SPADA: A Sparse Approximate Data Structure representation for data plane per-flow monitoring

ANDREA MONTERUBBIANO*, University of Rome La Sapienza, Italy
RAPHAEL AZORIN, Huawei Technologies Co. Ltd, France
GABRIELE CASTELLANO, Huawei Technologies Co. Ltd, France
MASSIMO GALLO, Huawei Technologies Co. Ltd, France
SALVATORE PONTARELLI, University of Rome La Sapienza, Italy
DARIO ROSSI, Huawei Technologies Co. Ltd, France

Accurate per-flow monitoring is critical for precise network diagnosis, performance analysis, and network operation and management in general. However, the limited amount of memory available on modern programmable devices and the large number of active flows force practitioners to monitor only the most relevant flows with approximate data structures, limiting their view of network traffic. We argue that, due to the skewed nature of network traffic, such data structures are, in practice, heavily underutilized, i.e., sparse, thus wasting a significant amount of memory.

This paper proposes a Sparse Approximate Data Structure (SPADA) representation that leverages sparsity to reduce the memory footprint of per-flow monitoring systems in the data plane while preserving their original accuracy. SPADA representation can be integrated into a generic per-flow monitoring system and is suitable for several measurement use cases. We prototype SPADA in P4 for a commercial FPGA target and test our approach with a custom simulator that we make publicly available, on four real network traces over three different monitoring tasks. Our results show that SPADA achieves 2× to 11× memory footprint reduction with respect to the state-of-the-art while maintaining the same accuracy, or even improving it.

CCS Concepts: • **Networks** → **Network measurement**; **Network monitoring**; *Programmable networks*; • **Theory of computation** → Data structures design and analysis.

Additional Key Words and Phrases: Data plane; Per-Flow Monitoring, Monitoring Data Structures

ACM Reference Format:
Andrea Monterubbiano, Raphael Azorin, Gabriele Castellano, Massimo Gallo, Salvatore Pontarelli, and Dario Rossi. 2023. SPADA: A Sparse Approximate Data Structure representation for data plane per-flow monitoring. *Proc. ACM Netw.* 1, CoNEXT3, Article 27 (December 2023), 25 pages. https://doi.org/10.1145/3629149

Authors' addresses: Andrea Monterubbiano, monterubbiano@di.uniroma1.it, University of Rome La Sapienza, Department of Computer Science, Viale Regina Elena, 295, 00161, Rome, Italy; Raphael Azorin, raphael.azorin@huawei.com, Huawei Technologies Co. Ltd, 18 quai du Point du Jour, 92100, Boulogne-Billancourt (Paris), France; Gabriele Castellano, gabriele.castellano@huawei.com, Huawei Technologies Co. Ltd, 18 quai du Point du Jour, 92100, Boulogne-Billancourt (Paris), France; Massimo Gallo, massimo.gallo@huawei.com, Huawei Technologies Co. Ltd, 18 quai du Point du Jour, 92100, Boulogne-Billancourt (Paris), France; Salvatore Pontarelli, pontarelli@di.uniroma1.it, University of Rome La Sapienza, Department of Computer Science, Viale Regina Elena, 295, 00161, Rome, Italy; Dario Rossi, dario.rossi@huawei.com, Huawei Technologies Co. Ltd, 18 quai du Point du Jour, 92100, Boulogne-Billancourt (Paris), France.

2834-5509/2023/12-ART27 $15.00
https://doi.org/10.1145/3629149

1 INTRODUCTION

Monitoring network traffic on a per-flow basis requires measuring several quantities related to the packets traversing network devices. These accurate measures provide network operators the necessary data for fine-grained Operations, Administration, and Management (OAM) algorithms such as responsive diagnosis [14, 50], precise fault localization [6, 39], traffic engineering [11], network accounting [23], anomaly detection [65], and many others. However, collecting per-flow metrics requires a considerable amount of resources, especially at high speed when the number of active flows might be very high. Considering that recent studies estimate the number of active flows in the order of 100K per Gbps of traffic [49], accurately monitoring Tbps of traffic might require several GBs for a few per-flow metrics. Programmable ASIC devices feature memories in the order of dozens of MBs to accommodate all network applications, including L2/L3 forwarding, among others. Consequently, the amount of memory allocated for monitoring is but a fraction of the total one, making accurate per-flow monitoring impractical due to its high memory requirements. Similarly, in FPGA-based SmartNICs, the amount of memory available for per-flow monitoring is scarce since the use of large memories such as Double Data Rate Synchronous Dynamic Random Access Memory (DDR-SDRAM) or High Bandwidth Memory (HBM) is limited by their high access latency, i.e., tens of clock cycles, and the required resource-hungry cache memory hierarchy.

