POLITECNICO DI TORINO

POLITECNICO DI TORINO

Master Degree course in Communications Engineering

Master Degree Thesis

# Performance Evaluation of HyperLogLog for In-Network Flow Cardinality Estimation on P4-Programmable Switches

**Supervisors**
Prof. Paolo Giaccone
Prof. Andrea Bianco

**Candidate**
Amanj Malaei

Academic Year 2025-2026

# Acknowledgements

My deepest gratitude goes to Prof. Paolo Giaccone, whose guidance, encouragement, and dedication made this thesis possible. I am also thankful to Prof. Andrea Bianco for his supervision throughout this research. Special thanks to Alessandro Cornacchia, whose patience and enthusiasm kept me motivated throughout the ups and downs of this work. My thanks also go to Zhihao Wang for his helpful contributions along the way. Finally, to my parents and my dear brother Arman—your love and support mean everything to me.

**Abstract**

Network traffic monitoring requires efficient cardinality estimation to count distinct flows for applications such as anomaly detection and traffic engineering. Traditional exact counting approaches are memory-intensive and do not scale to high-speed networks. Probabilistic algorithms like HyperLogLog (HLL) offer memory-efficient alternatives by trading exact accuracy for space efficiency.

This thesis presents a performance evaluation of HyperLogLog for in-network flow cardinality estimation on P4-programmable switches. We implement an optimized HLL variant on Intel Tofino hardware using the $P4_{16}$ programming language, focusing on developing a robust monitoring framework for systematic performance assessment under both synthetic and real network traffic.

We design an experimental methodology that includes: (1) a monitoring infrastructure with exponential random sampling for state observation, (2) ground truth validation using hardware counters, (3) synthetic traffic generation at controlled rates, and (4) trace replay capabilities for real-world traffic patterns. The system supports configurable monitoring intervals and deterministic reset periods ensuring accurate comparison between HLL estimates and ground truth.

This work demonstrates the practical viability of probabilistic cardinality estimation in programmable network hardware, providing empirical insights into accuracy–performance trade-offs and establishing a reproducible methodology for evaluating sketch-based algorithms in production network environments.

# Contents

# Chapter 1

# Introduction

The past twenty years have seen a significant change in network infrastructure. This change has been driven by the rapid growth of cloud computing and a constant need for connectivity. As hyperscale data centers have spread around the world to support various services—such as video streaming, social networking, financial transactions, and scientific computing—the networks linking these centers have become much larger and more complicated. Service providers managing these infrastructures face a major challenge. They need to maintain an understanding of traffic flow, track how many connections are active at any moment, and know where resources are being used. This understanding is crucial for functions like detecting anomalies, planning capacity, balancing loads, and meeting service agreements. However, the size and speed of modern networks make traditional monitoring methods less effective.

Counting distinct flows highlights the conflict between operational needs and practical limitations. Accurately counting flows requires tracking every individual flow, which uses memory based on the number of unique items. Given the data rates typical of today's networks—hundreds of gigabits per second with millions of active flows—this method demands an impractical amount of memory and cannot keep up with processing speeds. Researchers have long recognized this issue and have developed probabilistic data structures that sacrifice perfect accuracy for significant reductions in memory use. One such structure, HyperLogLog, is a particularly elegant solution. It uses only kilobytes of memory to estimate counts in the billions, with a controlled relative error, regardless of the actual count. Initially designed for database analytics, it has been adopted in production systems at companies like Google and Redis. HyperLogLog offers the right balance of accuracy and efficiency needed for network monitoring.

As researchers have made strides with probabilistic algorithms, network hardware has also quietly evolved. Traditional network equipment—like routers and switches with fixed functions—has been replaced by programmable data planes. These data planes allow operators to control packet processing logic. The development of Protocol-Independent Switch Architecture (PISA) and the P4 programming language has enabled experts to create custom packet processing pipelines directly in switching hardware, maintaining the same high throughput as fixed-function devices while allowing for different protocols and in-network calculations. Intel Tofino switches, which apply this approach, include stateful

memory that persists across packets. This feature makes it possible to run probabilistic algorithms like HyperLogLog entirely within the data plane, at full speed, without needing the control plane to process each packet.

This thesis looks into how practical it is to implement HyperLogLog for estimating flow cardinality on P4-programmable switches. We create a complete system called Windowed HyperLogLog (W-HLL). This system consists of a P4 data plane program running on Intel Tofino hardware and a Python control plane that coordinates regular observations and resets of the sketch state.

Beyond the implementation, this work sets up a formal experimental method for testing sketch-based algorithms on network hardware. We understand that simple periodic sampling can lead to systematic bias when traffic shows patterns over time. To address this, we use exponentially distributed read intervals based on the PASTA theorem from queueing theory. We validate our findings through two complementary scenarios: synthetic traffic that establishes controlled ground truth and real network data captured from the Politecnico di Torino campus network. Our experiments cover several window configurations from five to one hundred seconds, allowing us to systematically assess the accuracy-resolution trade-off.

Our results show that in-network HyperLogLog performs as well as theory predicts. With two thousand buckets using just two kilobytes of memory, our implementation achieves relative errors that align with theoretical expectations across various traffic conditions. These results confirm that probabilistic cardinality estimation is not just theoretical, but a practical tool for real-world network monitoring.

The rest of this thesis is organized as follows. Chapter 2 provides the technical foundation needed to understand our implementation. It covers the development of network programmability from Software-Defined Networking to PISA-based data planes, the P4 language and runtime environment, the HyperLogLog algorithm and its performance, and the PASTA theorem shaping our sampling strategy. Chapter 3 details the design and implementation of Windowed HyperLogLog, including both the P4 data plane setup and the Python control plane managing monitoring tasks. Chapter 4 discusses our experimental setup on the SUPERNET testbed at Politecnico di Torino, describes the methods for measuring ground truth and analyzing errors, and presents comprehensive results for both synthetic and real traffic. The thesis wraps up with a summary of contributions and future research directions.

# Chapter 2

# Background

This chapter lays the groundwork needed to understand our Windowed HyperLogLog (W-HLL) implementation on Intel Tofino programmable switches. We start with a look at how network programmability has changed from traditional Software-Defined Networking to data plane programmability in Section 2.1. Next, we explain the P4 language and runtime environment, which allows us to express packet processing logic in simple terms in Section 2.2. Finally, we introduce the HyperLogLog algorithm for estimating cardinality with a probabilistic approach in Section 2.3. This algorithm serves as the basis for our flow counting implementation.

## 2.1 Network Programmability

Network infrastructure has gone through several distinct phases in how much control operators actually have over their systems. In traditional networks, the control and data planes were tightly coupled inside proprietary hardware, which made it difficult to adapt to new requirements or experiment with different approaches [10, 22]. Software-Defined Networking changed this by separating the two planes—control logic moved to a central location, though the data plane itself remained fixed in what it could do [20]. The final piece came with programmable data planes, enabled by architectures like RMT [4] and the P4 language [3], which made it possible to redefine how packets are actually processed.

### 2.1.1 Software-Defined Networking and Its Limitations

Software-Defined Networking brought a clean separation between control and data planes [22]. Under this model, switches act as programmable forwarding elements that maintain flow tables—each entry specifying what to match on (header fields) and what to do about it (forward, drop, modify). A logically centralized controller keeps track of the overall network state, figures out the forwarding behavior, and pushes rules down to switches using protocols like OpenFlow [25]. This setup offered real benefits: having control in one place avoids the headaches of distributed protocol convergence [20], policies can be rolled out quickly through high-level programming, and standardized interfaces mean you're not locked into a single vendor [10].

That said, SDN only made the control plane programmable—the data plane stayed essentially fixed. OpenFlow switches can only match on fields and perform actions that are baked into their specifications. When new protocols come along (VXLAN, GENEVE, custom headers), the whole ecosystem has to catch up: specification updates, ASIC redesigns that take one to two years, and eventually replacing hardware across the network [3]. On top of that, OpenFlow offers stateless match-action forwarding with basic counters, but nothing more sophisticated. There are no register arrays, no atomic read-modify-write operations like increment or max, and no way to run custom computations beyond the predefined set [4, 31]. This rules out any network function that needs to update state on a per-packet basis—things like sketches for traffic measurement [34], stateful firewalls that track TCP connections [24], or token bucket rate limiters [2].

### 2.1.2 Programmable Data Planes and Intel Tofino

The Reconfigurable Match-Action Table (RMT) architecture [4] introduced a flexible pipeline built from identical stages that can be configured when the switch is deployed. Traditional ASICs dedicate each stage to a specific function, but RMT stages are generic—any stage can perform table lookups (TCAM, exact match, longest prefix match) on whatever packet fields you need, execute actions like modifying headers or updating registers or doing arithmetic, and then hand the packet off to the next stage. At the heart of RMT is the Packet Header Vector (PHV), essentially a wide register file that holds the extracted headers. The parser fills in the PHV, match-action stages read and modify its fields, and the deparser turns it back into an actual packet. By separating packet processing (which works on the PHV) from storage (the packet buffer), this design allows efficient memory access and the freedom to modify headers however you want [4].

The Protocol-Independent Switch Architecture (PISA) builds on RMT by defining the full switch architecture [3]. It includes programmable parsers that extract headers using configurable state machines (capable of handling arbitrary protocol sequences), separate ingress and egress pipelines with their own match-action stages, and programmable deparsers that reassemble packets from the modified headers. The "protocol-independent" label reflects something important: the architecture itself assumes nothing about what packets look like—that's entirely up to the program you load [3, 13]. Adding support for a new protocol becomes a matter of compiling and loading a new configuration, not redesigning silicon.

Intel Tofino [19] was the first commercial switch ASIC to put RMT and PISA into practice, achieving 6.5 Tbps of aggregate throughput with sub-microsecond latency for packet processing. The chip organizes its work across multiple parallel pipelines, each responsible for a share of the total port bandwidth. Figure 2.1 illustrates the architecture of a 6.5 Tbps Tofino chip, showing its four identical processing pipelines. Each pipeline implements the full PISA model. The ingress path handles parsing to extract headers, runs match-action processing in the control block, and reconstructs packets in the deparser. The traffic manager sits in the middle, buffering packets, handling replication, managing queues, and scheduling when things get sent out. Finally, the egress path re-parses headers, applies any port-specific operations, and finalizes the packet structure. The port blocks are flexible in how they can be configured—you can run 100 Gigabit interfaces,

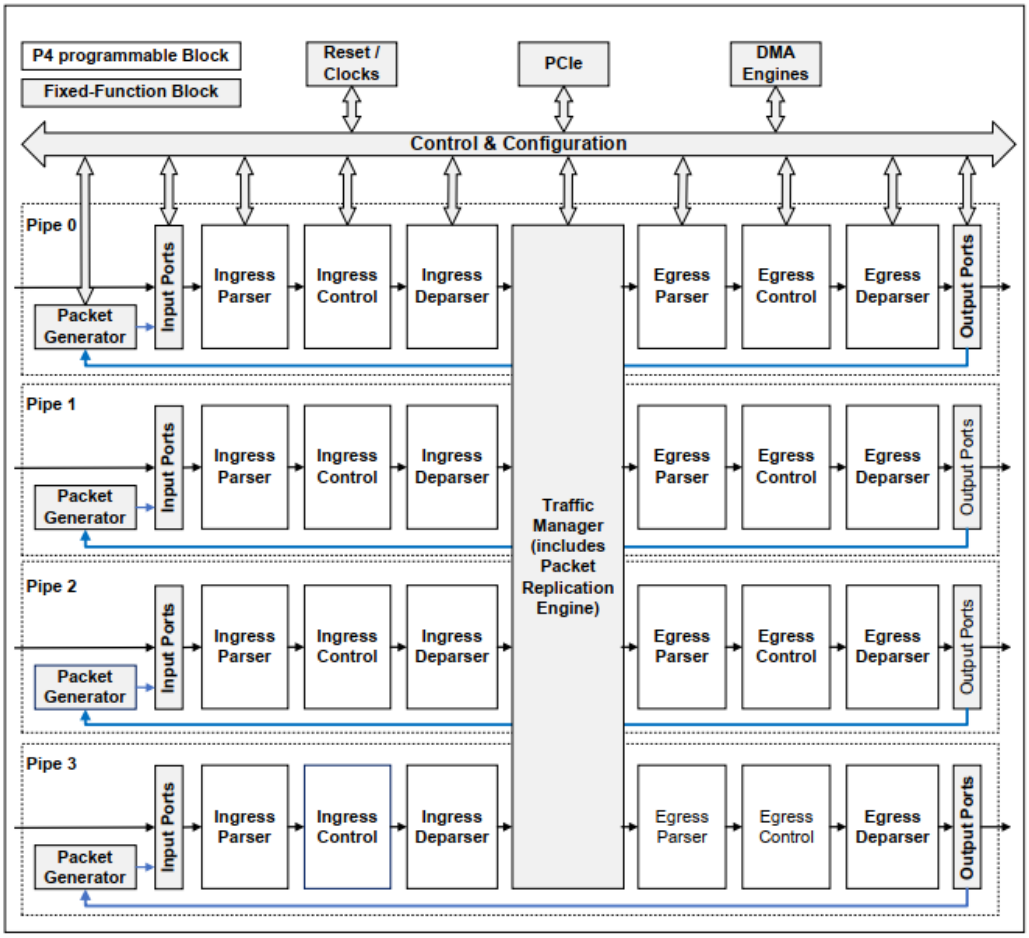


Figure 2.1. Intel Tofino block diagram showing four parallel processing pipelines. Each pipeline contains ingress and egress stages with programmable parsers, match-action control blocks, and deparsers. Packets flow through ingress processing, the traffic manager (which handles buffering, queueing, and replication), and egress processing before transmission (adapted from [19]).

split them into four independent 25 Gigabit channels, or use several other modes. One pipeline also connects to a CPU over PCIe for control plane communication [19].

For implementing sketch-based algorithms, Tofino provides some essential building blocks [30]. *Registers* give you persistent memory arrays that can be accessed across packets. *RegisterAction* enables atomic read-modify-write operations, which is critical for avoiding race conditions. And *counters* provide hardware-accelerated 64-bit tracking of both packet counts and byte counts. When it comes to runtime configuration, BFRuntime exposes gRPC-based APIs for programming tables, reading and writing registers, and pulling statistics.

## 2.2 The P4 Language and Runtime

Programmable hardware provides the foundation for flexible packet processing, but taking advantage of that flexibility requires the right abstractions. The P4 language [3] and the P4Runtime API [26] together form a complete programming environment—P4 lets you define data plane behavior at a high level, while P4Runtime handles runtime control through a standardized interface that doesn't depend on any particular protocol.

### 2.2.1 Programming Protocol-Independent Packet Processors

P4 was created to address a fundamental limitation of SDN: OpenFlow made the control plane programmable, but the data plane remained fixed [3]. P4 flips this around—rather than programming table entries on switches with predetermined functions, P4 programs define the tables, parsers, and processing logic themselves. Figure 2.2 shows where P4 fits in the SDN architecture. Traditional SDN uses OpenFlow to populate forwarding tables on fixed-function switches, whereas P4 works at a higher level by configuring what the switch actually does.

The language is built on three core principles [1]. *Reconfigurability* means operators can change how packets are parsed and processed on deployed switches without swapping out hardware. *Protocol independence* allows switches to handle arbitrary packet formats—whatever the P4 program defines. And *target independence* keeps programs portable across different hardware platforms, with vendor-supplied compilers handling the translation. Figure 2.3 shows the contrast between traditional switch architectures and P4-programmable ones.

P4 offers a set of fundamental abstractions for describing how packets should be processed [1,3]. *Headers* define the structure of packet fields. *Parsers* specify state machines that extract those headers. *Tables* map user-defined keys to actions. *Actions* describe how packets get transformed. *Control flow* determines the sequence of processing through the pipeline. And *extern objects* provide access to architecture-specific features—hash functions, checksums, registers—through well-defined APIs. Figure 2.4 illustrates this abstract forwarding model: packets pass through programmable parsers, then through multiple match-action stages, and finally through programmable deparsers.

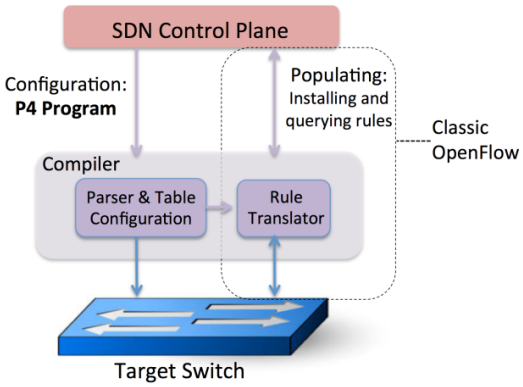


Figure 2.2. P4's role in network programmability. While traditional SDN APIs like OpenFlow populate predefined forwarding tables, P4 programs configure the switch data plane itself—defining parsers, tables, and actions. The control plane then manages the P4-defined entities through generated APIs (adapted from [3]).

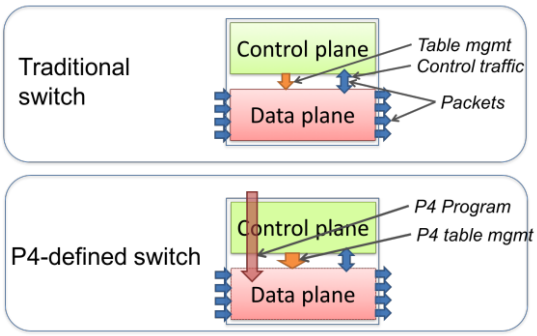


Figure 2.3. Comparison of traditional and P4-programmable switch architectures. Traditional switches implement fixed data plane functions defined by manufacturers; the control plane manages predefined tables through standard protocols. P4-programmable switches have no built-in protocol knowledge—the loaded P4 program defines all data plane functionality, and the control plane accesses dynamically generated APIs corresponding to program-defined entities (adapted from [1]).

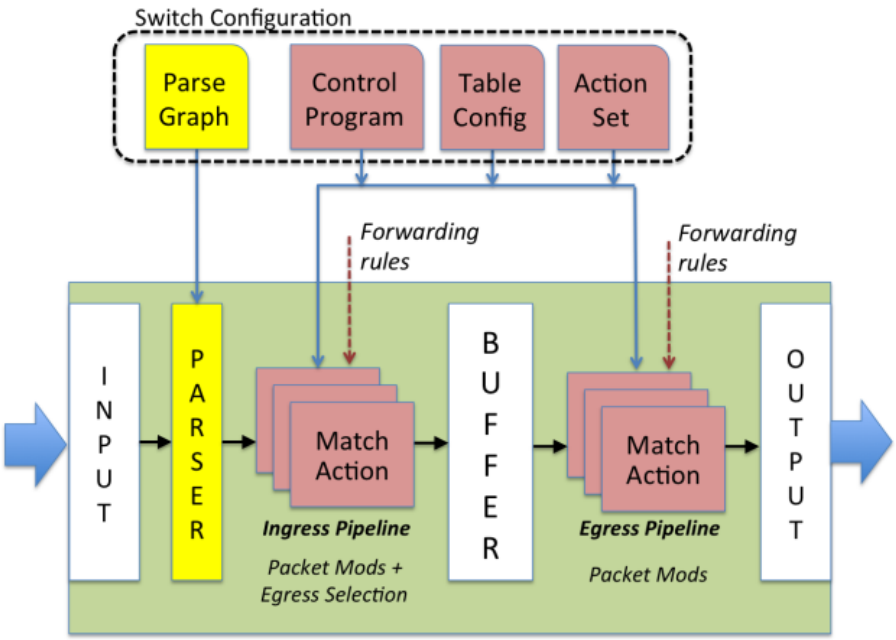


Figure 2.4. P4 abstract forwarding model showing packet flow through programmable components. The parser extracts headers into a structured representation; match-action stages in the ingress pipeline process packets and determine forwarding; the traffic manager handles buffering and queueing; egress match-action stages perform port-specific operations; the deparser reconstructs packets before transmission (adapted from [3]).

A P4 program describes packet processing for a specific target architecture. Manufacturers supply an architecture definition that specifies what components are available—parsers, pipelines, deparsers—and how they interface with each other. Programmers then write P4 code that conforms to this architecture, defining the headers, parsers, match-action logic, and control flow. Figure 2.5 shows the complete workflow from writing a P4 program to having it run on a switch.

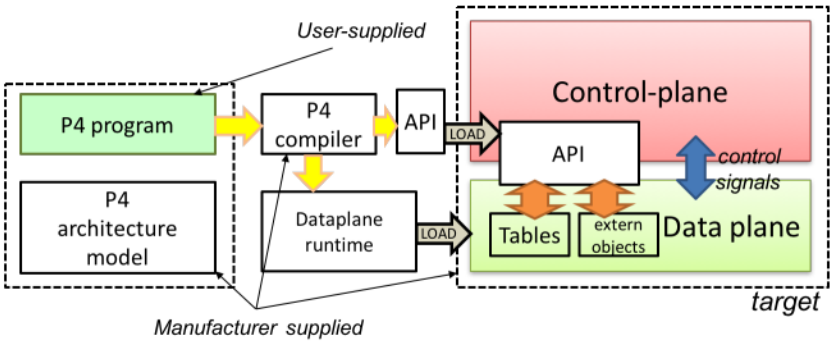


Figure 2.5. P4 programming workflow showing the relationship between P4 program, architecture model, compiler, and target device. The programmer writes target-independent P4 code; the vendor-supplied compiler translates this to target-specific configuration and generates control plane APIs; the control plane manages runtime state through these APIs while the data plane executes compiled packet processing logic (adapted from [1]).

