	17.19	

FPGA Table 3 lists the overhead of HARMONIA on the FPGA platform. We can see that the design requires a relatively small fraction of the available lookup tables, LUTRAM and registers, while the usage of BRAM (7.14%) and I/O (9.91%) is well within the limits of the device, leaving enough resources for other applications. Also, the throughput of HARMONIA is 82.2 Mpps, which demonstrates a high processing speed. These results validate that HARMONIA can be effectively deployed on the FPGA platform.

Table 3: FPGA Resource Utilization.

Metric	Look-Up Tables (LUTs)	LUTRAM	Registers	BRAM Blocks	I/O Pins
Usage	1,075	33	685	192	67
Percentage(%)	0.06	0.004	0.02	7.14	9.91

7 Conclusion

Detecting heavy flows is a crucial task in various network scenarios, including the identification of malicious traffic in IoT and data center networks. In this

paper, we introduced HARMONIA, a novel sketch designed to accurately detect heavy flows under stringent memory constraints. HARMONIA employs a probabilistic replacement strategy along with an effective threshold mechanism to ensure that heavy flows are not displaced by non-heavy ones within each bucket. Its straightforward and efficient design is well-suited for hardware with limited resources and computational capabilities, while maintaining high detection accuracy across various platforms, confirming its effectiveness.

References

1. The CAIDA anonymized internet traces, http://www.caida.org/data/overview/
2. Intel® p4 studio, https://www.intel.com/content/www/us/en/products/details/network-io/intelligent-fabric-processors/p4-studio.html
3. Agrawal, A., Kim, C.: Intel tofino2–a 12.9 tbps p4-programmable ethernet switch. In: 2020 IEEE Hot Chips 32 Symposium (HCS). pp. 1–32. IEEE Computer Society (2020)
4. Basat, R.B., Einziger, G., Mitzenmacher, M., Vargaftik, S.: Salsa: self-adjusting lean streaming analytics. In: 2021 IEEE 37th International Conference on Data Engineering (ICDE). pp. 864–875. IEEE (2021)
5. Booij, T.M., Chiscop, I., Meeuwissen, E., Moustafa, N., Den Hartog, F.T.: Ton_iot: The role of heterogeneity and the need for standardization of features and attack types in iot network intrusion data sets. IEEE Internet of Things Journal **9**(1), 485–496 (2021)
6. Bosshart, P., Daly, D., Gibb, G., Izzard, M., McKeown, N., Rexford, J., Schlesinger, C., Talayco, D., Vahdat, A., Varghese, G., et al.: P4: Programming protocol-independent packet processors. ACM SIGCOMM Computer Communication Review **44**(3), 87–95 (2014)
7. Boutros, A., Betz, V.: FPGA architecture: Principles and progression. IEEE Circuits and Systems Magazine **21**(2), 4–29 (2021)
8. Budiu, M., Dodd, C.: The p416 programming language. ACM SIGOPS Operating Systems Review **51**(1), 5–14 (2017)
9. Cormode, G., Muthukrishnan, S.: An improved data stream summary: the count-min sketch and its applications. Journal of Algorithms **55**(1), 58–75 (2005)
10. Datta, S., Venkanna, U.: A traffic prioritization framework for smart home iot networks using programmable data planes. In: 2023 IEEE Globecom Workshops (GC Wkshps). pp. 371–376. IEEE (2023)
11. Feist, T.: Vivado design suite. White Paper **5**(30), 24 (2012)
12. Huang, H., Yu, J., Du, Y., Liu, J., Dai, H., Sun, Y.E.: Memory-efficient and flexible detection of heavy hitters in high-speed networks. Proceedings of the ACM on Management of Data **1**(3), 1–24 (2023)
13. Huang, J., Zhang, W., Li, Y., Li, L., Li, Z., Ye, J., Wang, J.: Chainsketch: An efficient and accurate sketch for heavy flow detection. IEEE/ACM Transactions on Networking **31**(2), 738–753 (2022)
14. Kuzniar, C., Neves, M., Gurevich, V., Haque, I.: Poiriot: Fingerprinting iot devices at tbps scale. IEEE/ACM Transactions on Networking (2024)
15. Li, W., Li, Z., Bütün, B., Diallo, A.F., Fiore, M., Patras, P.: Pontus: A memory-efficient and high-accuracy approach for persistence-based item lookup in high-velocity data streams. In: Proceedings of the ACM on Web Conference 2025. pp. 1783–1794 (2025)