To better understand the per-flow monitoring problem, Figure 1ab illustrates at a high level how flows f_{1-K} are typically mapped to dedicated or shared arrays of counters (sketches). In particular, we consider a few concrete use cases: *super spreader detection*, *per-flow quantiles*, and *flow size* estimation. The goal of super spreader detection algorithms [28, 33, 59] is to estimate, for any given source IP (sIP), its "cardinality", i.e., the number of destination IPs (dIPs) it contacts. This task is often achieved using the HyperLogLog (HLL) [24] sketch, allocating one HLL data structure for each sIP. Unfortunately, achieving good accuracy with HLL requires a considerable amount of memory. For this reason, practitioners usually limit the number of monitored sIP to reduce memory occupancy, or decrease the accuracy, e.g., vHLL [33, 61]. Similarly, monitoring per-flow quantiles [16, 32] of relevant flow properties, e.g., packet Inter-Arrival Time (IAT), packet size, etc., requires processing the stream of all packets belonging to the same unidirectional 5-tuple using histogram-based data structures such as DDSketch [40]. However, using a dedicated sketch for each monitored flow makes quantile estimation challenging due to the high memory requirements. It is worth noting that even for basic measurements such as flow size estimation, the amount of memory required to monitor hundreds of thousands of flows is not negligible with ElasticSketch [63]. We remark that the above-mentioned monitoring algorithms, as well as other existing ones [5, 15, 17, 29, 38, 45, 56], require the allocation of a sketch composed by an array of shared or dedicated counters (also called buckets, bins) for each monitored flow (cf. Figure 1ab): this frequently translates into significant memory requirements making it difficult to deploy them even for only a fraction of the active flows. Therefore, we assert the need to reduce the memory requirements of per-flow network monitoring algorithms, enabling them to coexist with other critical network functions in the data plane.

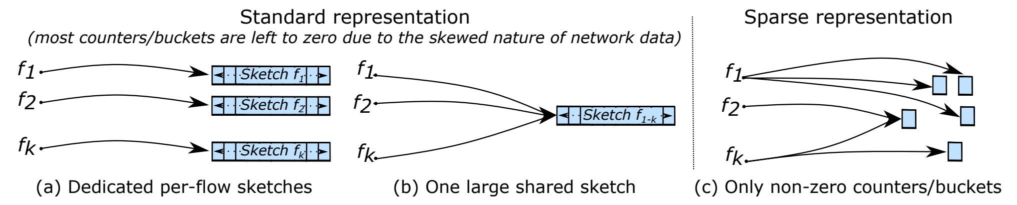
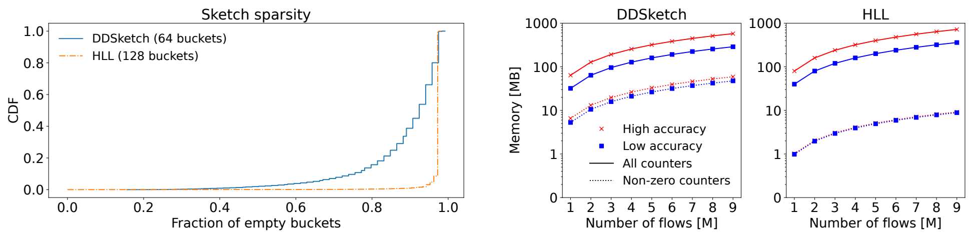

Standard representation
(most counters/buckets are left to zero due to the skewed nature of network data)
Sparse representation
f1
f2
fk
Sketch f1
Sketch f2
Sketch fk
Sketch f1-k
(a) Dedicated per-flow sketches
(b) One large shared sketch
(c) Only non-zero counters/buckets

Fig. 1. Per-flow monitoring using standard **(a)**, **(b)** and sparse **(c)** representations.



(a) Sketch counters sparsity.