The $P4_{16}$ specification [1] was a significant redesign from the original $P4_{14}$. It cut the keyword count from over 70 down to under 40 by moving functionality into libraries, introduced modular constructs (`parser`, `control`, `package`, `extern`) for describing both programmable and fixed components, and was designed with forward compatibility in mind for long-term program development [1]. Compared to programming hardware directly—writing microcode for custom ASICs, for example—P4 brings substantial advantages: it abstracts away hardware-specific details, makes programs portable across different platforms (Tofino, software switches, FPGAs), lets compilers handle resource management automatically, and remains expressive enough for complex algorithms. You also get the software engineering benefits you'd expect—type checking, modularity—along with reusable libraries from vendors and the ability to debug using software switch implementations [3, 31].

### 2.2.2 P4Runtime: Control Plane Integration

P4 defines what the data plane does, but network applications also need runtime control—installing forwarding rules, reading counters, querying statistics, and managing device state. P4Runtime [26] provides a standardized API for the control plane to interact with P4-programmed switches. The API has several important properties: it's *protocol independent*, meaning it adapts to whatever functionality the loaded P4 program defines; it's *target independent*, working across hardware ASICs, FPGAs, and software switches; it

offers *control plane flexibility*, supporting embedded, remote, or hybrid architectures; and it enables *multi-controller support*, where role-based arbitration lets multiple controllers connect while designating one as master for write operations [26].

Figure 2.6 shows the P4Runtime reference architecture with multi-controller support. The target device runs the P4 data plane, and one or more controllers manage its operation over gRPC. The P4Runtime architecture uses Protocol Buffers [15] for message

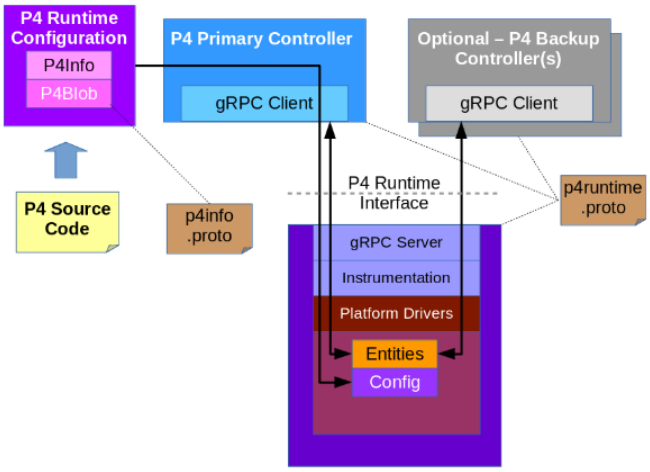


Figure 2.6. P4Runtime API reference architecture showing multi-controller support. Controllers communicate with the target device via gRPC. The P4 compiler generates P4Info metadata describing all controllable entities; controllers use this metadata to interact with tables, registers, counters, and other objects defined in the loaded P4 program. Role-based arbitration ensures only one master controller has write access to each entity type (adapted from [26]).

serialization and gRPC [14] for transport. The key components include the P4Runtime API itself, defined in the `p4runtime.proto` specification; P4Info metadata that compilers generate to describe all controllable entities (table names, match fields, action parameters, register dimensions); ForwardingPipelineConfig, which bundles compiled programs together with their metadata; and gRPC servers on the target devices that translate P4Runtime operations into hardware-specific commands [26].

P4Runtime supports a range of deployment scenarios. Figure 2.7 shows an embedded controller configuration, where the control plane runs directly on the switch CPU and communicates with the on-board ASIC through P4Runtime over a local gRPC connection. The core operations cover table management (inserting, modifying, and deleting match-action entries), register access (reading and writing stateful memory arrays), counter queries (retrieving packet and byte counts), pipeline configuration (installing or querying the ForwardingPipelineConfig), and packet I/O (sending and receiving packets between the control and data planes) [26]. When multiple controllers are connected, arbitration mechanisms keep state management consistent: each controller declares its role, the system designates one master controller per entity type, and only masters can perform writes while all controllers can read [26].

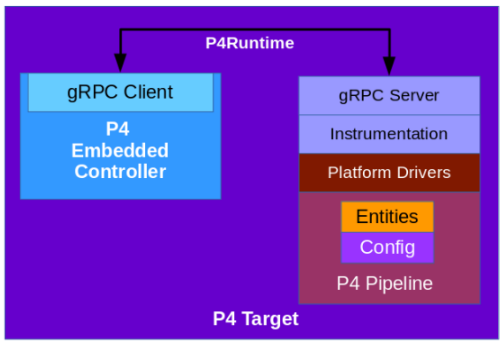


Figure 2.7. Single embedded controller architecture. The P4 embedded controller runs on the switch CPU and communicates with the switch ASIC via P4Runtime over a local gRPC connection. This configuration is suitable for appliances not requiring external SDN control, where P4Runtime serves as an efficient embedded API for managing the local data plane (adapted from [26]).

## 2.3 Cardinality Estimation

Network monitoring often needs to count distinct elements in large data streams—unique source IP addresses, active TCP connections, or distinct flow identifiers [9, 21]. Doing this exactly with hash tables becomes prohibitively expensive at scale; tracking millions of flows can require gigabytes of memory. Probabilistic cardinality estimation algorithms offer a practical alternative, trading perfect accuracy for dramatic reductions in space and making measurement feasible in high-speed networks [11].

### 2.3.1 Counting Distinct Elements Problem

Given a stream of elements $S = \{x_1, x_2, \ldots, x_n\}$ where elements may repeat, the goal is to estimate how many unique elements there are—this quantity is called the cardinality $n$. We want a method that processes each element as it arrives in a single pass, uses far less memory than storing every element, and produces an estimate $\hat{n}$ close to the true value: $|\hat{n} - n|/n \leq \epsilon$ with high probability [11].

Cardinality estimation addresses a range of network measurement challenges [9,21,34]. *Flow counting* estimates how many active TCP/UDP flows (unique 5-tuples) are present, which matters for capacity planning and anomaly detection. *Source diversity* counts the distinct source IPs contacting a server, useful for detecting DDoS attacks or flash crowds. *Spread estimation* measures how many distinct destinations a given source contacts, helping identify scanning behavior. *Traffic matrix estimation* counts flows between origin-destination pairs to inform traffic engineering decisions. And *protocol distribution* estimates how many distinct applications are in use, guiding QoS policies. Doing any of this exactly would mean maintaining sets of all observed identifiers—something that's simply not feasible at line rate when you're dealing with millions of flows [9].

### 2.3.2 The HyperLogLog Algorithm

HyperLogLog (HLL) [11] is a probabilistic cardinality estimator with remarkable space efficiency: it can estimate cardinalities up to billions using only kilobytes of memory, with typical error rates around 2%. The algorithm takes advantage of the statistical properties of hash functions. A uniform hash function maps elements to bit strings where every bit pattern is equally likely. Now consider the position of the first '1' bit—called the *rank* or *level*. Patterns with $k$ leading zeros occur with probability $1/2^{k+1}$. So if we observe a hash with rank $\rho = k$, we can infer that roughly $2^k$ distinct elements have been hashed—seeing rare patterns (high ranks) suggests many observations [11]. Figure 2.8 illustrates this concept. A single maximum, however, makes for a poor estimator—the variance is too high. HyperLogLog improves accuracy through *stochastic averaging* [11]: divide the input space into $m$ buckets, keep track of the maximum rank independently for each bucket, and then combine these estimates using the harmonic mean to bring down the variance. The algorithm works as follows: start by initializing an array of $m$ registers $M[0], \ldots, M[m-1] \leftarrow 0$. For each element $x$, compute its hash $h(x)$, extract the bucket index $j$ from the leftmost $b$ bits (where $m = 2^b$), compute the rank $\rho(w)$ from the remaining bits as the position of the first '1' bit, and update $M[j] \leftarrow \max(M[j], \rho(w))$ [11].

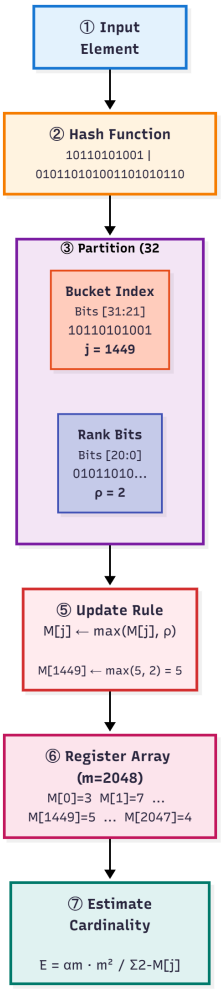


Figure 2.8. HyperLogLog conceptual workflow. Input elements are hashed to 32-bit values, partitioned into bucket index (11 bits) and rank bits (21 bits). The rank $\rho$ represents the position of the first '1' bit—high ranks occur rarely (probability $1/2^{\rho}$) and indicate many distinct observations. Each bucket maintains the maximum rank observed. The final cardinality estimate combines all bucket values using the harmonic mean, providing robustness against outlier buckets with unusually high ranks.

Table 2.1. HyperLogLog Algorithm (Pseudocode)

| |
|---|
| **Input:** Stream $S = \{x_1, x_2, \ldots, x_n\}$, number of buckets $m = 2^b$<br>**Output:** Cardinality estimate $\hat{n}$ |
| 1. Initialize $M[0 \ldots m-1] \leftarrow 0$<br>2. **For each** $x$ in $S$:<br>a. $h \leftarrow \text{hash}(x)$<br>b. $j \leftarrow$ leftmost $b$ bits of $h$ (bucket index)<br>c. $w \leftarrow$ remaining bits of $h$<br>d. $\rho \leftarrow$ position of leftmost '1' in $w$ (rank)<br>e. $M[j] \leftarrow \max(M[j], \rho)$<br>3. $Z \leftarrow \sum_{j=0}^{m-1} 2^{-M[j]}$<br>4. $E_{\text{raw}} \leftarrow \alpha_m \cdot m^2 / Z$<br>5. **If** $E_{\text{raw}} \leq 2.5m$ and $V > 0$ (empty buckets):<br>$E \leftarrow m \cdot \log(m/V)$ (small range correction)<br>**Else if** $E_{\text{raw}} > 2^L/30$:<br>$E \leftarrow -2^L \cdot \log(1 - E_{\text{raw}}/2^L)$ (large range correction)<br>**Else:** $E \leftarrow E_{\text{raw}}$<br>6. **Return** $E$ |

Once the stream has been processed, the cardinality is estimated using the harmonic mean [11]:

$$Z = \sum_{j=0}^{m-1} 2^{-M[j]} \tag{2.1}$$

$$E_{\text{raw}} = \alpha_m \cdot m^2 \cdot \frac{1}{Z} \tag{2.2}$$

where $\alpha_m$ is a bias correction constant that depends on $m$. For large $m$, it's approximately $\alpha_m \approx 0.7213/(1+1.079/m)$. The harmonic mean provides robustness against outliers—buckets with unusually high values contribute less weight to the final estimate [11].

The raw estimate is biased at extreme ranges and needs correction [17]. When $E_{\text{raw}} \leq 2.5m$ and there are empty buckets (count $V > 0$), a small range correction is applied: $E = m \cdot \log(m/V)$, based on linear counting [9]. When $E_{\text{raw}} > 2^L/30$ for $L$-bit hashes, a large range correction handles hash collisions: $E = -2^L \cdot \log(1 - E_{\text{raw}}/2^L)$ [17]. Figure 2.9 shows how these corrections keep the bias near zero across the full range of cardinalities. Table 2.3.2 summarizes the complete HyperLogLog algorithm in pseudocode form.

### 2.3.3 Performance Characteristics of HyperLogLog

HyperLogLog with $m$ buckets and $\ell$-bit registers uses $m \cdot \ell$ bits of memory—a constant amount regardless of how large the stream cardinality gets [11]. The relative standard error is $\sigma_{\text{HLL}} = 1.04/\sqrt{m}$, which doesn't depend on the actual cardinality [11]. With $m = 2048$ buckets and 8-bit registers, for example, you get 2 KB of memory usage

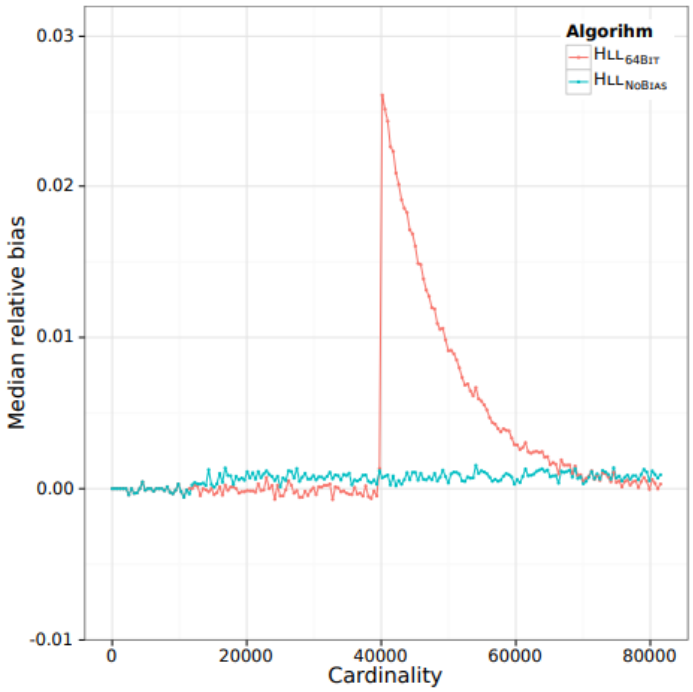


Figure 2.9. HyperLogLog median relative bias versus true cardinality. The raw HLL estimator exhibits significant bias peaking around intermediate cardinalities, while the bias-corrected estimator maintains near-zero bias across all ranges through empirical corrections (reproduced from [17]).

and roughly 2.3% error. This *scale invariance* sets HLL apart from sampling-based approaches, where error tends to grow with cardinality [9, 21].

The accuracy of HyperLogLog depends heavily on the quality of the hash function [11]. It needs uniformity (hash values spread evenly), independence (different inputs produce independent hashes), and the avalanche property (small changes in input lead to drastically different hashes). Common choices include MurmurHash3 and xxHash for software implementations, or hardware-accelerated CRC32 for switches [17]. One particularly useful property of HyperLogLog is that sketches can be merged: given two HLL sketches $\text{HLL}_A$ and $\text{HLL}_B$, you can compute $\text{HLL}_{A \cup B}$ simply by taking the element-wise maximum, $M_{A \cup B}[j] = \max(M_A[j], M_B[j])$ [11]. This makes distributed cardinality estimation possible across multiple switches without needing to exchange raw data.

HyperLogLog grew out of earlier probabilistic counting methods [7, 12]. Flajolet-Martin introduced probabilistic counting based on bit patterns [12]. Linear Probabilistic Counting used bit vectors but required $O(n)$ space. LogLog brought in stochastic averaging, though it relied on the arithmetic mean [7]. HyperLogLog's switch to the harmonic mean dramatically improved accuracy [11]. Later, HyperLogLog++ [17] added improved empirical bias correction and a sparse representation, and is now deployed in production systems like Google BigQuery, Redis, and Elasticsearch.

In short, HyperLogLog offers a practical solution for cardinality estimation in resource-constrained, high-throughput environments. A few kilobytes can estimate billions of elements, the error is bounded and independent of cardinality, the operations are simple enough for hardware implementation, sketches can be merged for distributed estimation, and single-pass constant-time updates allow line-rate operation [11, 17]. These properties make HLL well suited for implementation on programmable switches, where memory is limited to megabytes of SRAM and per-packet processing has to fit within pipeline stage constraints measured in nanoseconds [31, 34].

## 2.4 Unbiased Sampling via PASTA

Accurate network monitoring requires periodically reading statistics from data plane registers, but how you decide when to sample has a significant impact on measurement accuracy. This section examines why naive approaches tend to fail and how the PASTA property (Poisson Arrivals See Time Averages) leads to unbiased observation.

### 2.4.1 The Problem with Deterministic Sampling

A natural approach to periodic monitoring is fixed-interval sampling—reading registers every $\Delta t$ seconds. The problem with this deterministic strategy is that it risks *phase-locking* with periodic traffic patterns, consistently observing the same phase of the system rather than capturing all phases representatively [29, 32].

Figure 2.10 illustrates this phenomenon. Consider periodic bursty traffic that alternates between high-rate bursts and low-rate idle periods. If the sampling interval happens to align with the traffic period, observations will systematically capture only one phase—always the idle periods, for instance—which severely biases the estimated

average. This synchronization problem is especially concerning in networks, where traffic often follows diurnal patterns, periodic application behavior, or regular protocol timers.

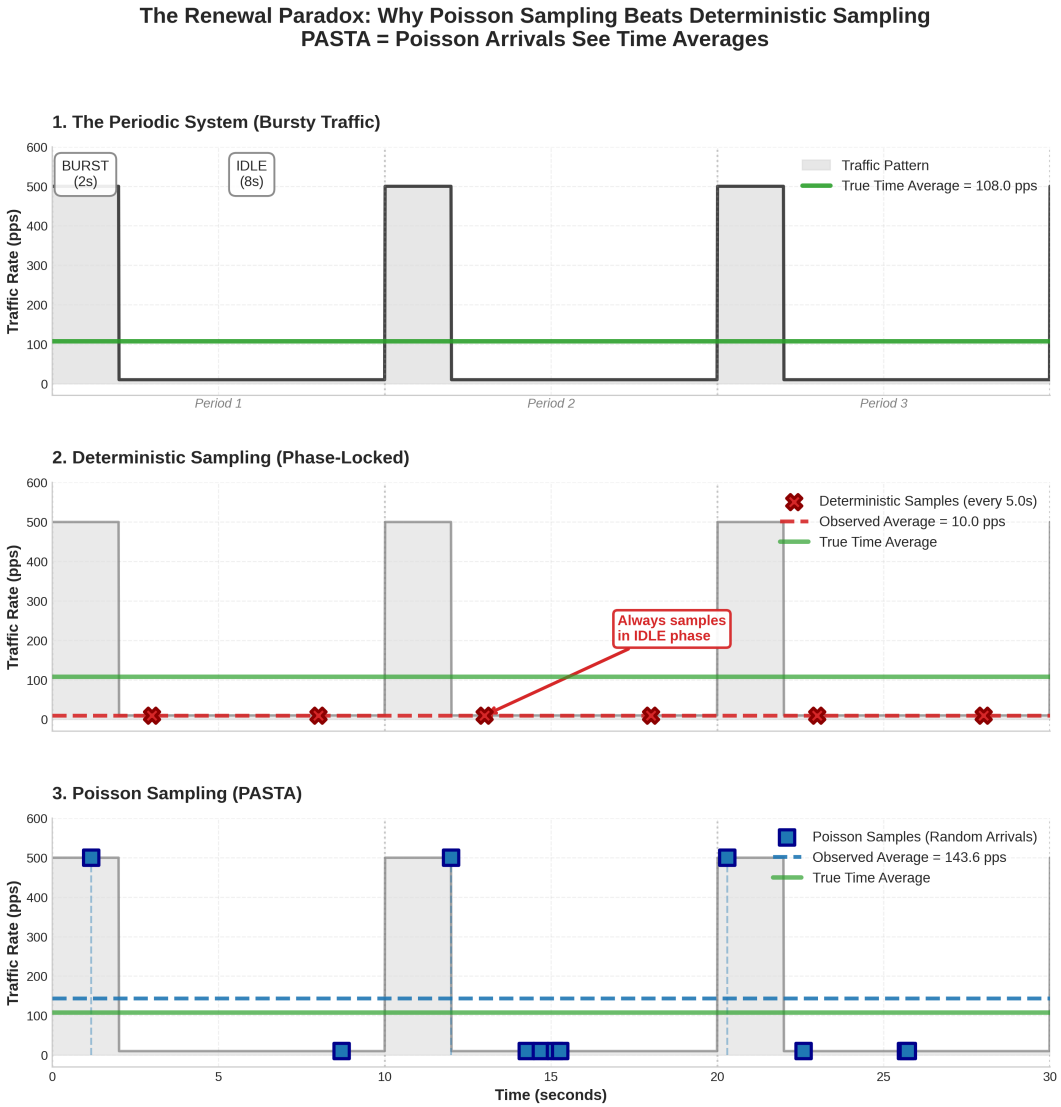


Figure 2.10. Deterministic sampling can synchronize with periodic traffic patterns, leading to biased observations that capture only specific phases rather than representative system behavior.

The fundamental issue is that deterministic sampling effectively "anticipates" when its future observations will happen. When those times correlate with periodicity in the system, the observations no longer reflect the true time-average behavior.

### 2.4.2 Theoretical Foundation: PASTA

The solution to the phase-locking problem comes from a classical result in queueing theory. The PASTA property, established by Wolff [32], states that for Poisson arrivals, the fraction of arrivals that find a system in a particular state equals the fraction of time the system spends in that state.

More formally, consider a stochastic process $\{N(t), t \geq 0\}$ representing the system state, and let $\{A(t), t \geq 0\}$ be a Poisson process with rate $\lambda$. Define:

- $V(t)$: the fraction of time during $[0, t]$ that $N$ is in some state $B$
- $Z(t)$: the fraction of Poisson arrivals in $[0, t]$ that find $N$ in state $B$

The PASTA theorem says that $V(t) \to V(\infty)$ almost surely if and only if $Z(t) \to V(\infty)$ almost surely.

What makes this result possible is the *Lack of Anticipation Assumption* (LAA): future increments of the arrival process must be independent of the system's past history. The Poisson process satisfies this naturally because of its memoryless property—each arrival time tells you nothing about when the next one will occur. Deterministic sampling, by contrast, violates the LAA since future observation times are entirely predictable from the current time.

### 2.4.3 Solution: Exponentially Distributed Sampling Intervals

Applying PASTA to our monitoring problem, we replace fixed intervals with random intervals drawn from an exponential distribution:

$$\Delta t \sim \text{Exp}(\lambda), \quad \text{where } \lambda = \frac{1}{\mu_{\text{read}}} \tag{2.3}$$

Here, $\mu_{\text{read}}$ is the desired mean interval between observations.