16. Li, W., Patras, P.: P-sketch: A fast and accurate sketch for persistent item lookup. IEEE/ACM Transactions on Networking **32**(2), 987–1002 (2023)
17. Li, W., Patras, P.: Tight-sketch: A high-performance sketch for heavy item-oriented data stream mining with limited memory size. In: Proceedings of the 32nd ACM International Conference on Information and Knowledge Management. pp. 1328–1337 (2023)
18. Li, W., Patras, P.: Stable-sketch: A versatile sketch for accurate, fast, web-scale data stream processing. In: Proceedings of the ACM on Web Conference 2024. pp. 4227–4238 (2024)
19. Li, Y., Yu, X., Yang, Y., Zhou, Y., Yang, T., Ma, Z., Chen, S.: Pyramid family: Generic frameworks for accurate and fast flow size measurement. IEEE/ACM Transactions on Networking **30**(2), 586–600 (2021)
20. Liu, J., Basat, R.B., De Wardt, L., Dai, H., Chen, G.: Disco: A dynamically configurable sketch framework in skewed data streams. In: 2024 IEEE 40th International Conference on Data Engineering (ICDE). pp. 4801–4814. IEEE (2024)
21. Sivaraman, V., Narayana, S., Rottenstreich, O., Muthukrishnan, S., Rexford, J.: Heavy-hitter detection entirely in the data plane. In: Proceedings of the Symposium on SDN Research. pp. 164–176 (2017)
22. Tang, L., Huang, Q., Lee, P.P.: A fast and compact invertible sketch for network-wide heavy flow detection. IEEE/ACM Transactions on Networking **28**(5), 2350–2363 (2020)
23. Tang, M., Wen, M., Shen, J., Zhao, X., Zhang, C.: Towards memory-efficient streaming processing with counter-cascading sketching on fpga. In: 2020 57th ACM/IEEE Design Automation Conference (DAC). pp. 1–6. IEEE (2020)
24. Tong, D., Prasanna, V.K.: Sketch acceleration on fpga and its applications in network anomaly detection. IEEE Transactions on Parallel and Distributed Systems **29**(4), 929–942 (2017)
25. Wei, C., Li, X., Yang, Y., Jiang, X., Xu, T., Yang, B., Wu, T., Xu, C., Lv, Y., Gao, H., et al.: Achelous: Enabling programmability, elasticity, and reliability in hyperscale cloud networks. In: Proceedings of the ACM SIGCOMM 2023 Conference. pp. 769–782 (2023)
26. Xu, W., Zhang, Z., Song, H., Liu, S., Feng, Y., Liu, B.: Isac: In-switch approximate cache for iot object detection and recognition. In: IEEE INFOCOM 2023-IEEE Conference on Computer Communications. pp. 1–10. IEEE (2023)
27. Yang, T., et. al: Elastic sketch: Adaptive and fast network-wide measurements. In: Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication. pp. 561–575 (2018)
28. Yang, T., Zhang, H., Li, J., Gong, J., Uhlig, S., Chen, S., Li, X.: Heavykeeper: an accurate algorithm for finding top-k elephant flows. IEEE/ACM Transactions on Networking **27**(5), 1845–1858 (2019)
29. Ye, J., Li, L., Zhang, W., Chen, G., Shan, Y., Li, Y., Li, W., Huang, J.: Ua-sketch: an accurate approach to detect heavy flow based on uninterrupted arrival. In: Proceedings of the 51st International Conference on Parallel Processing. pp. 1–11 (2022)
30. Zhang, Y., Liu, Z., Wang, R., Yang, T., Li, J., Miao, R., Liu, P., Zhang, R., Jiang, J.: Cocosketch: High-performance sketch-based measurement over arbitrary partial key query. In: Proceedings of the 2021 ACM SIGCOMM 2021 Conference. pp. 207–222 (2021)