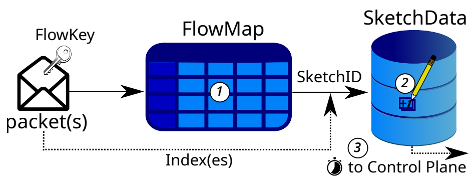
(b) Memory requirement.

Fig. 2. Flow sparsity **(a)** and memory requirements **(b)** analysis using one sketch per flow from a CAIDA trace.

Main idea and contributions. In this paper, we show that sizeable gains can be attained by *reducing the amount of unnecessarily allocated memory without compromising the monitoring accuracy.* In a nutshell, we remark that per-flow sketches are designed for extreme scenarios and hence underutilized in most cases. To solve this problem, we propose a data structure representation built around the simple concept of only storing counters that are actually used, as depicted in Figure 1c. To provide a quantitative idea of the under-utilization of sketches in typical measurement tasks, Figure 2a reports the CDF of the fraction of empty buckets in *high accuracy* HLL and DDSketch (128 and 64 buckets) on a CAIDA trace for 700K flows. In both cases, per-flow sketches are highly underutilized, i.e., 80% of per-flow DDSketches feature at least 80% of empty buckets, even more for HLL. This results in a huge waste of memory as observed in Figure 2b, which contrasts the memory occupied by the sketches with the one required by only non-zero counters: the picture shows one-to-two orders of magnitudes of potential memory savings, which holds even for *low accuracy* HLL and DDSketch (64 and 32 buckets). We remark that similar sparsity issues arise in different use cases where sketch-based data structures are employed, [5, 15, 17, 29, 38, 45, 56, 63]. Thus, it can be argued that sparsity is a property common to many measurement tasks.

Sparse monitoring data structure representations are not trivial to implement in networking because it is not possible to know a priori which flow will lead to sparse data. We acknowledge that sparse data representation is a well-known technique widely used in many fields ranging from signal processing to machine learning [12, 21, 48, 60, 64, 66] – yet, to the best of our knowledge, it has never been explored in the context of network monitoring. To support sparse monitoring data structures in the data plane we propose a Sparse Approximate Data Structure (SPADA) representation, which stores only relevant, i.e., non-zero, sketch counters. One key challenge of implementing sparse representations in the data plane is storing *<key, value>* pairs for the non-zero counters, taking into account hardware limitations and without a priori knowledge of flow sparsity. To address this challenge, we propose two alternative solutions: *(i)* a Cuckoo Hash Table with quotienting (qCHT), which can accommodate any kind of data at the price of non-constant insertion time, and *(ii)* a novel data structure, Perfect Invertible Bloom Lookup Table (pIBLT), featuring constant insertion time but only suitable for counter-based sketches. Our main contributions are as follows:

1. we introduce a sparse representation in the context of a generic data plane monitoring system which is beneficial for several use cases by reducing memory footprint with no accuracy penalty;
2. we design a batched Cuckoo Hash Table (CHT) compatible with modern programmable pipelines as it offers approximately constant insertion time by imposing limited recirculation overhead;
3. we design a novel data structure pIBLT, which improves over a traditional IBLT removing false positives at the cost of a small bitmap;
4. we provide simulation code and results with the CAIDA and MAWI datasets on three use cases, assessing memory reduction from 2× to over 11× with respect to the state-of-the-art;
5. we implement SPADA pipeline in P4 and benchmark it on a Xilinx FPGA-based SmartNIC target.



Use case	Data structure†	FM	+	SD	Recirc.
ⓐ	HLL - sta.	MAT	+	qCHT	✓
	HLL - dyn.	CHT	+	qCHT	✓
ⓑ	DDSketch - sta.	MAT	+	pIBLT	×
	DDSketch - sta.	MAT	+	qCHT	✓
	DDSketch - dyn.	CHT	+	pIBLT	✓
	DDSketch - dyn.	CHT	+	qCHT	✓
ⓒ	ES - dyn.	ES Heavy	+	pIBLT	×
	ES - dyn.	ES Heavy	+	qCHT	✓

† Flows are either statically inserted by the control plane in a simple Match-Action Table (MAT), or dynamically inserted by the data plane.

Fig. 3. Generic data plane monitoring system (left) and summary of SPADA configurations (right).