This creates a Poisson process of observation times, directly satisfying the LAA. By PASTA, these randomized observations yield the same time-averaged statistics as continuous monitoring, regardless of underlying traffic patterns. The exponential distribution's memoryless property [29] ensures that each observation time is statistically independent of previous observations and any system periodicities—breaking the synchronization that causes phase-locking in deterministic sampling.

# Chapter 3

# Windowed HLL (W-HLL)

A variety of sketch-based algorithms have been proposed for network monitoring [5,8,21]. Our work builds on these foundations by implementing HyperLogLog directly in programmable switch hardware. The implementation adapts the HyperLogLog algorithm [11] for windowed cardinality estimation in high-speed programmable switches. Unlike traditional HLL deployments that maintain a single continuous sketch, we implement a *windowed* variant (W-HLL) that periodically resets register states to provide cardinality estimates over fixed time intervals. This enables temporal traffic analysis [23] and anomaly detection while preserving the memory efficiency of probabilistic counting.

## 3.1 Design Overview

The windowed design introduces two key operational parameters:

- **Reset interval ($T_{\text{reset}}$)**: This defines how long each monitoring window lasts. Registers are cleared at exact multiples of $T_{\text{reset}}$ to establish deterministic temporal boundaries.
- **Read interval ($\mu_{\text{read}}$)**: This controls the mean time between register observations. Reads follow an exponential distribution to avoid periodic sampling artifacts.

The system supports two operational modes for validation:

- **Synthetic traffic mode**: Uses generated PCAP files where each packet has a unique source IP, allowing rapid validation with the hardware counter serving as ground truth.
- **Real trace mode**: Replays actual network captures, enabling comprehensive accuracy assessment with realistic traffic patterns where multiple packets may share the same source IP.

This chapter details the W-HLL implementation, beginning with the overall system architecture (Section 3.2), then covering data plane design (Section 3.3), control plane implementation (Section 3.4), and HLL estimation (Section 3.5).

**Comparison with existing P4-based HLL implementations:** Unlike existing P4 sketch libraries such as SketchLib [33], which use two separate hash functions—one for bucket selection and another for rank computation—our implementation uses a single 32-bit CRC hash that gets partitioned into two fields through bit indexing (Section 3.3.2). This reduces computational overhead by eliminating one hash operation per packet while still maintaining the statistical independence that HyperLogLog requires, since different bit ranges of a well-distributed hash function remain independent [11].

## 3.2 System Architecture Overview

The W-HLL monitoring system has two main components: the P4 data plane [3] running on the Intel Tofino switch [19], and a Python control plane that communicates through the BFRuntime API. Figure 3.1 shows the complete system architecture.

### 3.2.1 Data Plane Component

The P4 program implements the core packet processing pipeline through the following stages:

1. **Parsing**: Extracts Ethernet and IPv4 headers using `SwitchIngressParser`, with optional VLAN tag support.
2. **Hashing**: Computes a 32-bit CRC hash of the source IPv4 address using polynomial `0x790900f3`.
3. **Index splitting**: Partitions the hash into an 11-bit bucket index ($h[31:21]$) and a 21-bit rank value ($h[20:0]$).
4. **TCAM lookup**: Maps the rank value to a level (leading zero count + 1) using longest prefix match.
5. **Register update**: Updates the HLL bucket in the `cs_table` register array using the max operation—if `level > cs_table[index]`, then `cs_table[index]` $\leftarrow$ `level`.
6. **Counter increment**: Increments a 64-bit hardware packet counter.

The pipeline processes packets at line rate [4, 22] with deterministic single-pass updates. Figure 3.2 illustrates how packets flow through these stages.

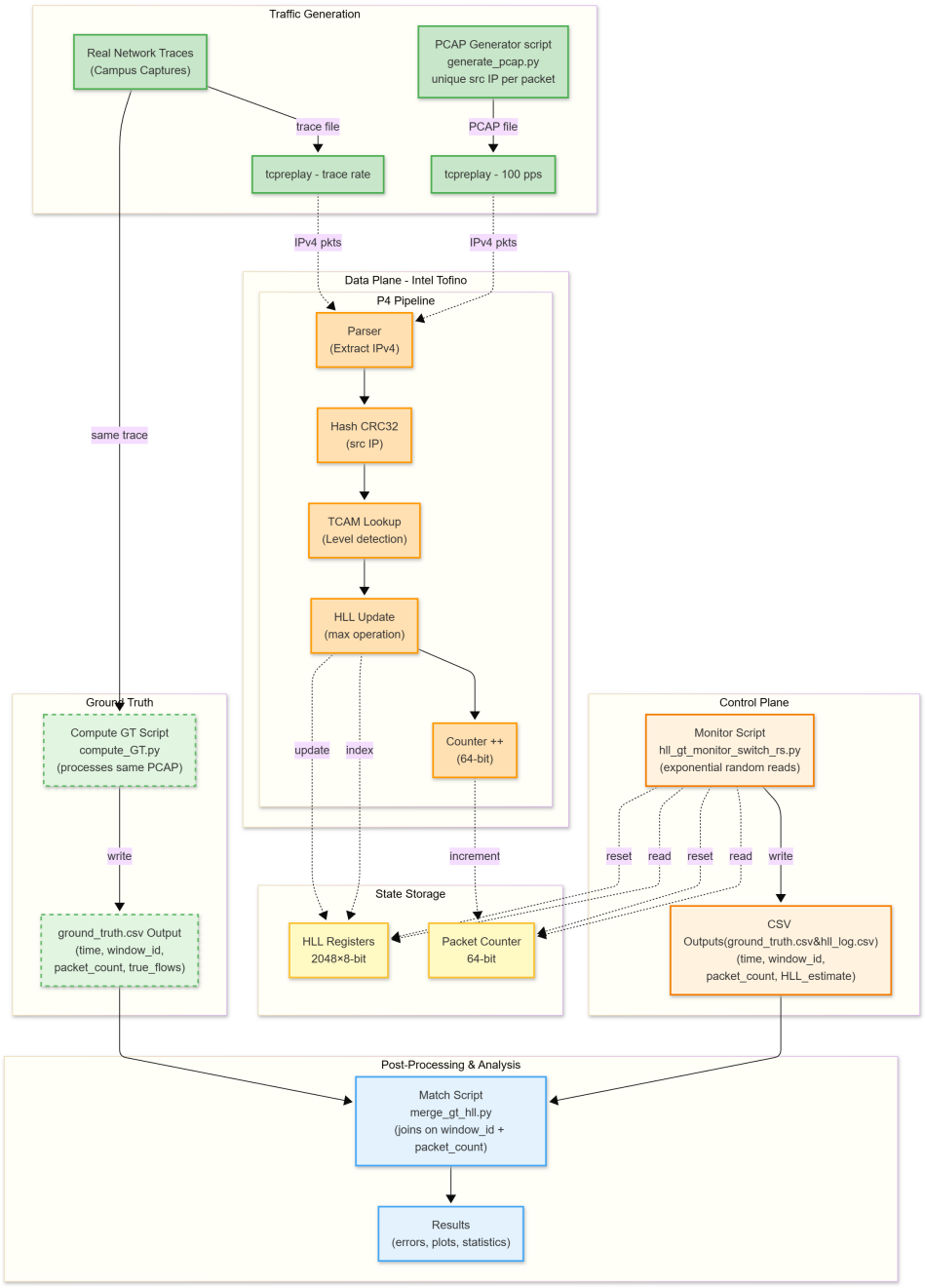


Figure 3.1. W-HLL system architecture showing the interaction between data plane processing and control plane monitoring.

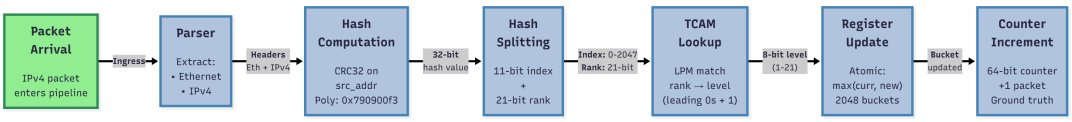


Figure 3.2. Packet processing pipeline showing the sequential stages.

### 3.2.2 Control Plane Component

A Python monitoring script orchestrates the monitoring cycle using event-driven scheduling. It performs five key operations:

1. **Counter reads**: Retrieves the 64-bit packet counter value
2. **Register reads**: Retrieves all 2048 register values, completing in approximately 245 ms
3. **HLL estimation**: Computes cardinality estimate using the standard HLL estimator with small-range correction and large-range correction as needed
4. **Data logging**: Appends measurements (timestamp, HLL estimate, counter value) to CSV file
5. **Reset operations**: Clears all HLL registers and the packet counter at fixed intervals $T_{\text{reset}}$

The control plane connects to the switch over gRPC (port 50052). Read intervals follow an exponential distribution with mean $\mu_{\text{read}}$ to implement Poisson sampling, which avoids synchronization with periodic traffic patterns. Priority-based scheduling ensures that when a read and reset coincide, the read executes first (priority 1) before the reset (priority 2). Figure 3.3 illustrates the control plane monitoring loop.

### 3.2.3 Data Flow

The W-HLL monitoring workflow proceeds as follows:

1. Traffic enters the Tofino switch through the ingress port.
2. The data plane processes each packet: it computes the hash, updates the HLL bucket, and increments the counter.
3. The control plane periodically reads the counter value.
4. The control plane reads all 2048 HLL registers.
5. The control plane computes the HLL estimate and logs the tuple (time, HLL estimate, counter) to a CSV file.
6. At each interval $T_{\text{reset}}$, the control plane clears all registers and the counter.
7. The monitoring cycle repeats with exponentially distributed read intervals.

This architecture ensures deterministic window boundaries and unbiased sampling of the HLL state throughout each monitoring interval.

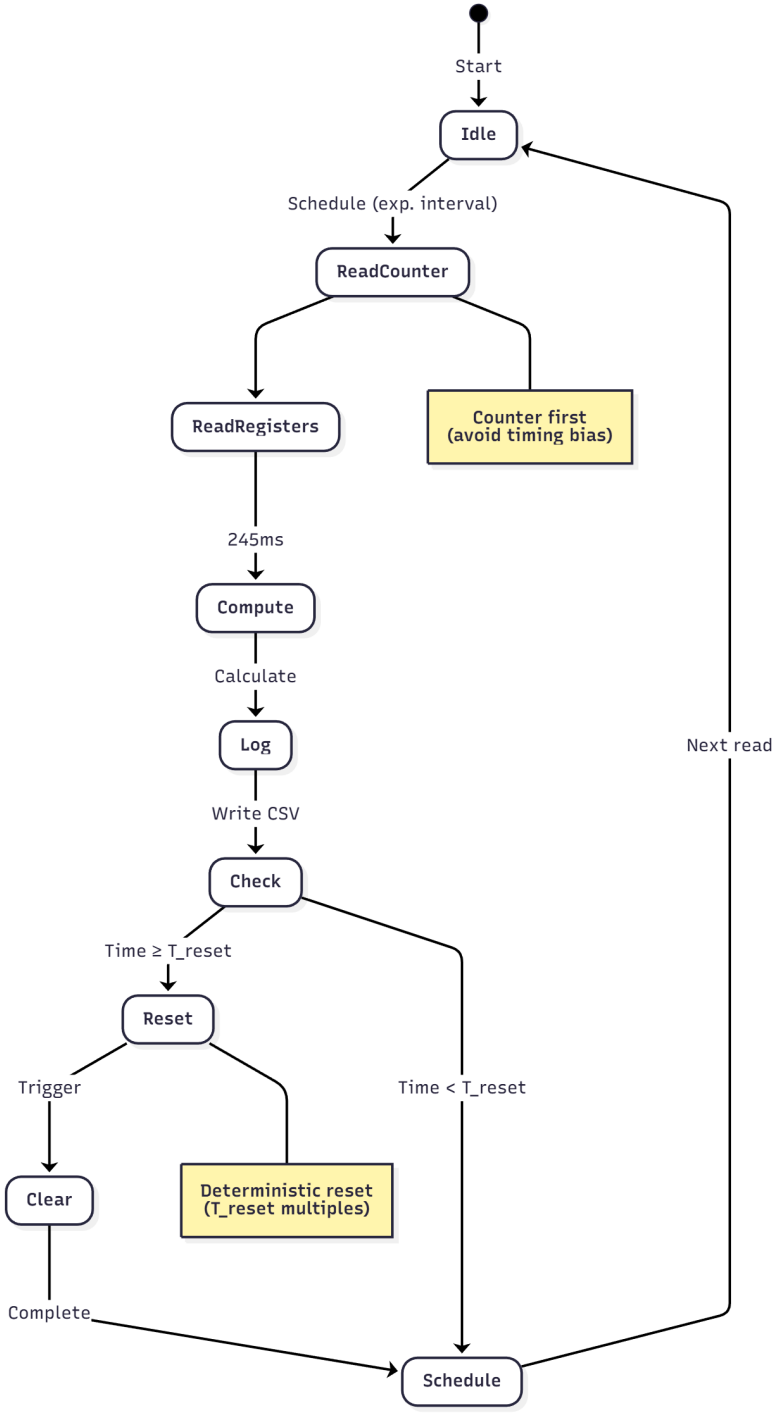


Figure 3.3. Control plane monitoring loop coordinating reads, estimation, and resets.

## 3.3 P4 Data Plane Implementation

The P4 data plane is organized into five main components: header definitions, parser logic, hash computation, TCAM-based rank lookup, and register update primitives.

### 3.3.1 Header Definitions and Parser

The `headers.p4` file defines standard Ethernet and IPv4 headers:

**Header Definitions (`headers.p4`)**

```
// Type definitions for network addresses
typedef bit<48> mac_addr_t;    // MAC: 48-bit address (6 bytes)
typedef bit<32> ipv4_addr_t;   // IPv4: 32-bit address (4 bytes)

// Ethernet header structure (14 bytes total)
header ethernet_h {
    mac_addr_t dst_addr;       // Destination MAC (6 bytes)
    mac_addr_t src_addr;       // Source MAC (6 bytes)
    bit<16>    ether_type;     // Protocol type (2 bytes)
}

// IPv4 header structure (20 bytes minimum)
header ipv4_h {
    bit<4>      version;        // IP version = 4
    bit<4>      ihl;            // Header length in 32-bit words
    bit<8>      tos;            // Type of Service / DSCP
    bit<16>     total_len;      // Total packet length (bytes)
    bit<16>     identification; // Fragment ID
    bit<3>      flags;          // Fragmentation flags
    bit<13>     frag_offset;    // Fragment offset (8-byte units)
    bit<8>      ttl;            // Time To Live counter
    bit<8>      protocol;       // Upper layer protocol (TCP=6, UDP=17)
    bit<16>     hdr_checksum;   // Header integrity checksum
    ipv4_addr_t src_addr;       // Source IP - HLL hash input
    ipv4_addr_t dst_addr;       // Destination IP
}
```

The `SwitchIngressParser` extracts these headers using a state machine:

**Parser Logic (`SwitchIngressParser`)**

```
parser SwitchIngressParser(...) {
    state start {
        pkt.extract(hdr.ethernet);               // Extract 14-byte Ethernet header
        transition select(hdr.ethernet.ether_type) {
            ETHERTYPE_IPV4: parse_ipv4;          // If EtherType=0x0800, parse IPv4
            default: accept;                     // Skip non-IPv4 (VLAN, ARP, etc.)
        }
    }

    state parse_ipv4 {
```

```
11          pkt.extract(hdr.ipv4);                    // Extract 20-byte IPv4 header
12          transition accept;                        // Parsing complete, go to ingress
13      }
14  }
```

This parser handles Ethernet and IPv4 only. VLAN tags and other L2/L3 protocols are not processed in the current implementation, but support can be added as needed.

### 3.3.2 Hash Computation

HyperLogLog requires a hash function that uniformly distributes flow identifiers across the output space. We use Tofino's hardware CRC unit configured with polynomial `0x790900f3`, which provides good uniformity [27] and can be computed in a single pipeline stage.

The hash function is defined in `API_common.p4`:

**Hash Computation Control**

```
1  // P4 macro defining hash computation for IPv4 source addresses
2  #define HASH_COMPUTE_SRCIP_32_32(polynomial)                    \
3  control HASH_COMPUTE_SRCIP_32_32(                               \ // Control block
       name
4      in ipv4_addr_t srcAddr,                                     \ // Input: 32-bit
       source IP
5      out bit<32> result)(bit<32> polynomial)                     \ // Output: 32-bit
       hash value
6  {                                                               \
7      Hash<bit<32>>(HashAlgorithm_t.CUSTOM, polynomial) hash;     \ // CRC hash with
       custom polynomial
8      apply { result = hash.get({srcAddr}); }                     \ // Compute hash of
       source IP
9  }
10
11 HASH_COMPUTE_SRCIP_32_32(32w0x790900f3) full_hash;               // Instantiate with
       polynomial 0x790900f3
```

In the main ingress control block, the hash is applied and partitioned:

**Hash Application and Partitioning**

```
1  bit<32> hash_val;                        // 32-bit CRC hash output
2  full_hash.apply(hdr.ipv4.src_addr, hash_val); // Compute hash of source IP
3
4  bit<11> index = hash_val[31:21];         // Extract upper 11 bits for bucket selection
5  bit<21> sampling_hash = hash_val[20:0]; // Extract lower 21 bits for rank computation
```

The 11-bit index selects one of 2048 HLL buckets. The 21-bit rank value determines the leading zero count via TCAM lookup. Figure 3.4 illustrates this partitioning.

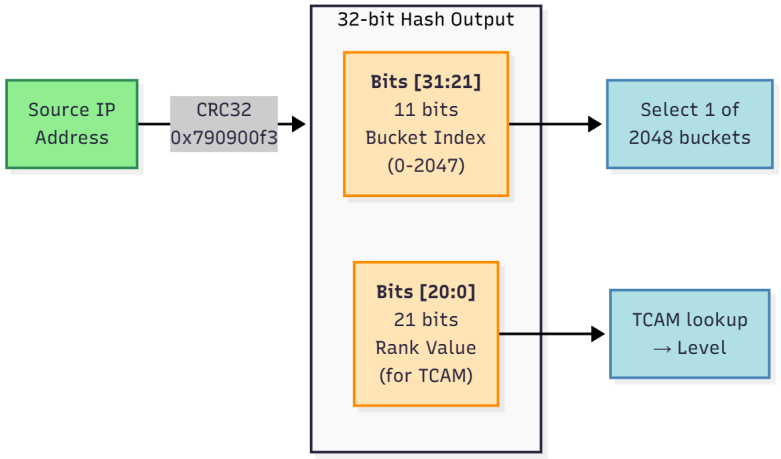


Figure 3.4. Hash partitioning: 32-bit hash split into 11-bit index and 21-bit rank value.

### 3.3.3 TCAM-Based Rank Lookup

To compute the leading zero count of the 21-bit rank value efficiently, We use a TCAM table with longest prefix match (LPM) [16] to map the 21-bit rank value to its leading zero count in $O(1)$ time.

The `lpm_optimization_32` control implements this mapping:

**TCAM-Based Rank Lookup Control**

```
control lpm_optimization_32(
    in bit<21> sampling_hash,       // Input: 21-bit hash value from lower bits
    out bit<8> ret_level)           // Output: leading zero count + 1
{
    action tbl_act(bit<8> level) {
        ret_level = level;          // Assign matched level to output
    }

    table tbl_select_level {
        key = {
            sampling_hash : lpm;    // Longest prefix match on hash bits
        }
        actions = {
            tbl_act;                // Single action: return level
        }
        const default_action = tbl_act(0); // Default: level 0 (no match)
    }

    apply {
        tbl_select_level.apply();  // Execute TCAM lookup
    }
}
```

The control plane populates this table with 21 entries at startup.The TCAM's LPM logic ensures correct operation: when a hash matches multiple entries, the longest matching prefix wins, correctly identifying the first '1' bit position. Figure 3.5 illustrates the TCAM structure.

### 3.3.4 Register Update

The HLL state is implemented as a Tofino register array:

**HLL Register Definition**

```
Register<bit<8>, bit<11>>(2048) cs_table;
```

Each of the 2048 buckets stores an 8-bit value representing the maximum level observed for that bucket. The register is updated using a `RegisterAction` that implements the HLL max operation:

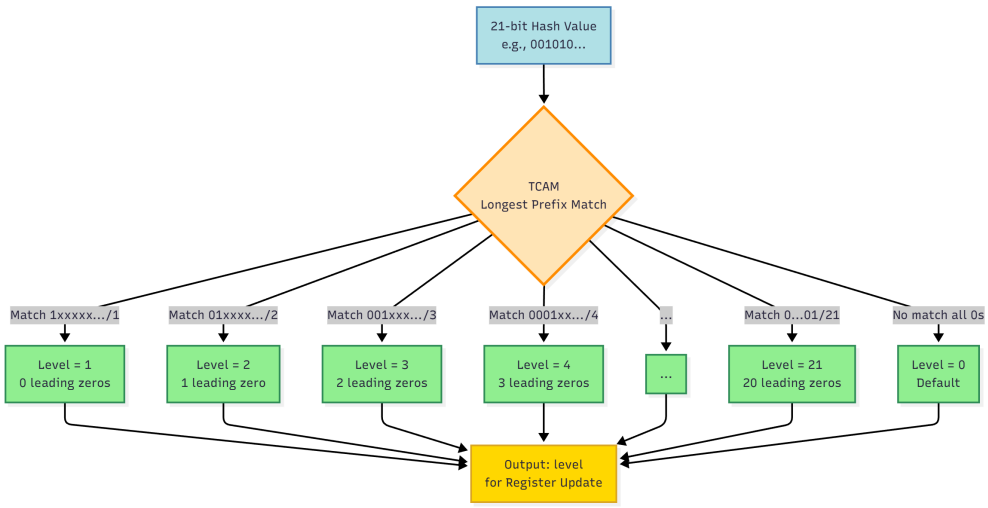


Figure 3.5. TCAM-based rank lookup mapping 21-bit values to levels in $O(1)$ time.

**HLL Register Update Action**

```
RegisterAction<bit<8>, bit<11>, void>(cs_table) cs_action = {
    void apply(inout bit<8> curr_val) {
        if (level > curr_val) { curr_val = level; }
    }
};
```

In the ingress pipeline, the action is invoked for each IPv4 packet:

**HLL Register Update Invocation**

```
cs_action.execute(index);  // index = hash_val[31:21]
```

This ensures that each bucket correctly tracks the maximum rank (rarest hash pattern) observed, as required by the HyperLogLog algorithm.

### 3.3.5 Ground Truth Counter

To enable validation, we include a hardware packet counter:

**Ground Truth Counter Definition**

```
Counter<bit<64>, bit<1>>(1, CounterType_t.PACKETS) ground_truth_counter;
```

This 64-bit counter increments for every IPv4 packet processed:

**Ground Truth Counter Increment**

```
ground_truth_counter.count(1w0);  // Single counter at index 0
```

The counter provides a hardware-based packet count that serves as a reference for validation. In synthetic traffic experiments where each packet has a unique source IP, the packet count equals the true cardinality directly. The control plane reads the counter through BFRuntime and resets it along with the HLL registers at each window boundary.

## 3.4 Control Plane Implementation

The Python control plane (`hll_gt_monitor_switch_rs.py`) manages the monitoring lifecycle through the BFRuntime API. TCAM initialization is handled separately at startup by `bfrt_populate.py` (Section 3.4.2).

### 3.4.1 BFRuntime Initialization

The script connects to the Tofino switch via gRPC [14] and retrieves table references using BFRuntime APIs [18]:

**BFRuntime Initialization**

```
import bfrt_grpc.client as gc

# Create client interface
interface = gc.ClientInterface(grpc_addr='localhost:50052',  # gRPC server address
                               client_id=4,                   # Unique client ID
                               device_id=0)                   # Target device

# Bind compiled P4 program
interface.bind_pipeline_config(PROGRAM_NAME)              # Load P4 program

# Create target object
target = gc.Target(device_id=0, pipe_id=0xFFFF)           # All pipes

# Get table references
hll_table = bfrt_info.table_get("SwitchIngress.update.cs_table")  # HLL registers
counter_table = bfrt_info.table_get("SwitchIngress.update.ground_truth_counter") #
    Packet counter
```

The `grpc_addr` specifies the BFRuntime server address on port 50052. The `client_id` distinguishes this control plane instance from any concurrent clients. Setting `pipe_id` to `0xFFFF` targets all pipes.

### 3.4.2 TCAM Population

At startup, the TCAM is populated by running "`bfshell -b bfrt_populate.py`":

**TCAM Population**

```
# Populate TCAM table with 21 entries for leading zero detection

for level in range(1, 22):                # Levels 1-21 map to 0-20 leading zeros
    prefix_value = (1 << (21 - level))    # Binary pattern: '0...01xxx...x' for level
    prefix_len = level                    # Match the first 'level' bits
    tcam_table.add_with_tbl_act(
        sampling_hash=prefix_value,        # Pattern to match(e.g., 0x100000 for level
    =1)
        sampling_hash_p_length=prefix_len, # Prefix length for LPM
        level=level)                       # Return value: leading_zeros + 1

# Example entries created by this loop:
#   level=1:  prefix=0x100000/1  -> matches '1xxxxx...' -> 0 leading zeros
#   level=2:  prefix=0x80000/2   -> matches '01xxxx...' -> 1 leading zero
#   level=21: prefix=0x00001/21  -> matches '0...01'    -> 20 leading zeros
# The TCAM performs longest prefix match (LPM): when multiple entries match,
# the one with the longest prefix wins, correctly identifying the position
# of the first '1' bit in O(1) hardware time.
```

The TCAM uses longest-prefix-match to detect the first '1' bit position. The table

encoding is:

$$\text{level} = (\text{leading zeros}) + 1 \tag{3.1}$$

This encoding is crucial for the HLL register update logic. The `RegisterAction` performs `max(current_val, level)`, correctly tracking the maximum rank observed in each bucket.

### 3.4.3 Register Read Operations

Register reads retrieve all 2048 HLL bucket values using a bulk grpc read operation with retry logic for robustness:

**HLL Register Read Function**

```
def read_registers(max_retries=3):
    """Read all 2048 HLL register values"""
    for attempt in range(max_retries):                # Retry loop
        try:
            registers = [0] * NUM_REGISTERS           # Init 2048-element list
            for entry, key in hll_table.entry_get(target):  # Fetch all entries
                index = key.to_dict()['$REGISTER_INDEX']['value']  # Get index
                value = entry.to_dict()['SwitchIngress.update.cs_table.f1'][0]  #
    Get level
                registers[index] = value              # Store at position
            return registers                          # Return complete array
        except Exception as e:                        # Handle errors
            if attempt == max_retries - 1:            # Last attempt
                print(f"Read failed: {e}")            # Log failure
                return None                           # Indicate error
```

The call `hll_table.entry_get(target)` returns a generator yielding all table entries, completing in approximately 245 ms.

### 3.4.4 Counter Read Operations

The packet counter is read using a similar approach:

**Counter Read Function**

```
def read_ground_truth_counter(max_retries=3):
    """Read the 64-bit packet counter"""
    for attempt in range(max_retries):
        try:
            for entry, key in counter_table.entry_get(target):
                packets = entry.to_dict()['$COUNTER_SPEC_PKTS']
                return packets
        except Exception as e:
            if attempt == max_retries - 1:
                print(f"Counter read failed: {e}")
                return None
```

### 3.4.5 Reset Operations

Reset operations clear both HLL registers and the packet counter using bulk deletion:

**Reset Function**

```
def perform_reset(sched):
    """Clear all HLL registers and packet counter"""
    hll_table.entry_del(target)      # Clear all 2048 registers
    counter_table.entry_del(target)  # Clear counter
    print(f"[{int(time.time() - start_time)}s] Reset")
    sched.enter(RESET_INTERVAL, 2, perform_reset, (sched,))
```

Calling `entry_del(target)` without specific keys deletes all entries, resetting all registers to their default value (0). Resets are scheduled at deterministic intervals that are exact multiples of `RESET_INTERVAL`.

### 3.4.6 Event Scheduling and Coordination

The monitoring loop uses Python's `sched.scheduler` to coordinate operations:

**Event Scheduling**

```
scheduler = sched.scheduler(time.time, time.sleep)

def perform_read(sched):
    """Execute monitoring cycle"""
    ground_truth = read_ground_truth_counter()  # Read counter
    registers = read_registers()  # Read HLL registers
    estimate = calculate_hll(registers)  # Compute estimate
    log_estimate(estimate, ground_truth)  # Log to CSV

    # Schedule next read with exponential delay
    delay = random.expovariate(1.0 / MEAN_READ_INTERVAL)
    sched.enter(delay, 1, perform_read, (sched,))

# Queue initial events
scheduler.enter(random.expovariate(1.0 / MEAN_READ_INTERVAL), 1,
                perform_read, (scheduler,))
scheduler.enter(RESET_INTERVAL, 2, perform_reset, (scheduler,))

# Start event loop
scheduler.run()
```

**Read interval randomization:** Read operations use exponentially distributed intervals, forming a Poisson process [29] with mean rate $\lambda = 1/\mu_{\text{read}}$. This provides statistical independence, avoids synchronization with periodic traffic, and enables unbiased sampling.

**Priority-based scheduling:** Read operations are assigned priority 1 and reset operations priority 2. When events coincide, the lower priority number executes first, so the final window state is captured before the reset clears it.

**Error handling:** Both read operations include retry logic with graceful error handling to ensure robustness against transient gRPC failures or network issues.

## 3.5 HLL Estimation implementation

The control plane computes HLL cardinality estimates using the standard estimator [11] with small-range correction. Large-range corrections [17] can also be implemented if the cardinality is expected to be very large (beyond $2^{32}$).

### 3.5.1 Standard HLL Estimator

Given 2048 register values $M[0], M[1], \ldots, M[2047]$, the raw HLL estimate is:

$$Z = \sum_{j=0}^{m-1} 2^{-M[j]} \tag{3.2}$$

$$E_{\text{raw}} = \alpha_m \cdot m^2 \cdot \frac{1}{Z} \tag{3.3}$$

where $m = 2048$ and $\alpha_{2048} \approx 0.721347$ is the bias correction constant:

$$\alpha_{2048} = \frac{0.7213}{1 + \frac{1.079}{2048}} \tag{3.4}$$

Implementation:

**HLL Estimation Function**

```
def calculate_hll(registers):
    """Compute HyperLogLog cardinality estimate"""
    if not registers:
        return -1  # Error case

    # Compute harmonic sum
    harmonic_sum = sum(2**(-x) for x in registers)
    if harmonic_sum == 0:
        return 0  # Avoid division by zero

    # Bias correction constant
    alpha = 0.7213 / (1 + 1.079 / NUM_REGISTERS)

    # Standard HLL estimator
    estimate = alpha * (NUM_REGISTERS ** 2) / harmonic_sum

    # Small range correction
    if estimate <= 2.5 * NUM_REGISTERS:
```

```
19          zeros = registers.count(0)
20          if zeros:
21              return int(NUM_REGISTERS * math.log(NUM_REGISTERS / zeros))
22
23      return int(estimate)
```

The harmonic sum $Z$ gives higher weight to buckets with larger values, which correspond to rarer hash patterns. The level values stored in registers are used directly as $M[j] = \text{level} = \text{leading\_zeros} + 1$. Small-range correction is applied when the estimate falls below $2.5 \times m$ and there are empty buckets.

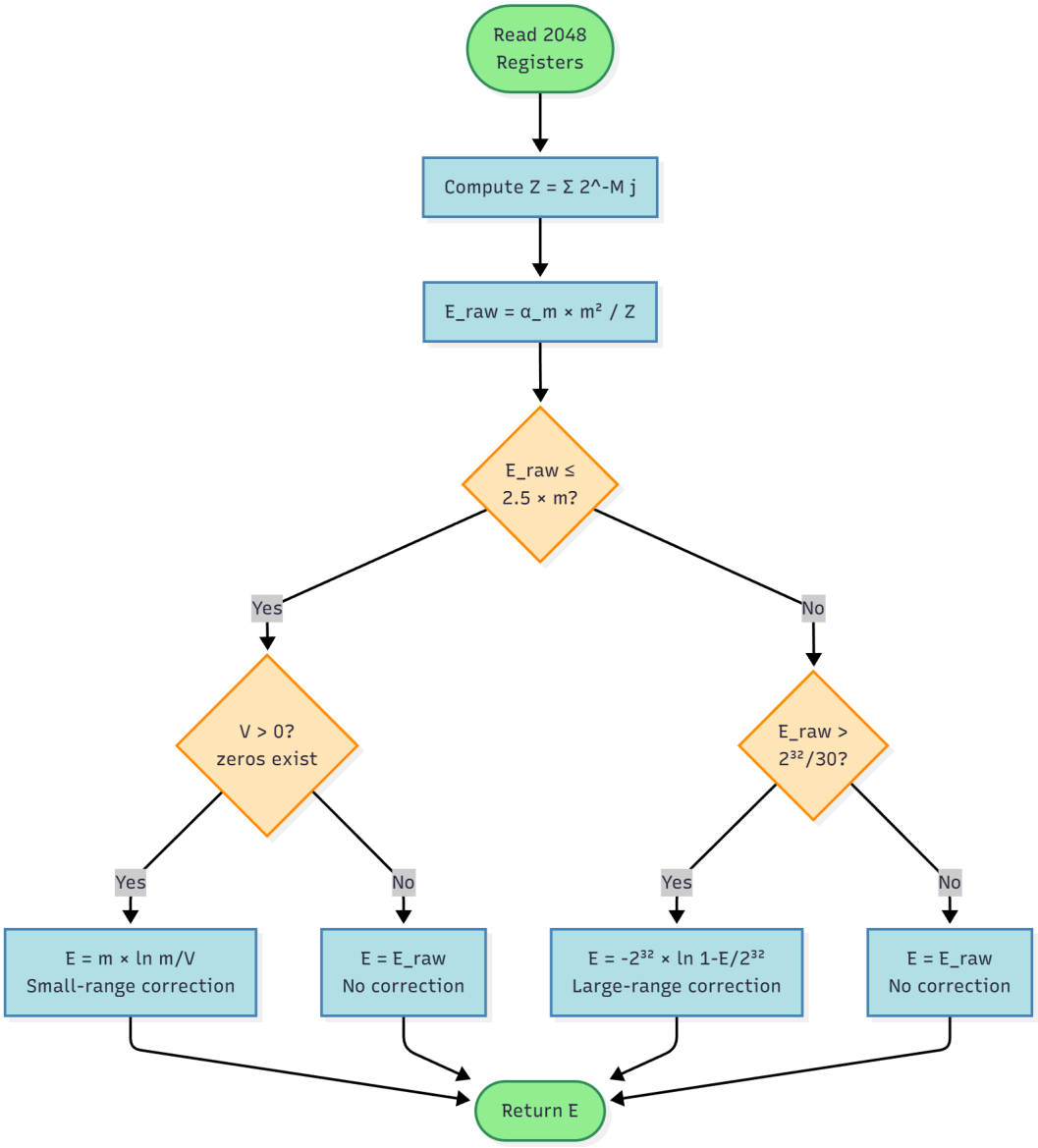


Figure 3.6. HLL estimation flowchart showing standard formula with small-range correction.

# Chapter 4

# Experimental Validation

This chapter presents the experimental validation of the W-HLL implementation described in Chapter 3. We evaluate accuracy under both synthetic and realistic traffic conditions, comparing HLL estimates against ground truth across a range of operational parameters.

## 4.1 Setup

Our experimental testbed uses the SUPERNET infrastructure at Politecnico di Torino, which provides dedicated high-speed programmable switching hardware with separate dataplane and management networks [28]. This section describes the testbed architecture, network configuration, and experimental parameters.

### 4.1.1 Testbed

The SUPERNET testbed provides a research platform for P4-programmable networking experiments, featuring two Intel Tofino switches connected by a 100 Gbps dataplane network [28].

#### Hardware Components

- **Tofino P4 Switches:**
  - `rest-bfsw01.polito.it`
  - `rest-bfsw02.polito.it`
  - Architecture: Intel Tofino ASIC with programmable pipeline
- **Control Plane Server:**
  - `restsrv01.polito.it`
  - Role: SSH bastion host for remote access to switches

- **Network Infrastructure:**
  - Cisco Catalyst 9200/9500 switches for management network
  - 100 Gbps dataplane network with QSFP DAC cables
  - Management network for SSH access and control

### 4.1.2 Network Configuration

The SUPERNET testbed uses a dual-network architecture to isolate experimental traffic from control operations. Figure 4.1 shows the complete network topology.

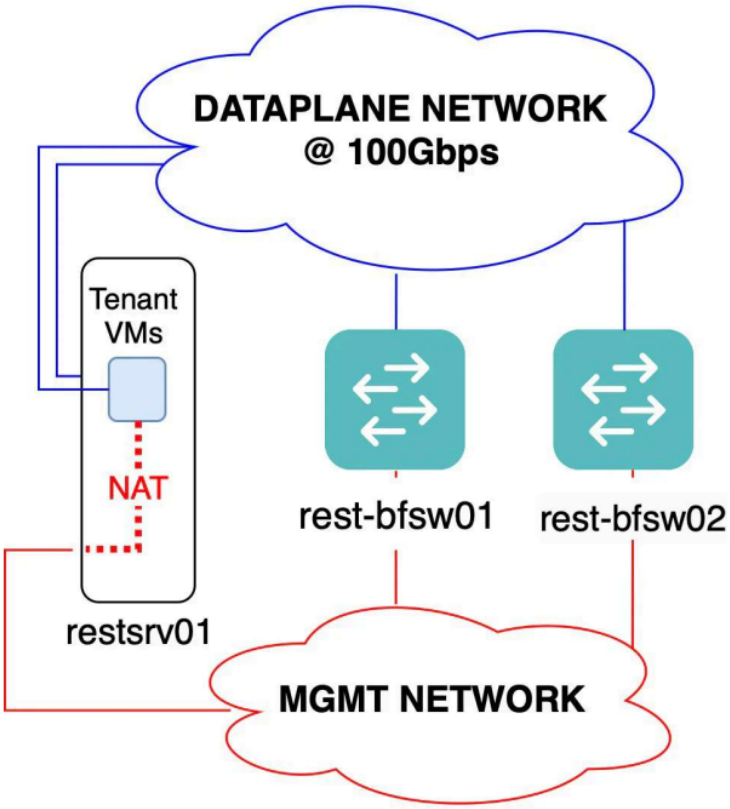


Figure 4.1. SUPERNET testbed logical architecture. The dataplane network (100 Gbps, VLAN 100) carries experimental traffic, while the management network handles control operations. The control plane server (`restsrv01`) provides SSH bastion access to both Tofino switches (adapted from [28]).

**Dataplane Network** The dataplane network provides high-speed connectivity for experimental traffic:

- **Link speed:** 100 Gbps using QSFP DAC cables

- **Purpose:** Traffic generation, packet forwarding, and experimental data flows

**Management Network** The management network enables control plane access and switch configuration:

- **Purpose:** SSH access, BFRuntime gRPC (port 50052), configuration management
- **Access:** Available within Politecnico di Torino network or via VPN

Figure 4.2 illustrates the components and connectivity of the SUPERNET testbed.

### 4.1.3 Software Environment

**P4 Development Tools**

- **P4 language:** $P4_{16}$ specification
- **Compiler:** Intel P4 Studio compiler for the Tofino architecture
- **Runtime API:** BFRuntime gRPC interface on port 50052
- **Switch OS:** Ubuntu 20.04 LTS with Intel SDE (Switch Development Environment)

**Control Plane Software** All monitoring scripts—register reads, packet counter access, state resets, and traffic injection—run locally on the Tofino switch (`rest-bfsw02`) CPU:

- **Python version:** Python 3.8+
- **BFRuntime client:** `bfrt_grpc` Python library for switch state access
- **Data processing:** Pandas and NumPy for post-experiment analysis

**Traffic Generation and Validation Tools**

- **Packet replay:** `tcpreplay` for controlled PCAP injection
- **PCAP generation:** Scapy library for synthetic traffic creation
- **Ground truth computation:** `tshark` for trace analysis
- **PCAP processing:** `tshark`, `editcap`, and `capinfos` for trace manipulation

### 4.1.4 Experimental Configuration

Our experiments use a single Tofino switch (`rest-bfsw02`), with all experimental components running locally on the switch itself. Traffic is injected from the switch CPU directly into the Tofino ASIC pipeline through an internal interface, `enp4s0f0`.

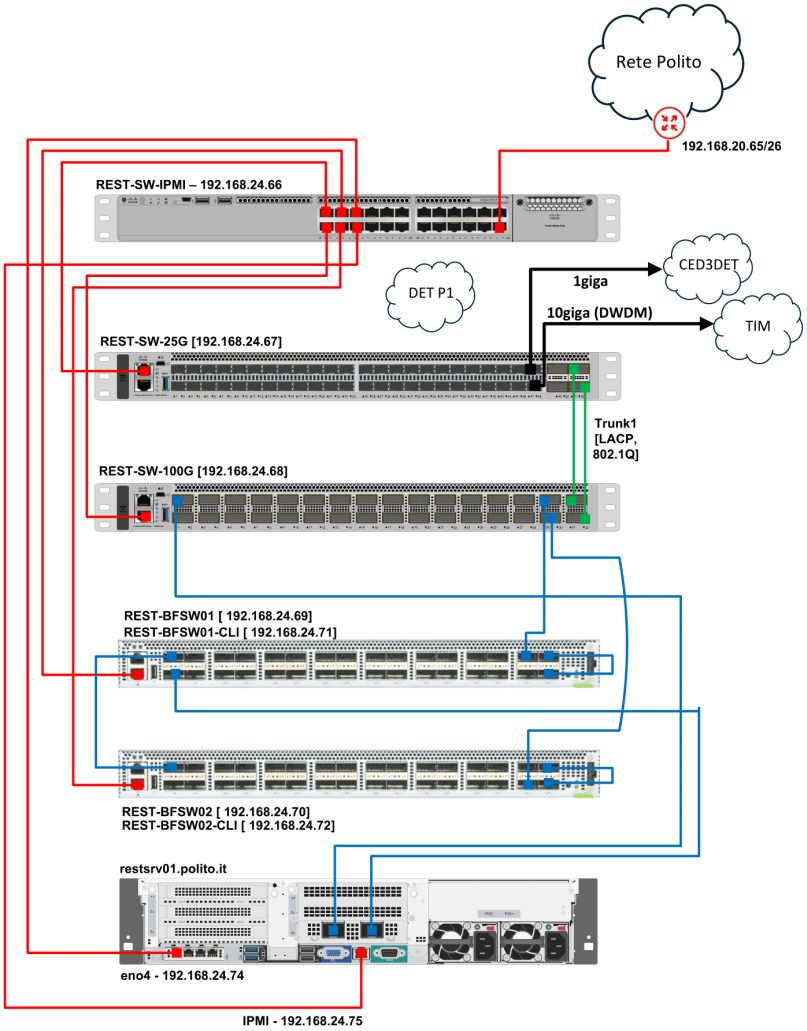


Figure 4.2. SUPERNET testbed components and connectivity (adapted from [28]).