SPADA in a nutshell. We provide a high-level overview of SPADA, with the help of the generic per-flow monitoring data plane pipeline depicted in Figure 3 (left). First ①, a flow key (e.g., IP address or 5-tuple) is extracted at packet arrival and mapped to a `SketchID` via a *Flow Map* (FM). Second ②, the sketch counters corresponding to that flow are updated in a *Sketch Data* (SD) store. Note that a wide set of network monitoring systems [9, 15, 61, 63, 67] can be cast into this two-stage monitoring system data plane. Finally ③, the measurements are exported or used locally for traffic management: as the focus of this paper is on the design and implementation of the data plane, the control plane is limited to the collection of sketches at the end of a measurement epoch.

A SPADA representation consists of a compression of the *Sketch Data* by exploiting its sparsity. To showcase the generality of SPADA, we consider three popular measurement use cases and two alternative SPADA configurations summarized in Figure 3 (right). Considered use cases are ⓐ super spreader detection, ⓑ per-flow packet IAT quantile estimation, and ⓒ flow size estimation, which can be performed using state-of-the-art sketches such as HLL [24], DDSketch [40] and ElasticSketch [63] respectively. For each use case, we consider two SPADA implementations, where flows are inserted in the *Flow Map* either statically by the control plane (*sta*), hence monitoring a predefined set of flows, or dynamically by the data plane (*dyn*), as new flows are received.

Paper outline. In Section 2, we review state-of-the-art sketches that can benefit from SPADA and detail three use cases we use as examples throughout the paper. In Section 3, we introduce the SPADA representation within a standard per-flow monitoring system. We then, provide its memory occupancy analytical model, and a detailed memory sizing discussion. The SPADA FPGA-based P4 implementation is described in Section 4. Trace-based simulation results and prototype assessment are reported in Section 5. Section 6 discusses the related work, and Section 7 concludes the paper.

2 USE CASES

In this section, we first identify a set of sparse monitoring data structures, then we detail state-of-the-art sketches used for three use cases: ⓐ super spreader detection (HLL), ⓑ per-flow packet IAT distribution (DDSketch), and ⓒ flow size estimation (ElasticSketch). Given their popularity and generality, we use them to showcase the benefits of SPADA throughout the rest of the paper.

Sparse monitoring data structures. We report in Table 1 a list of sketches that can exploit sparsity. Note that the actual benefit of the sparse representation depends on several factors, such as the number of flows under monitoring, the required accuracy, and the traffic skewness. Generally speaking, a sparse representation of a sketch can be beneficial when it is allocated per-flow (i.e., a sketch for each monitored flow) since, due to the natural skewness of traffic, many sketches will be significantly sparse as motivated before (e.g., use cases ⓐ, ⓑ). Another scenario in which a sparse representation is useful is when the sketch must provide high accuracy and thus reduce the collision probability (e.g., use case ⓒ) by increasing the sketch size. We note that with appropriate sampling strategies [10], any sketch can be sparsified if a small loss in accuracy is acceptable.

Table 1. Sketches that feature sparse data. Marked (✓) ones are used as reference in our analysis.

Measure	**Sketch name**	**Description**	**Note**
Cardinality (use case ⓐ)	HLL [24] ✓	Array of counters	Skewness similar to the HLL sketch used as reference for use case ⓐ.
	PCSA [25]	Array of bit-vectors	
	KVM [7]	Stores up to k minimum values	
	Fast-AGMS [17]	Array of counters	
	BeauCoup [15]	Bitvector	
Quantile (use case ⓑ)	DDSketch [40] ✓	Array of counters	Skewness similar to DDsketch used as reference for use case ⓑ. High-level KLL compactors not fully allocated.
	Circllhist [29]	Array of counters	
	KLL [34]	Array of compactors	
Flow Size (use case ⓒ)	Count-Min Sketch [18]	Array of counters	Sparse when high accuracy is needed.
	ECM-sketches [45]	Count-min sketch over sliding windows	
	Elastic Sketch [63] ✓	Split heavy and light flows	Similar to CMS, the light part can be sparsified for high accuracy.
	LearnedSketch [30]	Split heavy and light flows with ML	
Entropy	EntropySketch [38]	Up to k counters for stream entropy estimation.	