**Traffic Injection Architecture** Our setup takes advantage of the switch's integrated CPU to inject packets directly into the Tofino ASIC:

- **Traffic source:** PCAP files stored on the switch's local storage
- **Injection method:** `tcpreplay` sends packets from the CPU to the ASIC via `enp4s0f0`
- **HLL monitoring:** A BFRuntime client reads switch registers over gRPC
- **Remote access:** All components are accessed via SSH through the `restsrv01` bastion host

**Traffic Flow** The experimental workflow proceeds as follows:

1. A PCAP file is stored on `rest-bfsw02` local storage.
2. `tcpreplay`, running on the switch CPU, injects packets to the ASIC via the internal interface `enp4s0f0`.
3. Packets traverse the P4 pipeline: hash computation, HLL register update, and counter increment.
4. The HLL monitor, also running on the switch CPU, reads switch state through BFRuntime gRPC on port 50052.
5. All measurements are logged to CSV files for post-experiment analysis.
6. At each interval $T_{\text{reset}}$, both the HLL registers and packet counter are cleared.
7. A ground truth tracker running on `restsrv01.polito.it` computes the true cardinality of the injected traffic for accuracy evaluation.

### 4.1.5 Experimental Parameters

Table 4.1 summarizes the key experimental parameters used across all validation tests.

This configuration provides a controlled environment for systematic accuracy evaluation while still being realistic enough for practical validation.

## 4.2 Methodology

This section describes the experimental methodology used to validate W-HLL accuracy. We cover the validation approach, measurement procedures, synchronization mechanisms, and error metrics used to assess HLL performance under both controlled conditions and real traffic patterns.

Table 4.1. Experimental Parameters

| Parameter | Value | Description |
|---|---|---|
| *Data Plane Configuration* | | |
| HLL buckets ($m$) | 2048 | Register array size |
| Bucket size | 8 bits | Maximum level value: 255 |
| Hash function | CRC32 | Polynomial: `0x790900f3` |
| Hash bits (index) | 11 bits | Selects bucket (0–2047) |
| Hash bits (rank) | 21 bits | For leading zero detection |
| TCAM entries | 21 | Maps rank to level (LPM) |
| *Control Plane Configuration* | | |
| Reset interval ($T_{\text{reset}}$) | 5s,10s,50s,100s | Accumulation period |
| Mean read interval ($\mu_{\text{read}}$) | 0.5s,1s,5s,10s | Average sampling period |
| Read distribution | Exponential | Poisson process sampling |
| *Traffic Configuration* | | |
| Traffic type | Synthetic/Real | Two validation modes |
| Synthetic traffic | Unique IPs | Controlled cardinality |
| Real trace | CAMPUS | Realistic IP distribution |
| Traffic rate | 100 pps | Controlled replay rate |
| Real trace rate | Variable | Matches original timing |
| Synthetic/Real duration | 1320 s | 132K packets synthetic / 5M packets real mode |
| Interface | `enp4s0f0` | CPU-to-ASIC interface |
| Flow definition | Source IP | 5-tuple not used |

Table 4.2. Comparison of Validation Modes

| Characteristic | Synthetic Mode | Real Trace Mode |
|---|---|---|
| Traffic source | Scapy-generated PCAP | Campus network traces |
| IP distribution | One unique srcIP per packet | Realistic |
| Ground truth source | Packet counter (hardware) | Flow tracker (software) |
| GT computation | Direct (count = cardinality) | Set maintenance |
| Components | 2 processes (replay + monitor) | 3 processes (replay + monitor + tracker) |
| Use case | Controlled accuracy testing | Real-world validation |

### 4.2.1 Validation Approach

Our validation methodology uses two distinct modes to assess HLL accuracy under controlled and realistic conditions. Table 4.2 summarizes the key characteristics of each approach.

**Synthetic Traffic Mode** generates PCAP files where each packet has a unique source IP address. This allows rapid validation using the hardware packet counter and is useful for parameter tuning under known conditions.

**Real Trace Mode** replays realistic network captures where multiple packets may share the same source IP. This mode evaluates performance with bursty traffic, non-uniform IP distributions, and the temporal correlations typical of production networks.

### 4.2.2 Experimental Variables

We systematically vary the following parameters to evaluate their impact on HLL accuracy:

- **Reset interval ($T_{\text{reset}}$):** Window duration ranging from 5 to 100 seconds
- **Mean read interval ($\mu_{\text{read}}$):** Average time between HLL observations, from 0.5 to 10 seconds
- **Traffic type:** Synthetic (controlled) versus real network traces
- **Cardinality range:** From tens to tens of thousands of unique flows

### 4.2.3 Synchronization Mechanisms

Accurate validation requires precise synchronization between HLL monitoring and ground truth computation:

- **Window boundaries:** Both systems reset at exact multiples of $T_{\text{reset}}$, keeping monitoring windows aligned.
- **Composite key matching:** The tuple (`window_id`, `packet_count`) uniquely identifies each system state, enabling post-experiment log merging.
- **Read-before-reset priority:** When read and reset events coincide, priority-based scheduling ensures the read executes first (priority 1) before the reset (priority 2), capturing the final window state.

### 4.2.4 Measurement Procedures

**Measurement Loop** Each loop follows this sequence:

1. Wait for an exponentially distributed interval with mean $\mu_{\text{read}}$.
2. Read the hardware packet counter via BFRuntime.
3. Read all 2048 HLL registers via BFRuntime.
4. Compute the HLL estimate using the standard estimator with corrections.
5. Log the tuple: (`time`, `window_id`, `ground_truth`, `estimate`).
6. For real trace mode, ground truth is computed from the injected PCAP to produce: (`time`, `window_id`, `packet_count`, `ground_truth_flows`).

**Critical Timing Consideration** The packet counter is read *before* the HLL registers to prevent synchronization bias. Since reading all registers takes approximately 245 ms, reading them first would allow new packets to arrive during the operation. At 100 pps, this would result in roughly $245 \times 10^{-3} \times 100 \approx 24$ extra packets being counted but not reflected in the HLL state. By reading the counter first—which is fast—both measurements refer to approximately the same set of packets.

### 4.2.5 Error Metrics

We use multiple error metrics to comprehensively assess HLL accuracy:

**Relative Error** The primary metric is relative error, defined as:

$$\epsilon_{\text{rel}} = \frac{\hat{n} - n}{n} \tag{4.1}$$

where $\hat{n}$ is the HLL estimate and $n$ is the true cardinality. Positive values indicate overestimation, negative values underestimation.

**Absolute Relative Error** For aggregate statistics, we use absolute relative error:

$$|\epsilon_{\text{rel}}| = \left| \frac{\hat{n} - n}{n} \right| \tag{4.2}$$

**Theoretical Comparison** HLL with $m = 2048$ buckets has a theoretical standard error of:

$$\sigma_{\text{HLL}} = \frac{1.04}{\sqrt{2048}} \approx 0.023 = 2.3\% \tag{4.3}$$

We compare our empirical error statistics against this theoretical benchmark.

Figure 4.3 summarizes the complete workflow from setup through analysis. Post-experiment analysis merges logs based on synchronized window IDs for accuracy evaluation.

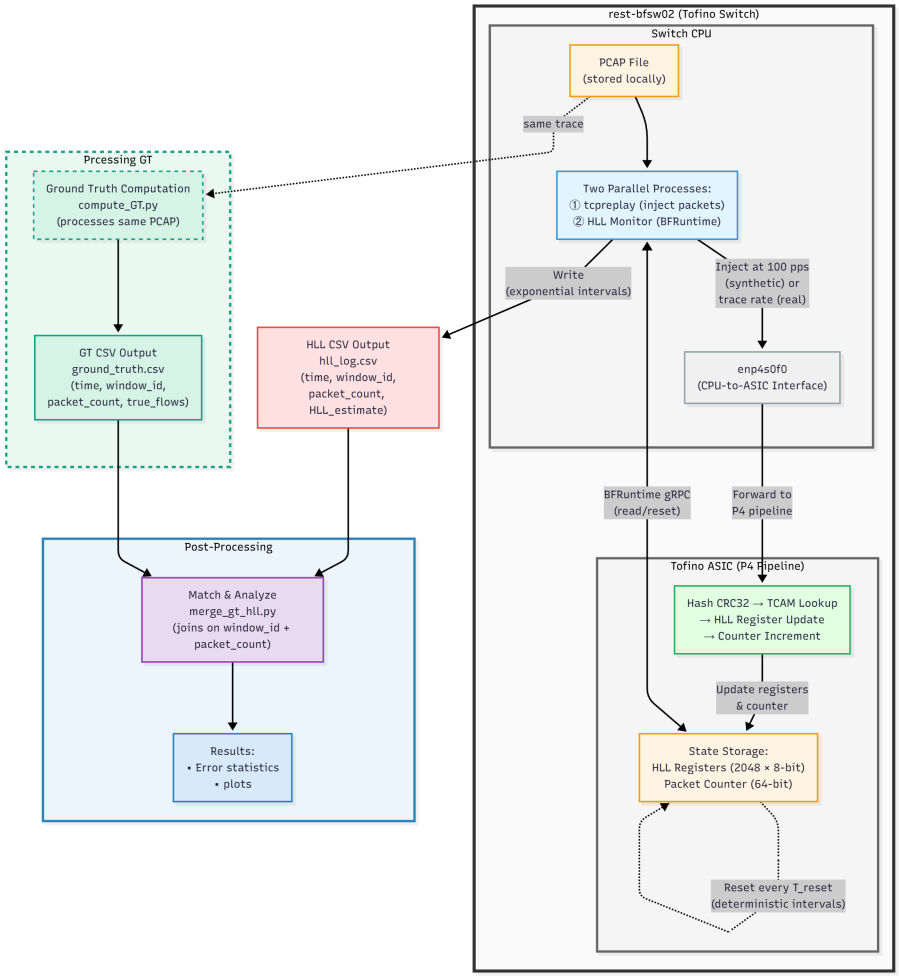


Figure 4.3. Experimental workflow showing the two validation modes, with parallel processes for HLL monitoring, traffic replay, and optional ground truth tracking.

### 4.2.6 PASTA Implementation

W-HLL uses exponentially distributed read intervals to ensure unbiased time-averaged measurements. This design is grounded in the PASTA theorem (Poisson Arrivals See Time Averages) [32], which guarantees that observations from a Poisson process yield the same statistics as continuous monitoring (Section 2.4). The key implementation details are as follows:

**Exponential Interval Generation**

```
import random

MEAN_READ_INTERVAL = 1.0  # seconds

def schedule_next_read(scheduler):
    """Schedule next read with exponentially distributed delay"""
    delay = random.expovariate(1.0 / MEAN_READ_INTERVAL)
    scheduler.enter(delay, priority=1, action=perform_read,
                    argument=(scheduler,))
```

#### Deterministic Reset Intervals

While read intervals are randomized for unbiased observation, reset intervals remain strictly deterministic, occurring at exact multiples of $T_{\text{reset}}$. This ensures reproducible experimental conditions and clear window boundaries for analysis. Combining Poisson reads with deterministic resets provides both statistical validity through PASTA and the experimental control needed for rigorous evaluation.

## 4.3 Traffic Generation and Replay

This section describes traffic preparation and replay procedures for both validation modes.

### 4.3.1 Synthetic Traffic Generation

Synthetic traffic provides controlled conditions for initial validation, where the ground truth is known by construction. Each packet has a unique source IP address, so the packet count equals the flow cardinality directly.

Our synthetic traffic generator creates PCAP files with the following specifications:

- **Experiment duration:** 1320 seconds (22 minutes) by default
- **Target rate:** 100 packets per second by default
- **Total packets:** $1320 \times 100 = 132{,}000$ packets
- **Unique flows:** 132,000 (one per packet)
- **IP range:** 10.0.0.0/8 with random allocation

The synthetic traffic generator uses Scapy to create PCAP files with randomized unique source IPs:

**Synthetic Traffic Generator (generate_pcap.py)**

```
#!/usr/bin/env python3

from scapy.all import wrpcap, Ether, IP, TCP
import random

EXPERIMENT_DURATION = 1320                          # Seconds
PACKETS_PER_SECOND = 100                            # Target rate
TOTAL_PACKETS = EXPERIMENT_DURATION * PACKETS_PER_SECOND
OUTPUT_FILE = "synthetic_traffic_random.pcap"

print(f"Generating {TOTAL_PACKETS} packets with RANDOM unique IPs...")

packets = []
used_ips = set()                                    # Track used IPs

for i in range(TOTAL_PACKETS):
    # Generate unique random IP in 10.0.0.0/8
    while True:
        src_ip = f"10.{random.randint(0,255)}." \
                 f"{random.randint(0,255)}." \
                 f"{random.randint(1,255)}"
        if src_ip not in used_ips:                  # Ensure uniqueness
            used_ips.add(src_ip)
            break

    dst_ip = "10.0.0.2"                             # Fixed destination

    # Create packet with explicit MAC addresses
    pkt = Ether(dst="f6:22:3b:a1:3a:fb",
                src="f6:e9:be:5f:4f:0b") / \
          IP(src=src_ip, dst=dst_ip) / \
          TCP()

    packets.append(pkt)

    # Progress indicator
    if (i + 1) % 10000 == 0:
        print(f"  Generated {i + 1}/{TOTAL_PACKETS} packets...")

# Write PCAP file
print(f"Writing to {OUTPUT_FILE}...")
wrpcap(OUTPUT_FILE, packets)
print(f"Done! Created {OUTPUT_FILE}")
print(f"Total unique flows: {len(used_ips)}")
```

Before using the generated PCAP, we verify its properties:

**PCAP Verification**

```
# Basic file info
capinfos synthetic_traffic_random.pcap

# Count total packets
tshark -r synthetic_traffic_random.pcap | wc -l
# Expected: 132000

# Count unique source IPs (should equal packet count)
tshark -r synthetic_traffic_random.pcap -T fields -e ip.src | \
    sort -u | wc -l
# Expected: 132000

# Verify no duplicate IPs
tshark -r synthetic_traffic_random.pcap -T fields -e ip.src | \
    sort | uniq -d
# Expected: (empty - no duplicates)
```

### 4.3.2 Real Traffic Traces

Real network traces are obtained from the Politecnico di Torino Jupyter-based cluster environment (https://jupyter.polito.it/expert/), capturing actual campus network traffic with bursty behavior, non-uniform IP distributions, and temporal correlations.

### 4.3.3 Traffic Replay

Both traffic types are replayed using `tcpreplay` on the CPU-to-ASIC interface (`enp4s0f0`). Synthetic traffic uses fixed rate control (`--pps=100`), while real traces can either preserve their original timing or use rate limiting for controlled experiments:

**Traffic Replay Commands**

```
# Synthetic traffic replay (100 pps)
python3 hll_gt_monitor_switch_rs.py & sudo tcpreplay --intf1=enp4s0f0 --pps=100
    synthetic_traffic.pcap

# Real trace replay (preserve timing)
python3 hll_gt_monitor_switch_rs.py & sudo tcpreplay --intf1=enp4s0f0 Real_1320.
    pcap
```

## 4.4 Ground Truth Measurement

Ground truth measurement strategies differ between validation modes based on traffic characteristics.

### 4.4.1 Synthetic Mode: Hardware Counter

In synthetic mode, the hardware packet counter (Section 3.3.5) provides ground truth directly, since each packet has a unique source IP. The HLL monitor reads the counter synchronously with the registers, logging (`window_id`, `packet_count`, `estimate`) tuples where the packet count equals the true cardinality.

### 4.4.2 Real Trace Mode: Software Flow Tracker

Real traces require explicit flow counting since multiple packets may share the same source IP. A software flow tracker maintains a set of observed unique source IPs, updating it within each window. The tracker logs the packet count and window ID, which are used to synchronize with the HLL estimates.

**Software Flow Tracker (`compute_GT.py`)**

```
#!/usr/bin/env python3

import csv
import subprocess
import sys

# Configuration
PCAP_FILE = "Real_1320.pcap"
RESET_INTERVAL = 100.0
OUTPUT_FILE = "ground_truth.csv"

print(f"Processing PCAP: {PCAP_FILE}")

# Open CSV for writing
csv_file = open(OUTPUT_FILE, 'w', newline='')
csv_writer = csv.writer(csv_file)
csv_writer.writerow(['time', 'window_id', 'packet_count', 'ground_truth_flows'])

# State
flow_set = set()
packet_counter = 0
window_id = 0
start_time = None
new_reset_time = RESET_INTERVAL

try:
    # Run tshark with streaming output
    process = subprocess.Popen(
        ['tshark', '-r', PCAP_FILE, '-T', 'fields',
         '-e', 'frame.time_epoch', '-e', 'ip.src',
         '-Y', 'ip', '-E', 'separator=,'],
        stdout=subprocess.PIPE, text=True, bufsize=1
    )

    # Process line by line
    for line in process.stdout:
```

```
        parts = line.strip().split(',')
        if len(parts) < 2:
            continue

        timestamp = float(parts[0])
        src_ip = parts[1].strip()

        if start_time is None:
            start_time = timestamp

        current_time = timestamp - start_time

        # Reset if needed
        if current_time > new_reset_time:
            flow_set.clear()
            packet_counter = 0
            new_reset_time += RESET_INTERVAL
            window_id += 1

        # Update state
        flow_set.add(src_ip)
        packet_counter += 1

        # Write row
        csv_writer.writerow([f"{current_time:.3f}", window_id,
                             packet_counter, len(flow_set)])

    csv_file.close()
    print(f"Done! Output: {OUTPUT_FILE}")

except Exception as e:
    print(f"Error: {e}")
    csv_file.close()
    sys.exit(1)
```

### 4.4.3 Synchronization and Verification

Both validation modes rely on synchronized window boundaries at $T_{\text{reset}}$ intervals. Real trace mode additionally requires composite keys (`window_id`, `packet_count`) for matching HLL estimates with ground truth. After the experiment, logs are merged using `merge_gt_hll.py`:

**Log Merging and Error Calculation**

```
import pandas as pd

# Read CSV files
hll_df = pd.read_csv('hll_log.csv')
gt_df = pd.read_csv('ground_truth.csv')

# Rename column for consistency
```

```
hll_df = hll_df.rename(columns={'ground_truth': 'packet_count'})

# Merge on composite key (window_id, packet_count)
merged = pd.merge(
    hll_df,
    gt_df[['window_id', 'packet_count', 'ground_truth_flows']],
    on=['window_id', 'packet_count'],
    how='inner'
)

# Calculate error metrics
merged['relative_error'] = (
    (merged['estimate'] - merged['ground_truth_flows'])
    / merged['ground_truth_flows']
)
merged['absolute_error'] = abs(merged['relative_error'])

# Save results
merged.to_csv('verification.csv', index=False)
```

This aligns each HLL estimate with its corresponding ground truth measurement at the exact system state, enabling plotting and statistical analysis of accuracy.

## 4.5 Experimental Results

We evaluate the accuracy of our HyperLogLog implementation on the Intel Tofino switch through two experimental scenarios: synthetic traffic generation and real network trace validation. Each scenario tests four window configurations with mean read intervals ($\mu_{\text{read}}$) of 0.5s, 1s, 5s, and 10s, paired with reset intervals ($T_{\text{reset}}$) of 5s, 10s, 50s, and 100s respectively, maintaining a consistent 10:1 ratio. All experiments follow the methodology described in Section 4.2 using the SUPERNET testbed [28].

### 4.5.1 Synthetic Traffic Experiments

#### Overview

We generated synthetic traffic at 100 pps with uniformly distributed flow identifiers, following the procedure in Section 4.3.1. Ground truth was obtained directly from the hardware packet counter (Section 4.4), since each packet contains a unique source IP address by construction. Table 4.3 summarizes the accuracy metrics across all configurations, showing consistent improvement with longer monitoring windows.

| **Read (s)** | **Reset (s)** | **Avg. Error (%)** | **Median Error (%)** | **Samples** | **Max Card. (flows)** |
|---|---|---|---|---|---|
| 0.5 | 5 | 3.42 | 1.46 | 1,654 | 500 |
| 1 | 10 | 2.65 | 1.32 | 1,066 | 1,000 |
| 5 | 50 | 1.47 | 1.21 | 273 | 5,000 |
| 10 | 100 | 1.41 | 1.16 | 133 | 10,000 |

Table 4.3. HyperLogLog accuracy for synthetic traffic experiments at 100 pps. Results show consistent improvement in estimation accuracy as monitoring window duration increases.

### Results Across Window Configurations

**Configuration 1: 0.5s Read, 5s Reset** With the shortest monitoring window (Figures 4.4 and 4.5), the system tracks cardinalities up to 500 flows with an average relative error of 3.42% and a median error of 1.46% over 1,654 measurements. The higher average error reflects outliers at low cardinalities where small-range correction is active. Scatter plots show that estimates closely follow the ideal diagonal line, with increased scatter at lower cardinalities.

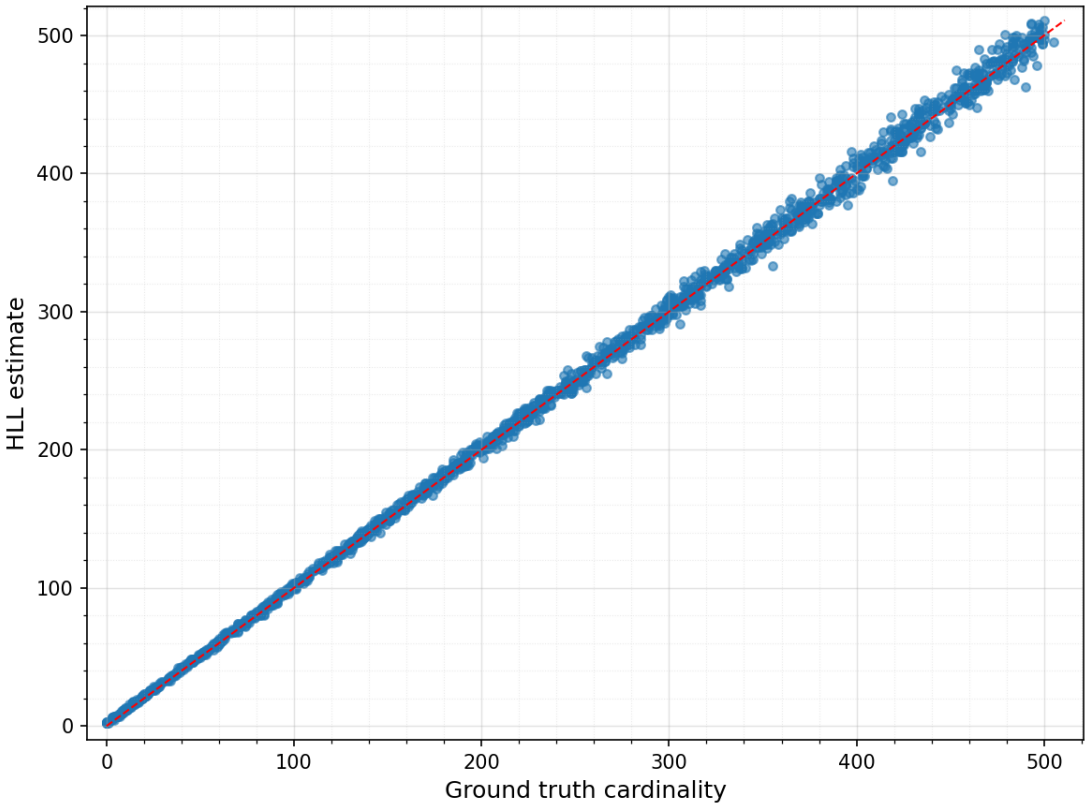


Figure 4.4. HyperLogLog estimates vs. ground truth for synthetic traffic with 0.5s read interval and 5s reset interval (linear scale). The system tracks cardinalities up to 500 flows with average relative error of 3.42% and median error of 1.46% over 1,654 measurements.

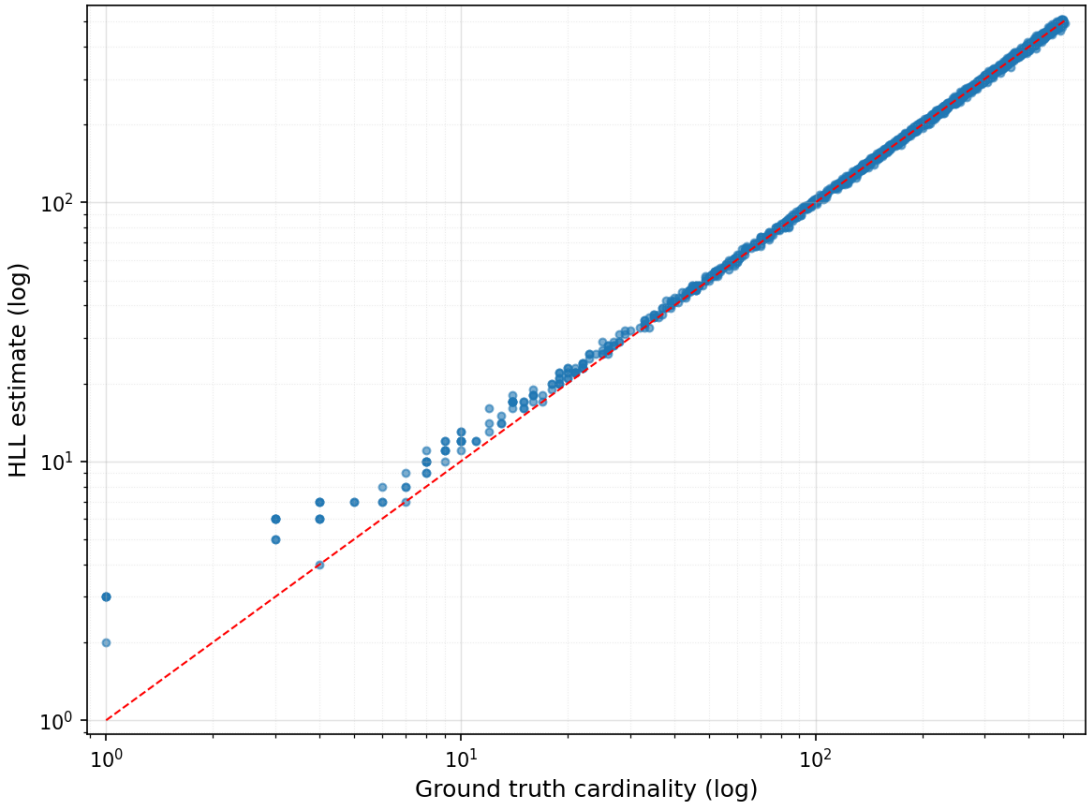


Figure 4.5. HyperLogLog estimates vs. ground truth for synthetic traffic with 0.5s read interval and 5s reset interval (log scale). The logarithmic representation reveals estimation behavior across the full dynamic range, with some scatter visible at lower cardinalities.

**Configuration 2: 1s Read, 10s Reset** Doubling the window duration (Figures 4.6 and 4.7) extends cardinality tracking to 1,000 flows with improved accuracy: average error of 2.65% and median error of 1.32% over 1,066 measurements. The tighter clustering around the ideal line shows that longer accumulation periods reduce variance by collecting more samples per bucket, reflecting the trade-off between temporal resolution and per-window accuracy.

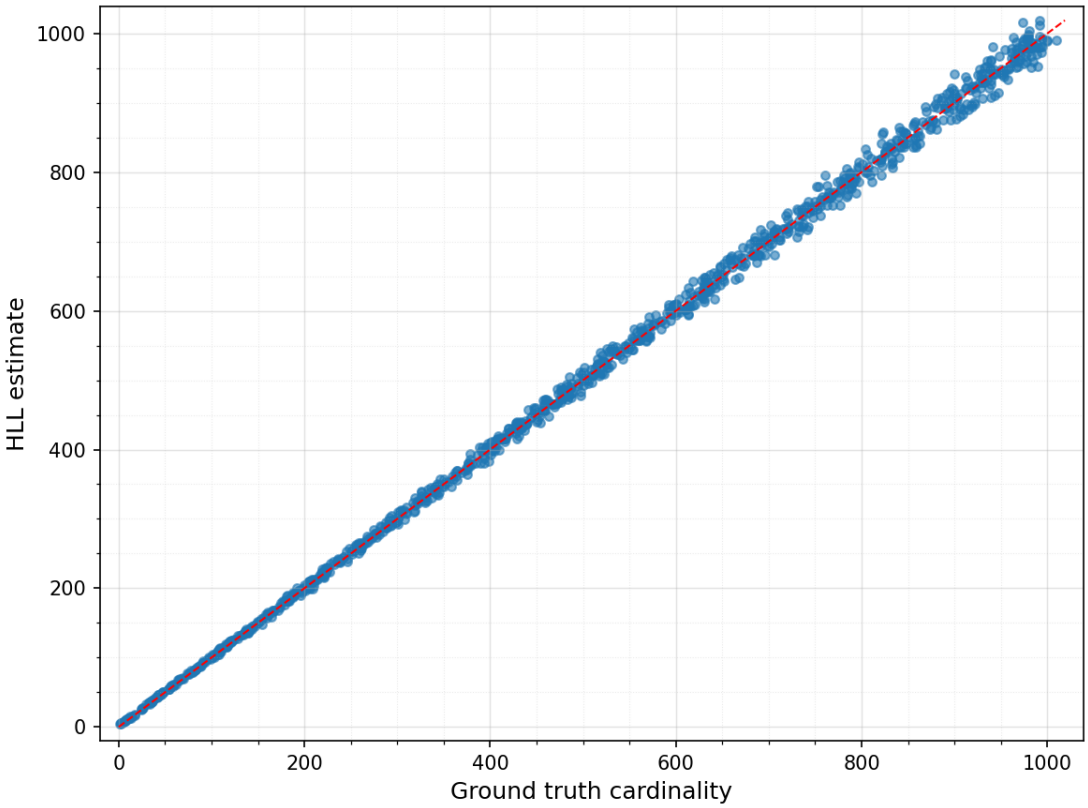


Figure 4.6. HyperLogLog estimates vs. ground truth for synthetic traffic with 1s read interval and 10s reset interval (linear scale). The extended monitoring window improves accuracy to 2.65% average error and 1.32% median error over 1,066 measurements, with cardinality range extending to approximately 1,000 flows.

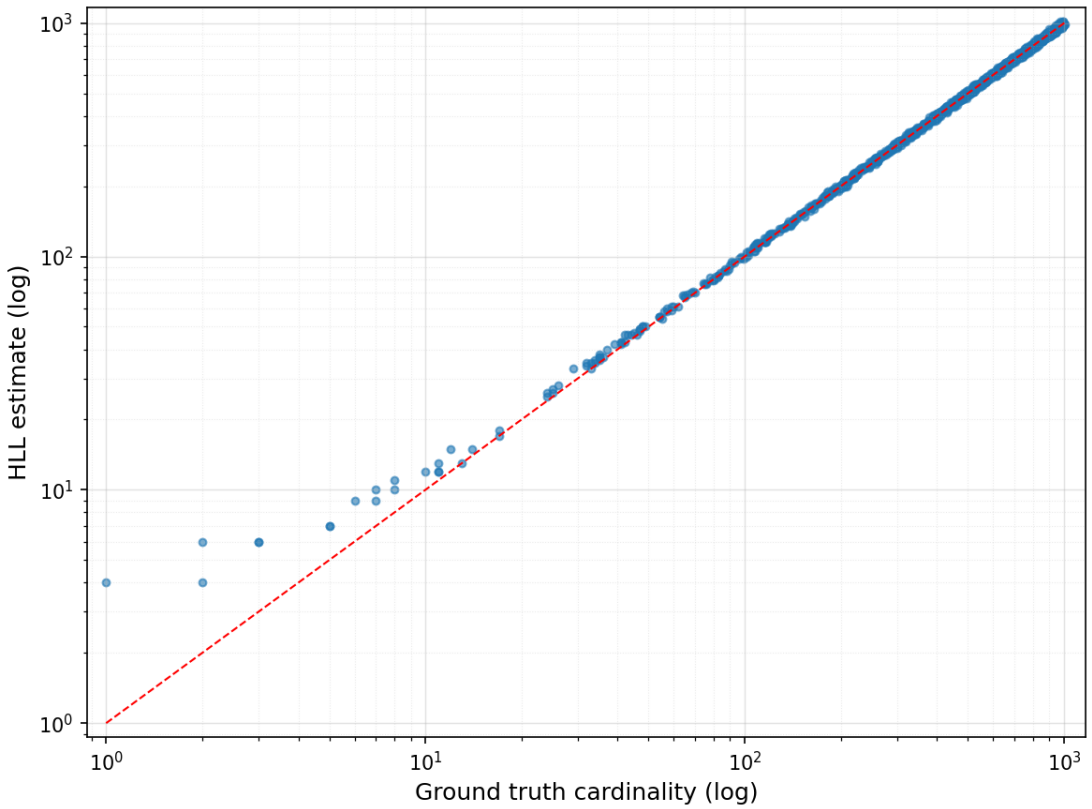


Figure 4.7. HyperLogLog estimates vs. ground truth for synthetic traffic with 1s read interval and 10s reset interval (log scale). Tighter clustering around the ideal line compared to the 0.5s configuration demonstrates improved accuracy with longer observation periods, approaching the theoretical HLL standard error of 2.3%.

**Configuration 3: 5s Read, 50s Reset** Performance improves substantially with 50s windows (Figures 4.8 and 4.9): average error of 1.47% and median error of 1.21% across 273 measurements tracking up to 5,000 flows. Both metrics fall well below the theoretical bounds which is 2.3%. With 5,000 flows across 2048 buckets, virtually all buckets capture observations, providing robust harmonic mean estimates (Equation 3.3). Tight clustering at both scales confirms excellent accuracy throughout the range.

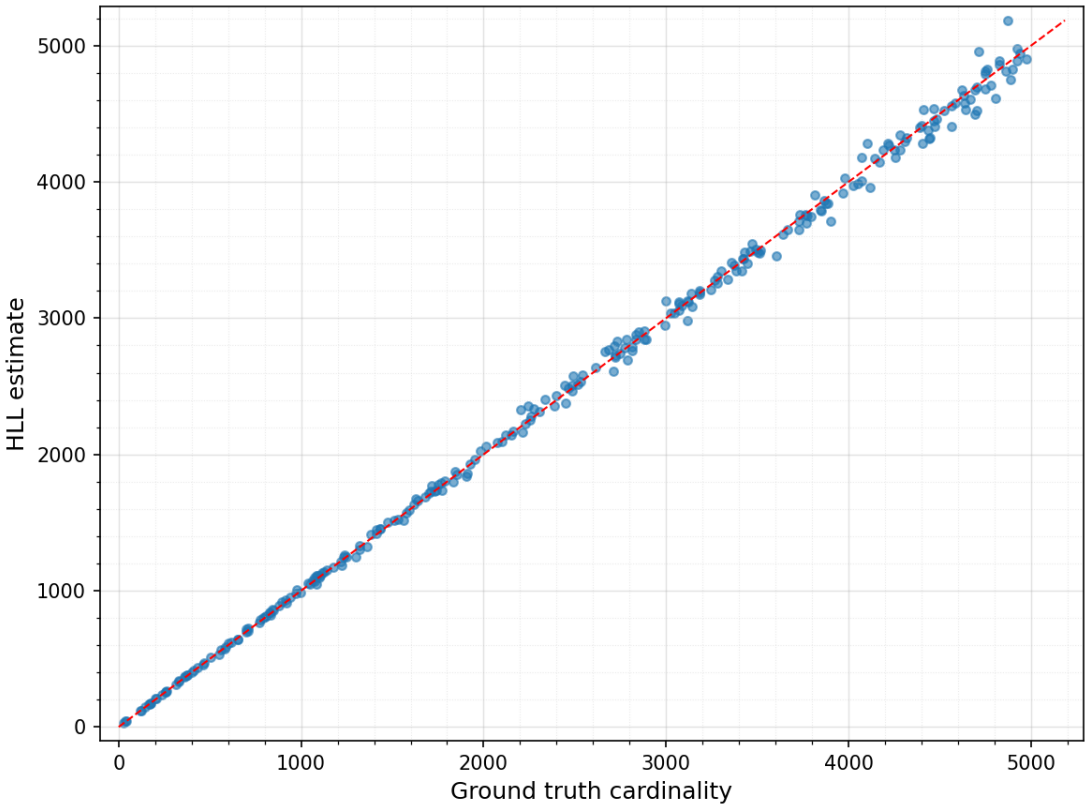


Figure 4.8. HyperLogLog estimates vs. ground truth for synthetic traffic with 5s read interval and 50s reset interval (linear scale). Substantially improved performance with 1.47% average error and 1.21% median error across 273 measurements, accurately tracking cardinalities up to 5,000 flows.

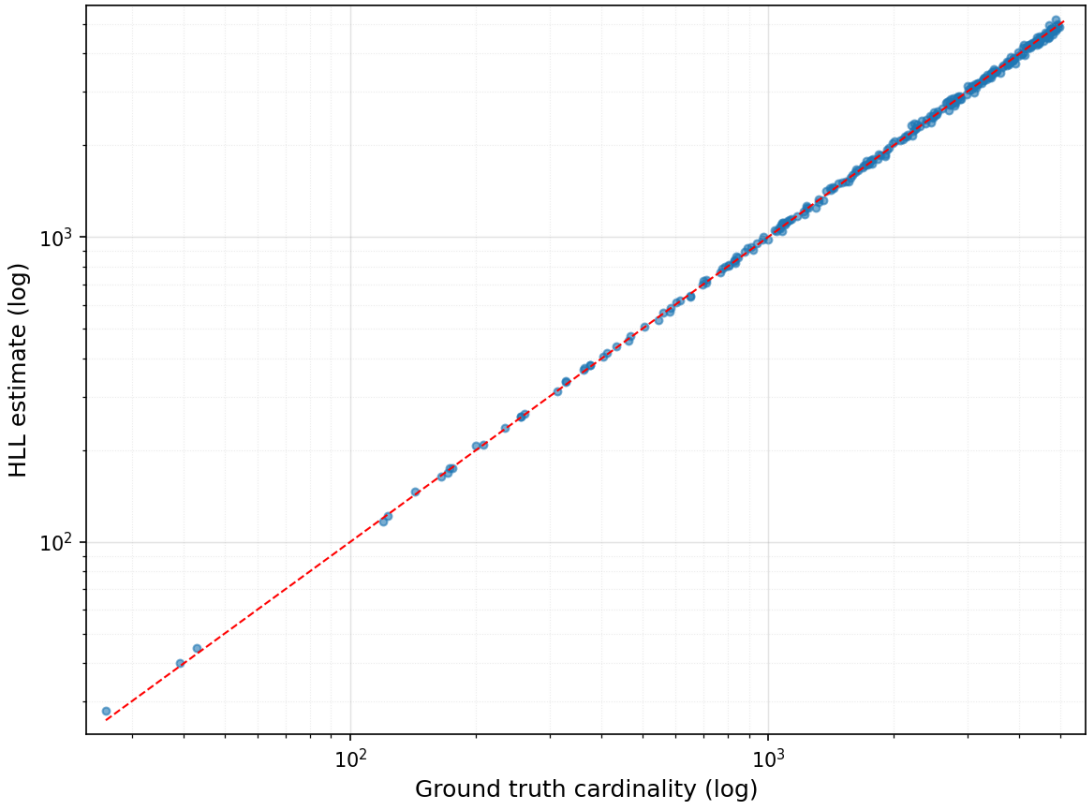


Figure 4.9. HyperLogLog estimates vs. ground truth for synthetic traffic with 5s read interval and 50s reset interval (log scale). Excellent accuracy maintained throughout the entire cardinality range with very tight clustering around the ideal diagonal line.

**Configuration 4: 10s Read, 100s Reset** The longest window achieves near-optimal performance (Figures 4.10 and 4.11): average relative error of 1.41% and median error of 1.16% over 133 measurements tracking up to 10,000 flows. The remarkably tight clustering across the full dynamic range shows that the implementation is well within theoretical HyperLogLog accuracy bounds.

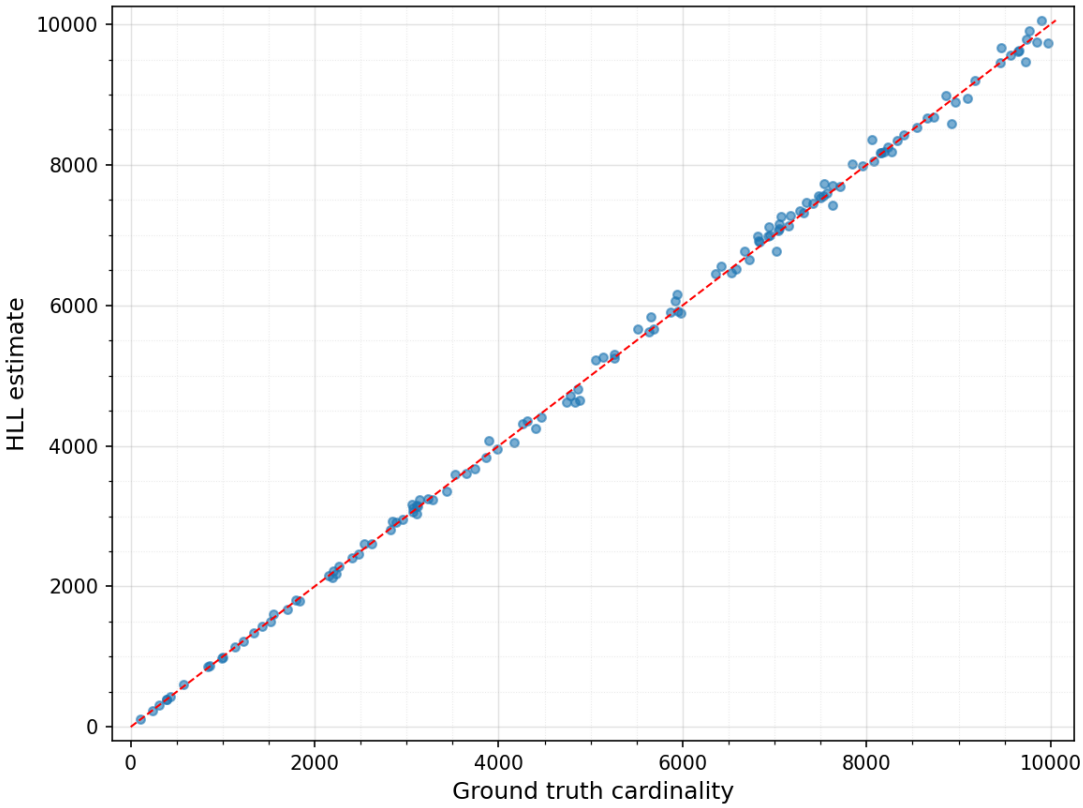


Figure 4.10. HyperLogLog estimates vs. ground truth for synthetic traffic with 10s read interval and 100s reset interval (linear scale). Best accuracy achieved with 1.41% average error and 1.16% median error over 133 measurements, accurately estimating cardinalities up to 10,000 flows.