ⓐ **Super spreader detection.** HLL [24] is a data structure for cardinality estimation based on the probabilistic counting method developed in [25]. An HLL sketch is composed of m counters c_i, and for each item x, one of the counters is selected using a hash function and updated as follows $c_i = max(c_i, \rho(h(x)))$, where $\rho()$ denotes the position of the leftmost 1 in $h(x)$ binary representation. After seeing n distinct items, from a statistical point of view, c_i roughly approximates $log_2(n)$. Hence, the overall cardinality can be estimated as the harmonic mean of the values 2^{c_i} from the HLL counters. On the one hand, using a large number of counters m reduces the estimation error. On the other hand, the use of many counters leads to sparse HLLs, as the number of distinct flow counters updated is proportional to the cardinality itself, which is typically small. For example, on a CAIDA trace, less than 10% of sIP have a cardinality higher than 3 (90% of the HLLs only use three out of the m allocated counters). However, knowing in advance which flows will have high cardinality is challenging. Given this large number of unused counters, moving to a sparse HLL representation can significantly reduce the memory footprint of a super spreader detection system.

ⓑ **Packet IAT distribution.** DDSketch [40] is a data structure used to estimate the quantiles for a set of real positive values. Given a multiset S of size n over $\mathbb{R}$, the q-quantile $x_q \in S$ is the item x whose rank $R(x)$ in the sorted multiset S is $\lfloor 1 + q(n-1) \rfloor$ for $0 \leq q \leq 1$, where the rank $R(x)$ is the number of elements in S smaller than or equal to x. A DDSketch comprises m counters tracking the number of values falling in a range $[\gamma^{i-1}, \gamma^i]$, with γ depending on the required accuracy [40]. To insert a new measured value v in the DDSketch, its corresponding index is computed, and the counter at that index is incremented by 1. The q-quantile estimation $\hat{x}_q$ is computed by iteratively summing the counters from the first one, stopping once the sum is bigger than $q(n-1)$. The quantile is then estimated by $2\gamma^w/(\gamma+1)$ where w is the bin from the last iteration. DDSketch can be modified to accept a limited number of bins m depending on the desired accuracy α. It is worth highlighting that in per-flow IAT monitoring, most DDSketch counters are left to zero, as most flows consist of only a few packets, and samples, e.g., IAT, tend to cluster around a few values. Thus, an efficient sparse DDSketch representation would significantly reduce its memory requirements.

ⓒ **Flow size estimation.** ElasticSketch [63] is among the state-of-the-art for flow size estimation. It comprises i.e., *heavy* and *light* parts storing elephant and mouse flows respectively. A technique called *ostracism* is used to identify the mouse flows that will be monitored using the light part. The heavy part comprises multiple hash tables with per-flow counters, while the light part is a Count-Min Sketch (CMS) [18]. One of the effects of segregating elephants in the heavy part is that the CMS can be composed of a single row ($d = 1$) of 8-bit counters, thanks to the reduced number of flows and their size. To achieve good accuracy, the CMS still needs to be dimensioned to keep the amount of collisions considerably low. This means budgeting a large number of counters, even though most of them are left to zero making the CMS sparse.

3 DESIGN

In this section, we first describe the architectural components of a generic monitoring system data plane, cf. Figure 3. We then detail and contrast two approaches: a baseline that allocates one full sketch per monitored flow, and its corresponding SPADA representation that only stores non-zero counters. We propose two alternative SPADA representations that improve over the baseline in terms of memory footprint with different trade-offs. Finally, we discuss the pros and cons of each representation and analyze SPADA memory sizing.

3.1 Architectural components

Flow Map. In the data plane, a first component performs a `FlowToSketch` mapping to associate incoming flow packets to a specific sketch in the system. This *Flow Map* takes as input a flow key (e.g., the packet TCP/IP 5-tuple) and outputs a `SketchID`. Note that the way the mapping is performed might depend on the monitoring use case. The mapping can be *direct*, i.e., a flow stored in the *Flow Map* is associated with a specific sketch (cf. Figure 1a), or *indirect*, i.e., if a flow is not in the *Flow Map*, then it is associated with a default sketch as in [63] (cf. Figure 1b). Finally, flows stored in the *Flow Map* can be associated with additional flow metadata, e.g., last packet timestamp.

Sketch Data. The `SketchID` retrieved from the *Flow Map* is used to access the measurement counters that need to be updated for a particular flow. These counters are stored in the second component, which we refer to as *Sketch Data*. Based on the monitoring task, this component may store different things. For instance, in the case of per-flow IAT quantile estimation, the `SketchID` is used to access a dedicated per-flow DDSketch. Aside from the specific way information is structured within the *Sketch Data*, from a high-level perspective it is an architectural component that stores one or multiple sketches and provides access to the monitoring counters associated with a specific flow, sometimes shared with other flows.