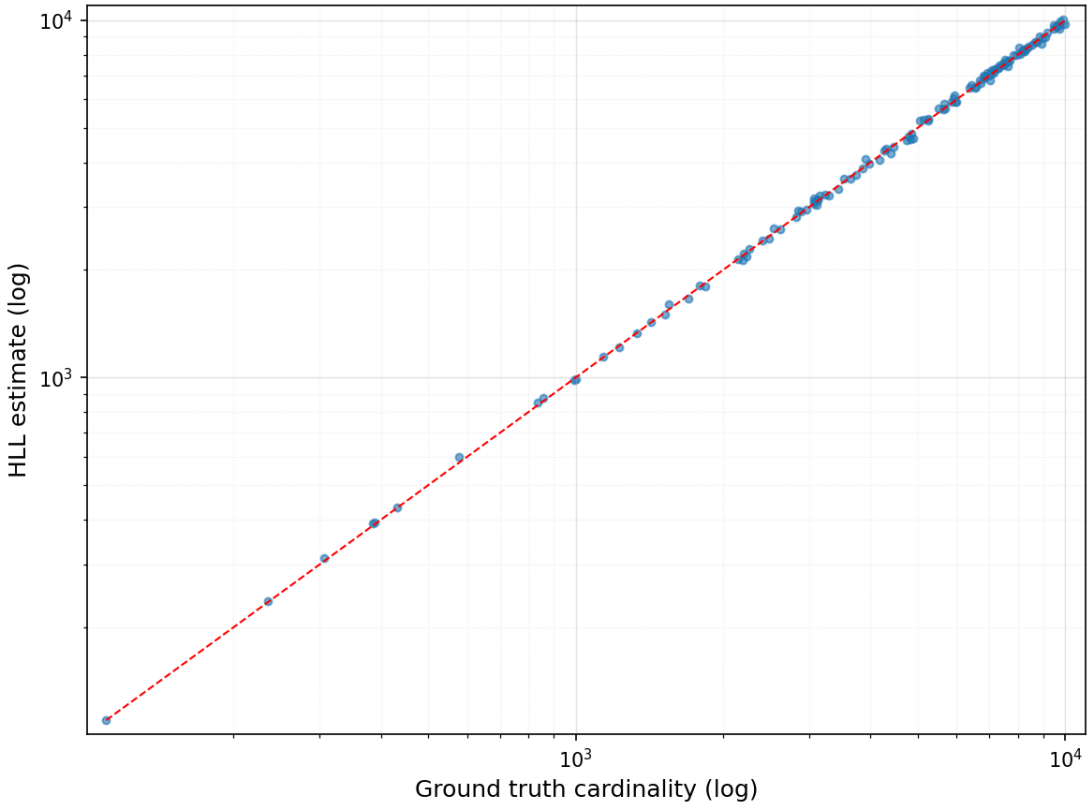


Figure 4.11. HyperLogLog estimates vs. ground truth for synthetic traffic with 10s read interval and 100s reset interval (log scale). Remarkably tight clustering across the full dynamic range is observed.

### 4.5.2 Real Network Trace Validation

#### Overview

We validate implementation accuracy under realistic conditions using traces from Polito campus network provided at `https://jupyter.polito.it/expert/`. Approximately 5M packets were replayed at 3,769 pps over 1,320 seconds, following the procedure in Section 4.3.2. We apply the same window configurations as in the synthetic experiments. Ground truth is computed using the software flow tracker (Section 4.4) with window synchronization and composite key matching (`window_id`, `packet_count`). Table 4.4 shows significant accuracy improvement with longer windows, though short-window errors are higher than in the synthetic case due to traffic burstiness and non-uniformity.

| Read (s) | Reset (s) | Avg. Error (%) | Median Error (%) | Samples | Max Card. (flows) |
|---|---|---|---|---|---|
| 0.5 | 5 | 8.51 | 2.56 | 1,620 | 2,000 |
| 1 | 10 | 5.34 | 2.63 | 984 | 2,800 |
| 5 | 50 | 2.32 | 2.05 | 244 | 7,000 |
| 10 | 100 | 1.69 | 1.40 | 135 | 10,000 |

Table 4.4. HyperLogLog accuracy for real network trace. Longer monitoring windows significantly reduce estimation errors and approach synthetic traffic performance.

#### Results Across Window Configurations

**Configuration 1: 0.5s Read, 5s Reset** The shortest window (Figures 4.12 and 4.13) tracks up to 2,000 flows over 1,620 measurements. The average error is 8.51% and the median error is 2.56%, both exceeding the theoretical bound of 2.3%. Compared to synthetic traffic (3.42% average, 1.46% median), real traffic shows 2.5 times higher average error and 1.75 times higher median error. This disparity stems from burstiness and non-uniform IP distributions in real traffic, which cause uneven bucket occupancies and increased variance. Scatter plots reveal significant deviation from the ideal line, particularly at lower cardinalities where bursts lead to overestimation.

**Configuration 2: 1s Read, 10s Reset** Doubling the window duration (Figures 4.14 and 4.15) yields substantial improvement: average error of 5.34% and median error of 2.63% over 984 measurements tracking up to 2,800 flows. The gap between real and synthetic traffic narrows to 2.0 times for the average error, down from 2.5 times, indicating that longer windows begin to average out burstiness, though occasional bursts still produce outliers. The median error remains slightly above the theoretical bound.

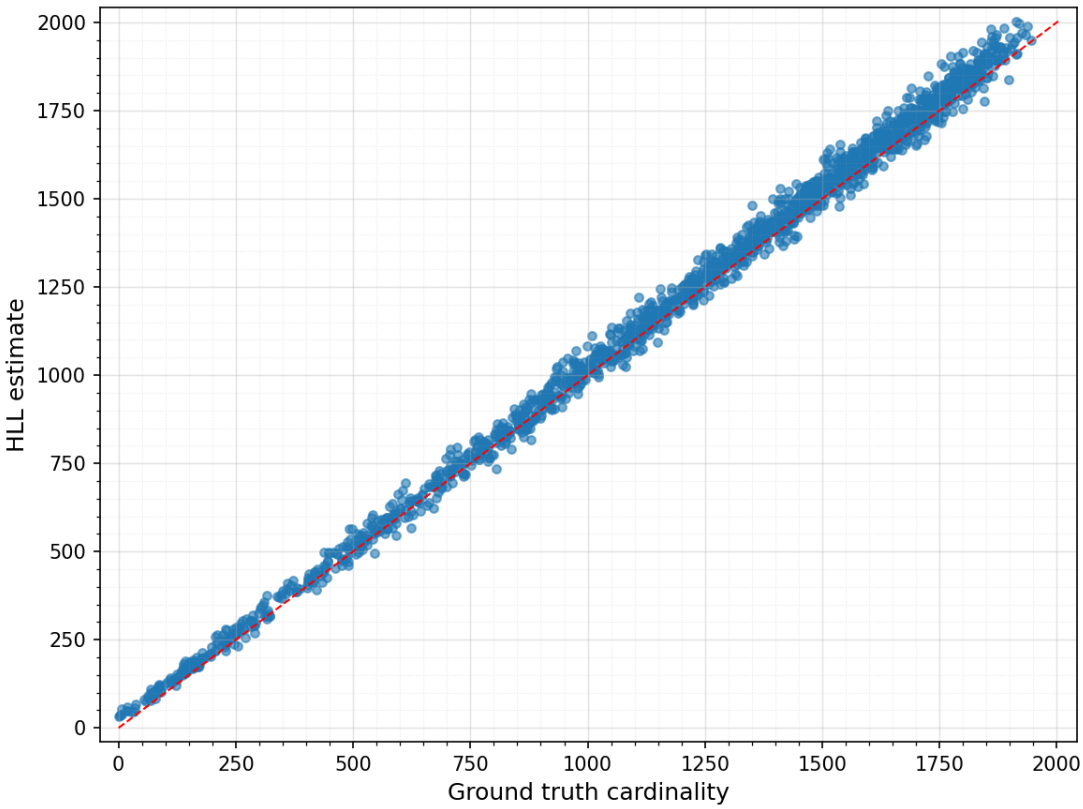


Figure 4.12. HyperLogLog estimates vs. ground truth for real network trace with 0.5s read interval and 5s reset interval (linear scale). Testing with real traffic at 3,769 pps yields 8.51% average error and 2.56% median error over 1,620 measurements, tracking cardinalities up to 2,000 flows.

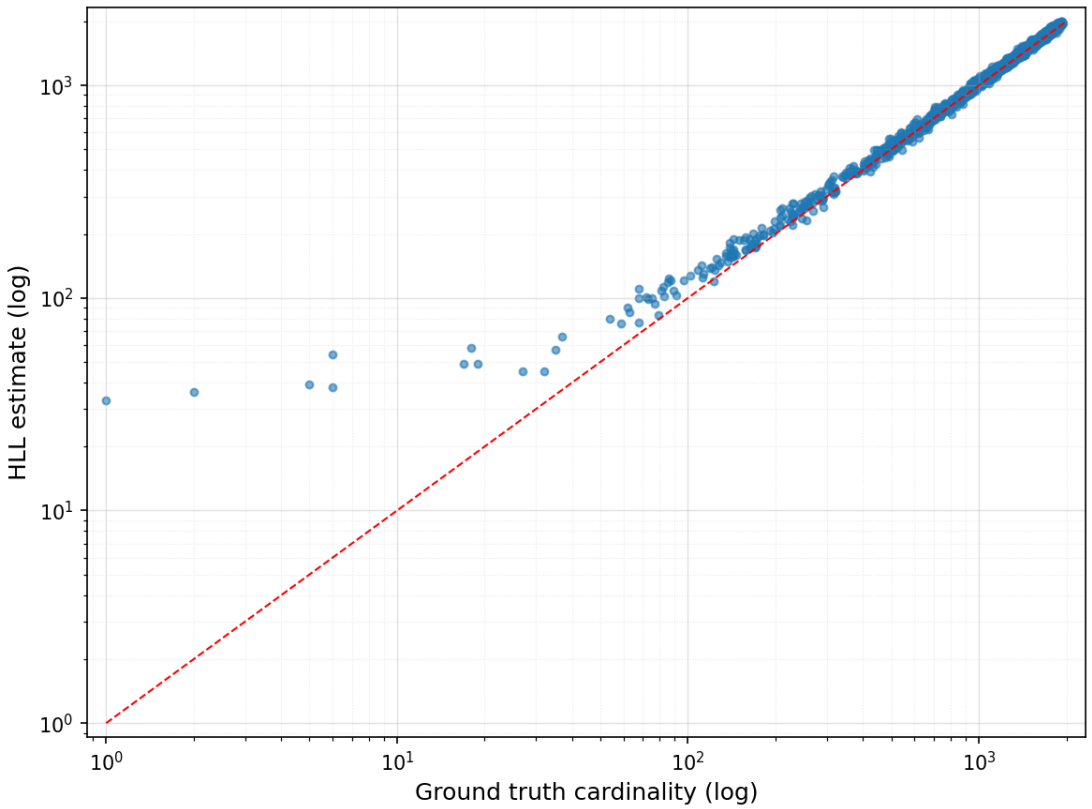


Figure 4.13. HyperLogLog estimates vs. ground truth for real network trace with 0.5s read interval and 5s reset interval (log scale). Increased scatter compared to synthetic traffic reflects the complex and variable nature of real network traffic patterns, leading to higher estimation errors, particularly at lower cardinalities.

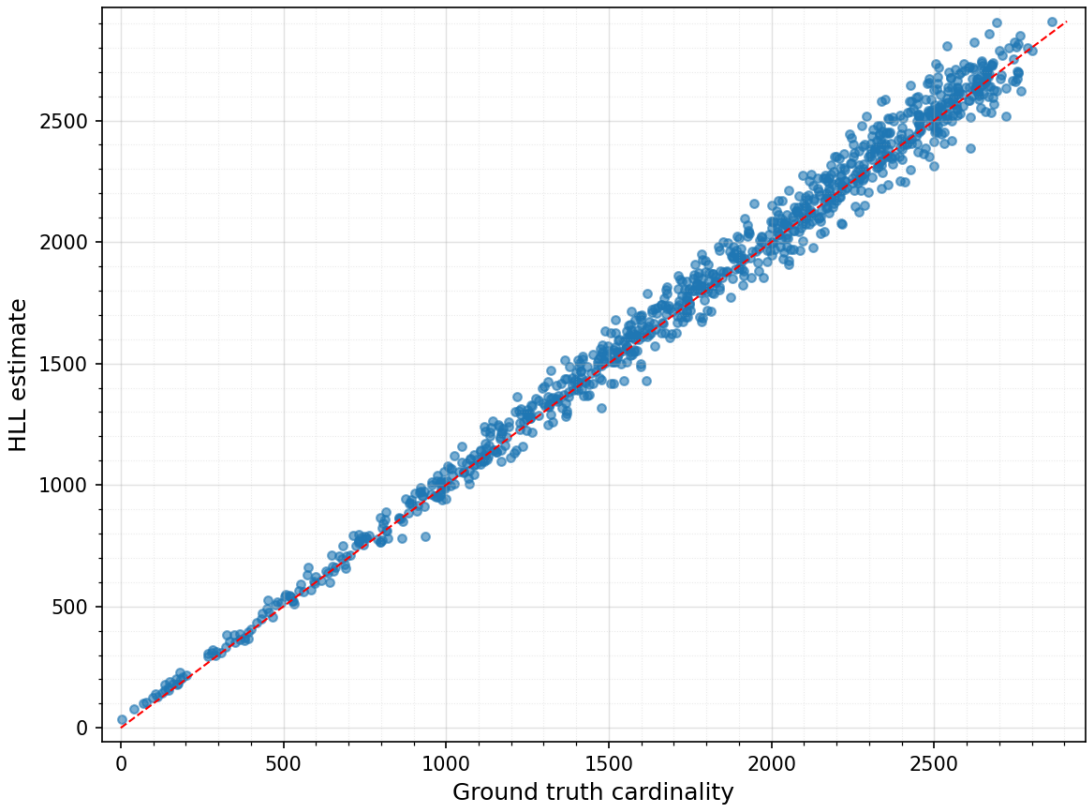


Figure 4.14. HyperLogLog estimates vs. ground truth for real network trace with 1s read interval and 10s reset interval (linear scale). Improved accuracy with 5.34% average error and 2.63% median error over 984 measurements, extending cardinality range to approximately 2,800 flows.

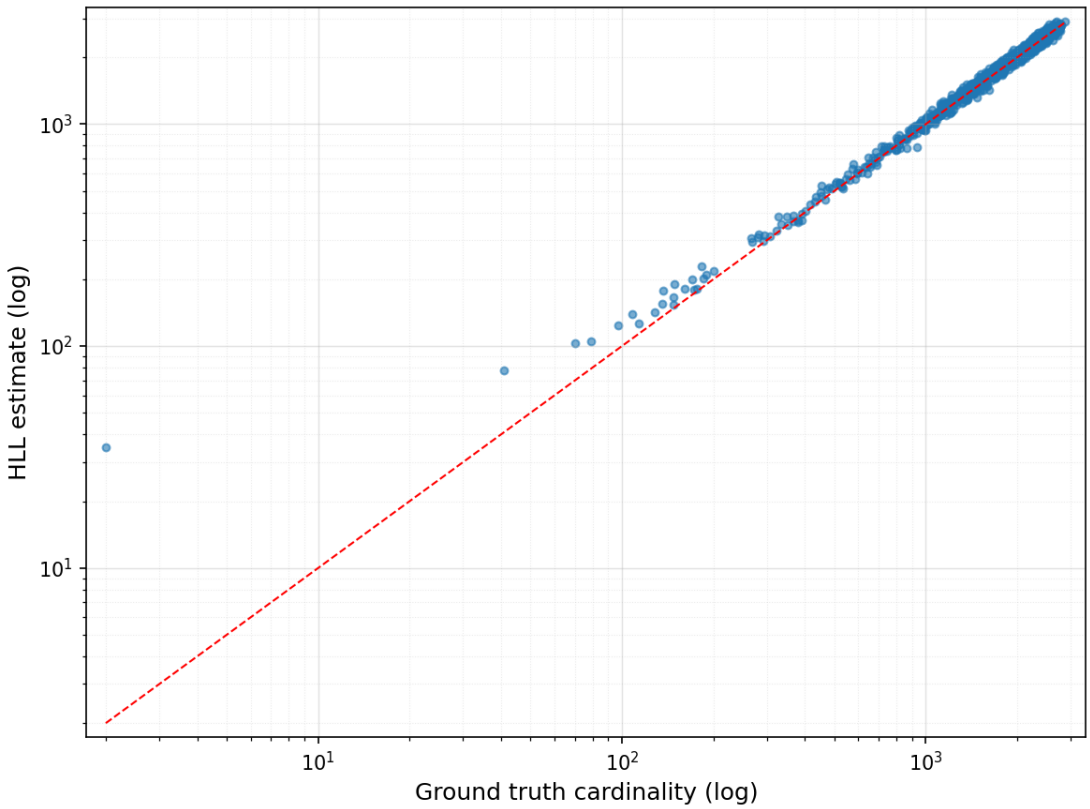


Figure 4.15. HyperLogLog estimates vs. ground truth for real network trace with 1s read interval and 10s reset interval (log scale). While visible scatter remains due to traffic burstiness, estimates generally follow the ideal line well across the dynamic range, showing reduced outlier frequency compared to shorter windows.

**Configuration 3: 5s Read, 50s Reset** Significant improvement appears at 50s windows (Figures 4.16 and 4.17): average error of 2.32% and median error of 2.05% across 244 measurements tracking up to 7,000 flows. The gap between real and synthetic traffic nearly closes, reaching 1.58 times for average error and 1.69 times for median error, demonstrating that longer windows effectively average out temporal variability.

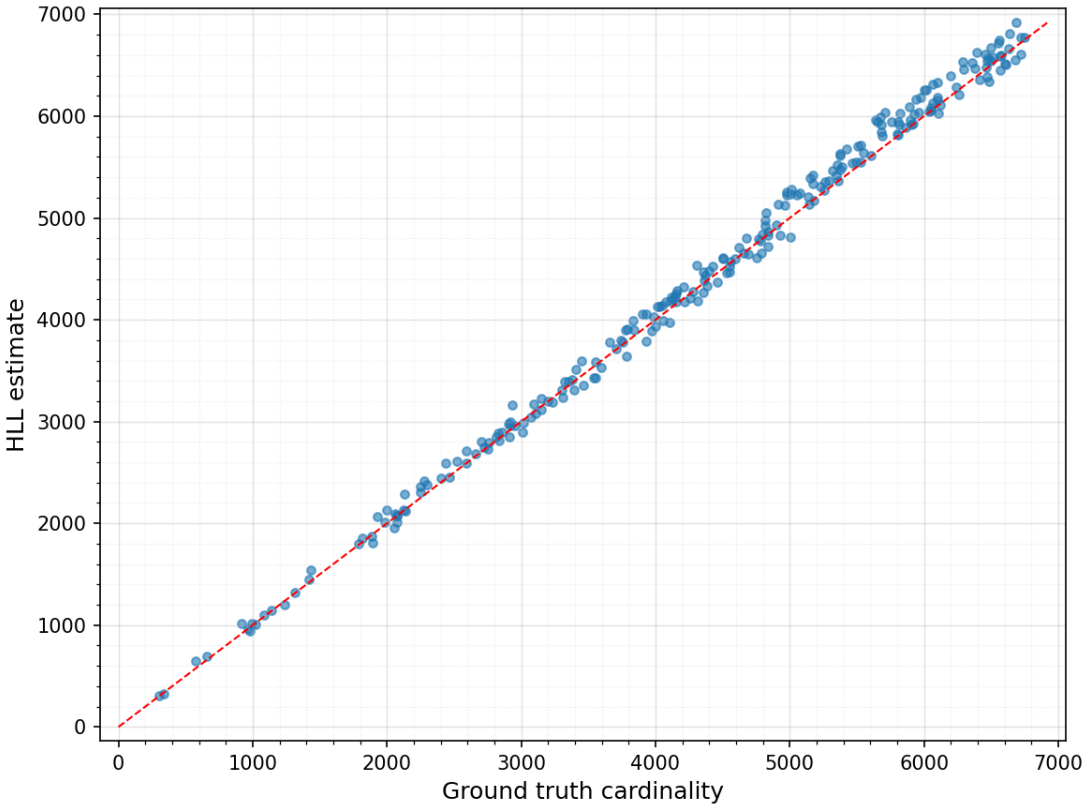


Figure 4.16. HyperLogLog estimates vs. ground truth for real network trace with 5s read interval and 50s reset interval (linear scale). Significant accuracy improvement to 2.32% average error and 2.05% median error across 244 measurements, accurately tracking cardinalities up to 7,000 flows.

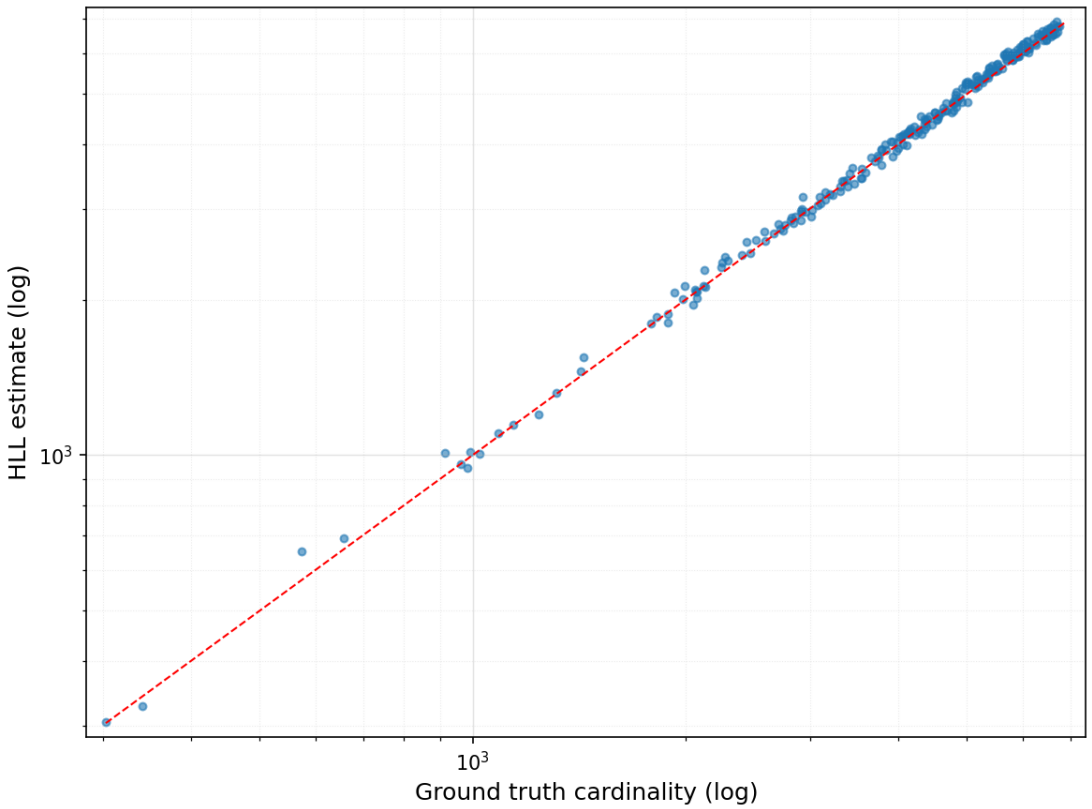


Figure 4.17. HyperLogLog estimates vs. ground truth for real network trace with 5s read interval and 50s reset interval (log scale). Much tighter clustering compared to shorter monitoring windows demonstrates that extended observation periods effectively mitigate realistic traffic variability, approaching theoretical HLL performance bounds.

**Configuration 4: 10s Read, 100s Reset** Excellent performance is achieved at the longest window (Figures 4.18 and 4.19): average error of 1.69% and median error of 1.40% ($0.61 \times \sigma_{\text{HLL}}$) over 135 measurements tracking up to 10,000 flows. Real and synthetic performance converge to within 1.20 times for average error and 1.21 times for median error, validating the hypothesis that sufficiently long windows make HLL performance largely independent of short-term traffic characteristics. Over 100 seconds, the system observes approximately 10,000 flows, providing enough samples for all 2048 buckets. The median error at 61% of theoretical closely matches the synthetic result (50%), demonstrating that W-HLL (Chapter 3) achieves near-optimal accuracy for realistic traffic when appropriate window sizes are chosen.

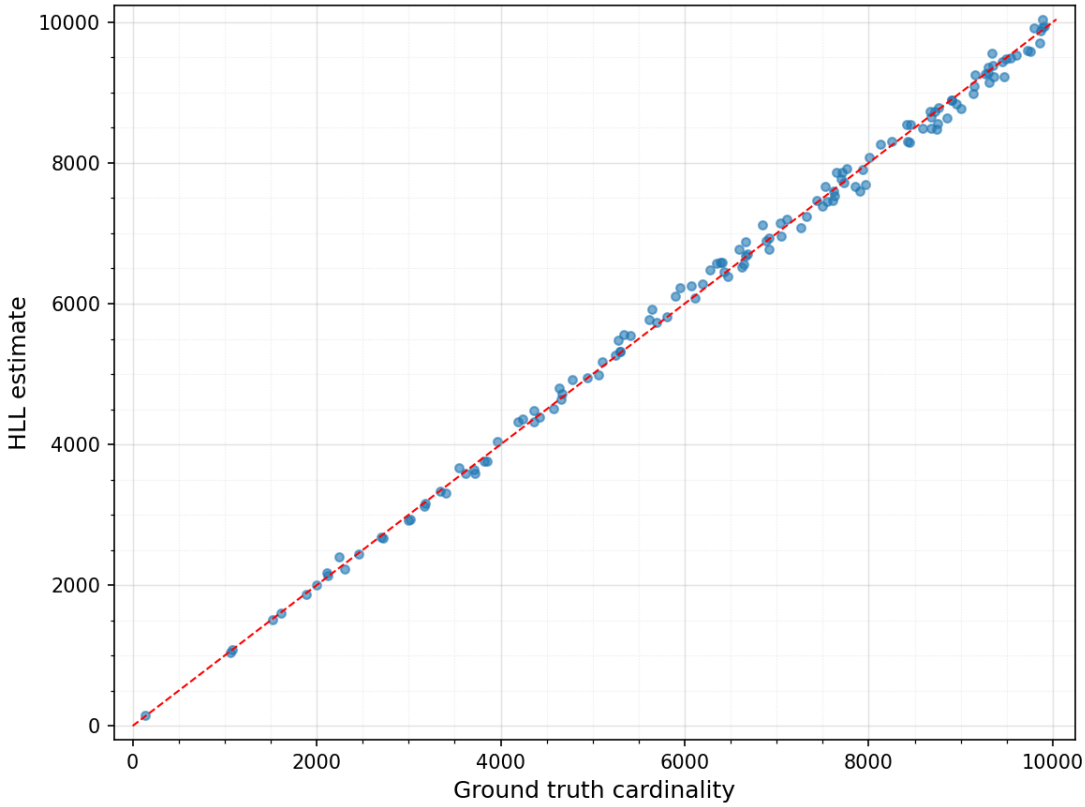


Figure 4.18. HyperLogLog estimates vs. ground truth for real network trace with 10s read interval and 100s reset interval (linear scale). Excellent performance with 1.69% average error and 1.40% median error over 135 measurements, approaching synthetic traffic accuracy across cardinalities up to 10,000 flows.

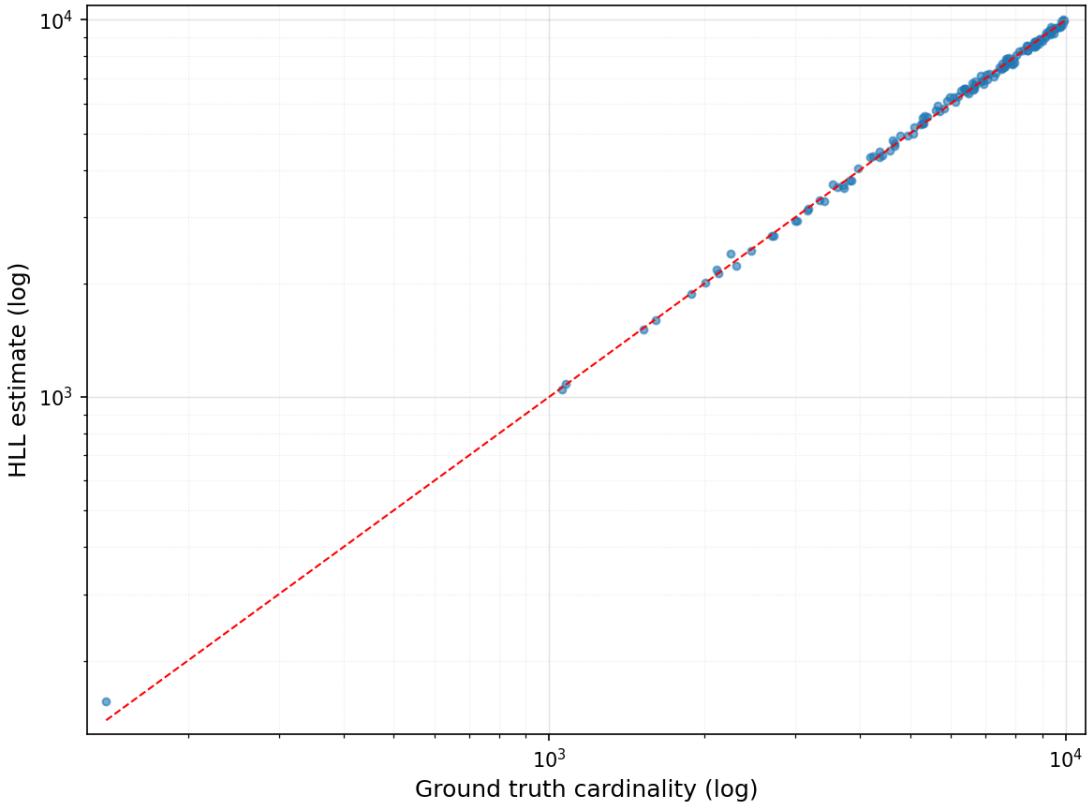


Figure 4.19. HyperLogLog estimates vs. ground truth for real network trace with 10s read interval and 100s reset interval (log scale). Very tight clustering throughout the full dynamic range demonstrates that longer observation periods allow the implementation to achieve high accuracy even with complex real-world traffic patterns, converging with synthetic traffic performance.

### 4.5.3 Results Analysis

Tables 4.3 and 4.4 reveal a clear pattern: as monitoring windows increase, real traffic performance converges toward synthetic traffic performance. The synthetic-to-real error ratio improves systematically:

- **5s window:** Real traffic shows 2.5 times higher average error (8.51% vs. 3.42%)
- **10s window:** The gap narrows to 2.0 times (5.34% vs. 2.65%)
- **50s window:** Further narrows to 1.6 times (2.32% vs. 1.47%)
- **100s window:** Nearly converged at 1.2 times (1.69% vs. 1.41%)

At the longest window (100s), both traffic types achieve similar accuracy: real traffic has a median error of 1.40% compared to 1.16% for synthetic, demonstrating that sufficient observation time makes HLL performance largely independent of traffic characteristics.

The HyperLogLog algorithm has a theoretical standard error of $\sigma_{\text{HLL}} = 1.04/\sqrt{2048} \approx 2.3\%$ for our 2048-bucket implementation [11]. Our experimental results validate this bound:

- **Synthetic traffic, 100s:** Median error of 1.16% (50% of theoretical)
- **Real traffic, 100s:** Median error of 1.40% (61% of theoretical)

Both cases achieve sub-theoretical performance, confirming that our P4 implementation correctly realizes the HLL algorithm without introducing additional error sources.

This experimental validation shows that windowed HyperLogLog on P4-programmable switches achieves practical accuracy for in-network flow cardinality estimation. The implementation handles both controlled synthetic traffic and complex real-world patterns effectively.

Key findings:

1. **Longer windows improve accuracy.** Average error drops from 8.51% to 1.69% for real traffic and from 3.42% to 1.41% for synthetic traffic as windows increase from 5s to 100s.
2. **Real and synthetic performance converge.** At 100s windows, the accuracy gap nearly closes (1.69% vs. 1.41%), indicating that sufficient observation time overcomes traffic variability.
3. **Theoretical bounds are met.** Both traffic types achieve errors below the 2.3% theoretical bound, confirming implementation correctness.

# Conclusion

Modern network infrastructures carry traffic at unprecedented scales. Hyperscale data centers and service provider networks handle millions of concurrent flows at aggregate throughputs measured in terabits per second. Operating such networks requires continuous visibility into traffic dynamics—understanding how many distinct flows traverse network elements, detecting anomalies that may signal attacks or failures, and planning capacity to meet evolving demands. Traditional exact counting approaches, which maintain per-flow state, cannot scale to these requirements: memory consumption grows linearly with flow count, and the processing overhead of maintaining hash tables or trees cannot sustain line-rate operation. Probabilistic data structures offer a compelling alternative, trading perfect accuracy for dramatic reductions in memory and computational complexity. Among these, HyperLogLog stands out for its ability to estimate cardinalities in the billions using only kilobytes of memory, with bounded relative error that does not depend on the actual count.

This thesis investigated the practical deployment of HyperLogLog for flow cardinality estimation directly within programmable network hardware. Rather than sampling packets to external collectors or processing flows in software, our approach executes the complete HyperLogLog algorithm in the data plane of Intel Tofino switches, achieving line-rate processing without per-packet control plane involvement. We argued that this method could provide memory-efficient cardinality estimation suitable for high-speed network monitoring, and that the emergence of P4-programmable switches makes such implementations feasible with current technology.

We began by introducing the necessary background. We reviewed the evolution of network programmability from Software-Defined Networking through PISA-based architectures, the P4 language and its runtime environment, and the HyperLogLog algorithm with its theoretical performance characteristics. We also presented the PASTA theorem, which guides our sampling strategy to avoid systematic bias. In Chapter 3, we presented our Windowed HyperLogLog (W-HLL) implementation, detailing the P4 data plane architecture that employs TCAM-based rank computation and atomic register updates, alongside the Python control plane that coordinates exponentially distributed observations and deterministic resets. We proposed a methodology based on synchronized window boundaries and composite key matching, enabling accurate comparison between HLL estimates and ground truth measurements.

We then assessed system performance through comprehensive experimental validation on the SUPERNET testbed. Our implementation achieves excellent accuracy across

diverse traffic conditions. For synthetic traffic with controlled ground truth, median estimation errors ranged from 1.46% at 5-second windows down to 1.16% at 100-second windows—the latter representing just 50% of the theoretical 2.3% standard error. Real network traces from the Politecnico di Torino campus network exhibited higher variability at short windows (8.51% average error at 5 seconds) due to traffic burstiness, but performance converged toward the synthetic results as window duration increased. At 100-second windows, real traffic achieved 1.40% median error, demonstrating that sufficient observation time largely overcomes the effects of non-uniform flow distributions and temporal correlations.

We confirmed that the accuracy gap between synthetic and real traffic narrows systematically with longer windows: from 2.5 times at 5 seconds to just 1.2 times at 100 seconds. This convergence validates the fundamental soundness of deploying probabilistic cardinality estimation in production environments, where traffic patterns inevitably differ from idealized distributions. The sub-theoretical error rates achieved in both scenarios confirm that our P4 implementation correctly realizes the HyperLogLog algorithm without introducing additional error sources from hardware constraints or pipeline limitations.

This research could be valuable in several network monitoring contexts. Flow cardinality estimation supports anomaly detection systems that identify distributed denial-of-service attacks through sudden increases in source diversity, network forensics that characterize traffic composition, and capacity planning that tracks connection density across network segments. The memory efficiency of HyperLogLog—two kilobytes for our 2048-bucket implementation—enables deployment at scale across many monitoring points without straining switch resources. Furthermore, the mergeable property of HyperLogLog sketches opens possibilities for hierarchical aggregation, where estimates from multiple switches combine to provide network-wide cardinality views.

We conclude by outlining further developments reserved for future work. A particularly promising direction involves implementing Staggered HyperLogLog (ST-HLL) [6], which extends the windowed approach developed in this thesis to enable near-continuous-time cardinality rate estimation. Rather than resetting all registers simultaneously as in our current W-HLL design, ST-HLL resets registers in a staggered fashion—one register per time slot—allowing the system to track flow arrival rates rather than absolute counts while maintaining the same memory footprint as vanilla HLL. This extension would be particularly valuable for anomaly detection applications where tracking the dynamics of flow spreading behavior—such as sudden increases in source diversity indicating distributed denial-of-service attacks—matters more than absolute cardinality counts.

Beyond ST-HLL, the methodology established here for HyperLogLog evaluation could be extended to other sketch-based algorithms—Count-Min Sketch for frequency estimation, or Bloom filters for membership queries—building toward a comprehensive toolkit for in-network traffic analysis on programmable switches.

# Bibliography


[1] P4 language and related specifications. https://p4.org/wp-content/uploads/sites/53/2024/10/P4-16-spec-v1.2.5.html, 2025. Accessed: November 24, 2025.

[2] M. Bonola, I. Drago, G. Petralia, S. Pontarelli, D. Rossi, G. Siracusano, G. Ventre, and G. Bianchi. Poise: Practical software defined intelligent ethernet switch. In *Proceedings of the 2018 IEEE Conference on Network Function Virtualization and Software Defined Networks (NFV-SDN)*, pages 1–7. IEEE, 2018.

[3] P. Bosshart, D. Daly, G. Gibb, M. Izzard, N. McKeown, J. Rexford, C. Schlesinger, D. Talayco, A. Vahdat, G. Varghese, et al. P4: Programming protocol-independent packet processors. *ACM SIGCOMM Computer Communication Review*, 44(3):87–95, 2014.

[4] P. Bosshart, G. Gibb, H.-S. Kim, G. Varghese, N. McKeown, M. Izzard, F. Mujica, and M. Horowitz. Forwarding metamorphosis: Fast programmable match-action processing in hardware for sdn. *ACM SIGCOMM Computer Communication Review*, 43(4):99–110, 2013.

[5] G. Cormode and S. Muthukrishnan. An improved data stream summary: the count-min sketch and its applications. *Journal of Algorithms*, 55(1):58–75, 2005.

[6] A. Cornacchia, G. Bianchi, A. Bianco, and P. Giaccone. Staggered hll: Near-continuous-time cardinality estimation with no overhead. *Computer Communications*, 193:168–175, 2022.

[7] M. Durand, P. Flajolet, and P. Nicodème. Loglog counting of large cardinalities. *Algorithmica*, 37(1):59–80, 2003.

[8] 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*, pages 323–336, 2002.

[9] C. Estan and G. Varghese. New directions in traffic measurement and accounting. *IEEE Network*, 17(3):40–48, 2003.

[10] N. Feamster, J. Rexford, and E. Zegura. *The Road to SDN: An Intellectual History of Programmable Networks*. ACM SIGCOMM Computer Communication Review, 2014.

[11] P. Flajolet, É. Fusy, O. Gandouet, and F. Meunier. Hyperloglog: the analysis of a near-optimal cardinality estimation algorithm. In *Proceedings of the 2007 Conference on Analysis of Algorithms (AofA)*, pages 137–156. Discrete Mathematics and Theoretical Computer Science, 2007.

[12] P. Flajolet and G. N. Martin. Probabilistic counting algorithms for data base applications. *Journal of Computer and System Sciences*, 31(2):182–209, 1985.

[13] G. Gibb, G. Varghese, M. Horowitz, and N. McKeown. Design principles for packet parsers. In *Proceedings of the Ninth ACM/IEEE Symposium on Architectures for Networking and Communications Systems (ANCS)*, pages 13–24. IEEE, 2013.

[14] Google. grpc: A high performance, open source universal rpc framework. https://grpc.io/, 2015.

[15] Google. Protocol Buffers: Google's Data Interchange Format. https://developers.google.com/protocol-buffers, 2024. Accessed: 2025-11-15.

[16] P. Gupta and N. McKeown. Algorithms for packet classification. *IEEE Network*, 15(2):24–32, 2001.

[17] S. Heule, M. Nunkesser, and A. Hall. Hyperloglog in practice: Algorithmic engineering of a state of the art cardinality estimation algorithm. In *Proceedings of the 16th International Conference on Extending Database Technology*, pages 683–692, 2013.

[18] Intel Corporation. Barefoot runtime (bfruntime) api. Barefoot SDE Documentation, 2020.

[19] Intel Corporation. Intel tofino: World's fastest p4-programmable ethernet switch asics. https://www.intel.com/content/www/us/en/products/network-io/programmable-ethernet-switch.html, 2020.

[20] D. Kreutz, F. M. Ramos, P. E. Verissimo, C. E. Rothenberg, S. Azodolmolky, and S. Uhlig. Software-defined networking: A comprehensive survey. *Proceedings of the IEEE*, 103(1):14–76, 2015.

[21] A. Kumar, J. Xu, J. Wang, O. Spatschek, and L. Li. Data streaming algorithms for efficient and accurate estimation of flow size distribution. In *ACM SIGMETRICS Performance Evaluation Review*, number 1, pages 177–188. ACM, 2004.

[22] N. McKeown, T. Anderson, H. Balakrishnan, G. Parulkar, L. Peterson, J. Rexford, S. Shenker, and J. Turner. Openflow: enabling innovation in campus networks. *ACM SIGCOMM Computer Communication Review*, 38(2):69–74, 2008.

[23] M. Mitzenmacher and E. Upfal. Measuring and mining the internet: Methods, applications, and future challenges. *ACM SIGCOMM Computer Communication Review*, 44(5):49–55, 2014.

[24] M. Moshref, M. Yu, R. Govindan, and A. Vahdat. Flowradar: A better netflow for data centers. In *Proceedings of the 13th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, pages 311–324, 2016.

[25] Open Networking Foundation. OpenFlow Switch Specification. Technical report, ONF, 2015. Accessed: 2025-11-15.

[26] P4.org API Working Group. P4Runtime Specification. Technical report, P4 Language Consortium, 2025. Version 1.5.0-dev, dated 2025-10-16. Accessed: November 24, 2025.

[27] W. W. Peterson and D. T. Brown. Cyclic codes for error detection. *Proceedings of the IRE*, 49(1):228–235, 1961.

[28] Politecnico di Torino. Supernet. http://restsrv01.polito.it/supernet, 2024. Accessed: 2024-11-22.

[29] S. M. Ross. *Introduction to probability models*. Academic press, 11th edition, 2014.

[30] M. Shahbaz, J. H. Choi, B. Pfaff, C. Kim, N. Feamster, J. Rexford, and R. Clark. Pisces: A programmable, protocol-independent software switch. In *Proceedings of the 2016 ACM SIGCOMM Conference*, pages 525–538, 2016.

[31] A. Sivaraman, A. Cheung, M. Kim, G. Varghese, M. Budiu, M. Alizadeh, H. Balakrishnan, S. Chole, G. Covington, A. Fingerhut, et al. Packet transactions: High-level programming for line-rate switches. In *Proceedings of the 2016 ACM SIGCOMM Conference*, pages 15–28, 2016.

[32] R. W. Wolff. Poisson arrivals see time averages. *Operations Research*, 30(2):223–231, 1982.

[33] Z. Xiong, Q. Liu, T. Wang, D. Li, and T. Zhang. Sketchlib: Enabling efficient sketch-based monitoring on programmable switches. In *USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, pages 1689–1706, 2023.

[34] M. Yu, L. Jose, R. Miao, J. Rexford, and M. J. Freedman. Software defined traffic measurement with opensketch. *Proceedings of the 10th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, pages 29–42, 2013.