Monitoring routine. In a measurement epoch, the system data plane updates the counters stored in the *Sketch Data* based on incoming packets, as depicted in Figure 3 (left). First, at packet arrival, the flow key (e.g., 5-tuple) is extracted and the *Flow Map* is queried to retrieve the associated `SketchID`, optionally updating any flow metadata. Second, the `SketchID` is used to access the *Sketch Data* and locate the sketch for the flow. Finally, depending on the monitoring task, the measurement associated with the last packet is used to determine one index *within* the sketch and modify the corresponding counter. For IAT quantile estimation for example, the last IAT value is mapped to a sketch bucket with the DDSketch algorithm, and the counter therein is incremented. At the end of the epoch, the *Flow Map* and *Sketch Data* are read by the control plane, which reconstructs per-flow sketches by looking up all counters belonging to a flow and computes the required metrics.

Notations. SPADA design is based on the assumption, justified by our experiments, that among m sketch counters the ratio p of non-zero ones is typically small. In order to quantify the memory footprint of the monitoring system data plane, we define $n_u = p \cdot m$ as the average number of sketch counters different than zero, i.e., the lower n_u, the higher the sparsity. With SPADA, we provide a series of memory reduction techniques whose efficiency is inversely proportional to n_u. To analytically estimate the efficiency of SPADA, we derive their memory footprint by using the notation reported in Table 2.

Table 2. Summary of main notation.

Symbol	Meaning	Symbol	Meaning
n_s	number of sketches to store (e.g., one per flow)	s_c	size (in bits) of a single sketch counter
s_k	size (in bits) of the flow key	p	ratio of useful (non-0) counters in a sketch
m	number of sketch counters (buckets)	n_u	average number of non-0 counters in a sketch

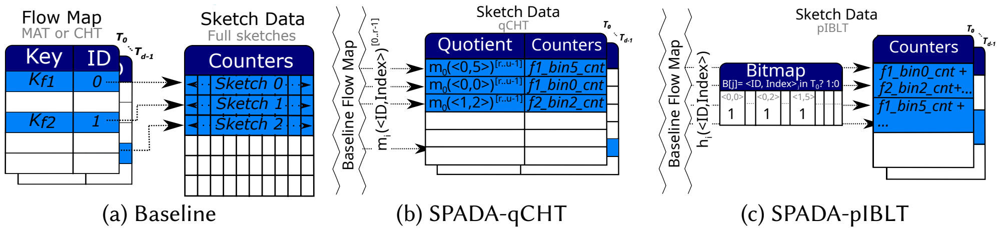


Fig. 4. **(a)** Baseline *Flow Map* and *Sketch Data* representations in a traditional per-flow monitoring data plane, and **(b)**, **(c)** apply SPADA representation to the SketchData through two possible implementations.

3.2 Baseline components

Flow Map. A simple *direct* `FlowToSketch` mapping can be realized using a MAT to provide a unique `SketchID` for every incoming flow. Given the huge number of entries required, this approach is not suitable for handling previously unknown flows. Instead, it requires populating the *Flow Map statically* from the control plane with a known set of flows to monitor. Another solution consists of using a hash of the flow key as `SketchID`. Although this simple approach suffers from collisions, it can be used when approximate measurements are acceptable. Between these two simple options, there is a wide range of possibilities trading off accuracy for memory efficiency.

A unique, direct, `FlowToSketch` mapping can be realized with a CHT. CHT provides a constant lookup time and a high memory utilization, thus is widely used to implement this kind of per-flow mapping. However, this solution requires a careful design: whereas cuckoo hashing is feasible in programmable data planes, it relies on several stages and has non-constant insertion time which is handled by packet recirculation that may severely impact the system performance. In Section 4, we provide P4 implementation details and show how to reduce recirculation impact. Regardless of the *Flow Map* implementation, whenever a new `FlowToSketch` mapping is needed, i.e., when a packet from a previously unseen flow key is received, a new `SketchID` can be obtained from a free ID counter. Then, the `<flow key, SketchID>` mapping is stored in the CHT. Note that using a counter to generate unique `SketchIDs` does not constitute a limitation since the system is designed for epoch-based measurements and the counter is reinitialized at the beginning of each epoch. Finally, we note that an indirect `FlowToSketch` mapping can be realized using the *Flow Map* as a filter. This means that flows stored in the *Flow Map* are monitored using their metadata while the remaining ones are monitored via a separate sketch, as it is done in ElasticSketch [63].

Sketch Data. A simple way of storing measured values is to allocate enough memory for all the counters required by every per-flow sketch. Hence, in the baseline implementation, the *Sketch Data* component is a list of full sketches, indexed through the `SketchID` that corresponds to the location of the first bucket of the sketch as depicted in Figure 4a. For instance, in the case of per-flow IAT quantile estimation, the `SketchID` is used to locate the first bin of the DDSketch dedicated to a particular flow. At that point, the bin to be modified is retrieved simply as an offset from the `SketchID` (i.e., `SketchID` + bin). Note that, depending on the specific task, a dedicated per-flow sketch might not be required, e.g., ElasticSketch only requires a single Count-Min Sketch.

Memory footprint. We derive the memory needed by the baseline described above assuming that, within one epoch, the monitoring system requires n_s different sketches, i.e., one sketch per flow, considering n_s expected different flows. With a CHT[1] *Flow Map*, the memory requirement is:

$$Memory = n_s \cdot (Row_{FM-CHT} + Row_{SD-Base}) \tag{1}$$

[1]We note that the CHT cannot be filled up to 100% [36] and the expected number of flows needs to be overestimated.

where Row_{FM-CHT} and $Row_{SD-Base}$ are the memory required to store an entry in the *Flow Map* and a sketch in the *Sketch Data* respectively. Row_{FM-CHT} and $Row_{SD-Base}$ can be defined as:

$$Row_{FM-CHT} = s_k + \lceil log_2(n_s) \rceil \qquad Row_{SD-Base} = m \cdot s_c$$

where s_k and $\lceil log_2(n_s) \rceil$ are the flow key and the `SketchID` bit sizes respectively, while m and s_c are the number of sketch counters and their size in bits. Note that independently from the Sketch Data, the Flow Map size could be further decreased by saving key fingerprints i.e., less than s_k bits.

3.3 The SPADA representation

In this section, we describe the SPADA representation for the *Sketch Data*. The main idea is to replace per-flow sketches with a series of non-zero counters, addressable with the pair `<SketchID, index>` where `index` is the sketch counter position in the logical per-flow sketch. The key difference with respect to the baseline is that the memory required by each sketch depends on the number of its non-zero elements. Indeed, while the baseline *Sketch Data* statically reserves an entire sketch upon receiving the first packet of a flow, SPADA creates a "virtual sketch" by reserving a unique `SketchID` whenever a new flow is received, and dynamically assigns pre-reserved counters whenever a new `<SketchID, index>` pair is required. We design two possible implementations of the sparse *Sketch Data*: the first one uses a qCHT, while the second one uses a modified version of the Invertible Bloom Lookup Table (IBLT), namely perfect IBLT (pIBLT). While the former has the drawback of requiring data plane recirculation, it provides additional flexibility compared to the pIBLT, as the latter can only be used with sketches whose update operation consists of a linear increase.

3.3.1 Sparse Sketch Data with qCHT. This version of the *Sketch Data* is based on a Cuckoo Hash Table (CHT), a key-value data structure with constant lookup time. As shown in Figure 4b, we use `<SketchID,index>` pairs (of u bits) as CHT keys and use it to store non-zero counters. A CHT comprises d tables ($d = 4$ in our settings), hence `<SketchID,index>` pairs are hashed d times to generate one index per table. The CHT stores both key and value, and key conflicts are resolved by moving entries across the various tables; this comes at the price of a non-constant insertion time. To reduce the size of keys in the table, SPADA uses a CHT with quotienting (qCHT). A qCHT relies on d bijective functions to hash the keys, uses the least r significant bits of the hashes to index the tables, and only stores the remaining $u - r$ bits for conflict resolutions instead of the whole key of u bits. A full description of the CHT and of the quotienting technique is provided in Appendix A.1.

Memory footprint. Before analytically deriving the memory required by a qCHT *Sketch Data*, let us first analyze the simpler case of a sparse *Sketch Data* using a CHT:

$$Memory = n_s \cdot (Row_{FM-CHT} + n_u \cdot Row_{SD-CHT}) \tag{2}$$

where the first part of the equation is the memory required for a CHT *Flow Map* as for the baseline while the second part is the size of one entry in the sparse *Sketch Data* with CHT, multiplied by the expected number of non-zero sketch counters n_u. We can then express Row_{SD-CHT} as follows:

$$Row_{SD-CHT} = (\lceil log_2(n_s \cdot m) \rceil) + s_c$$

where $\lceil log_2(n_s \cdot m) \rceil$ is the size of each key stored in the CHT, i.e., `<SketchID, index>`, and s_c the size of each counter in bits. Now that we have derived the memory requirement of the sparse *Sketch Data* using a simple CHT, let us detail the case when using a qCHT. Since we use a qCHT composed of 4 tables, each table contains up to $(n_s \cdot n_u)/4$ elements. Therefore, it can be addressed by using only $r = \lceil log_2(n_s \cdot n_u/4) \rceil$ bits instead of $u = \lceil log_2(n_s \cdot m) \rceil$, i.e., the full length of the key. To detect collisions, the remaining $u - r$ quotient bits are stored in the table:

$$u - r = \lceil log_2(n_s \cdot m) \rceil - \lceil log_2(n_s \cdot n_u/4) \rceil \approx 2 + log_2(m/n_u) = 2 + log_2(1/p).$$

The memory requirements for the sparse *Sketch Data* using qCHT can be summarized as:

$$Row_{SD-qCHT} \approx (2 + log_2(1/p)) + s_c.$$

It is worth noting that $u - r$ is small since the sparse *Sketch Data* stores a significant fraction of the items in the `<SketchID, index>` universe. This is a quite different setting compared to the *Flow Map* one, as in the latter case the key size may exceed 100 bits, leading to negligible savings.

Example. To provide a rough idea of SPADA-qCHT memory saving, we report here a concrete monitoring use case for ⓑ IAT quantile estimation. We use the 5-tuple as flow key, a DDSketch with $m = 64$ counters, i.e., bins, of size $s_c = 8$ bits. Assuming a conservative bin usage ratio[2] of $p = 0.15$ and small $n_s = 100K$ number of sketches, the baseline implementation requires 7.9 MB overall, of which 6.4 MB for the *Sketch Data* store. SPADA-qCHT requires a 1.5 MB *Flow Map* and a 1.5 MB sparse *Sketch Data*, for 3 MB overall (62% memory saving). Note that savings result from both the quotienting technique and sparse representation: a sparse *Sketch Data* using CHT without quotienting would lead to an overall memory footprint of 5.2 MB (35% memory saving).

Limitations. SPADA-qCHT has two main drawbacks. First, each non-zero counter requires more memory with respect to its baseline counterpart. Indeed, we store the quotient of the key `<SketchID, index>` in addition to the counter itself. However, this overhead is greatly compensated by the fact that only non-zero counters are stored. Memory saving thus depends on the sparsity factor p: the lower p, the more negligible this per counter overhead, leading to higher memory savings. A quantitative analysis of this effect can be appreciated in Section 3.4 where we show memory savings at different sparsity factors. The second drawback is that CHTs in programmable data planes are challenging as they require a non-constant number of memory accesses at insertion time, which might not be acceptable on constrained hardware. In Section 4.2 we address this challenge and propose a CHT implementation compatible with programmable data planes.

3.3.2 Sparse Sketch Data with pIBLT. To overcome the limitations of qCHT, we propose an alternative SPADA representation based on Invertible Bloom Lookup Table (IBLT), a structure that provides key-value counters with a fixed number of memory accesses. IBLT aggregates multiple keys within the same bucket, while each key is stored in d separate buckets in order to resolve collisions (cf. Appendix A.2). Using IBLT introduces two challenges. First, the per-bucket overhead is higher than the qCHT, as each entry requires a key counter i.e., the number of keys stored in a bucket, the XOR of all the colliding keys, and the sum of all the values associated to such keys. Second, to extract stored values, the IBLT uses a peeling procedure named `ListEntries`, that imposes a strict upper bound to the load factor, i.e., 82% achieved using $d = 3$ [58].

We propose an improved IBLT that uses a bitmap B to keep track of `<SketchID, index>` pairs i.e., it features $2^u = n_s \cdot m$ bits, one for each possible pair. We call this modified structure perfect IBLT (pIBLT). Differently from a IBLT, our pIBLT does not need to store XORed keys to take note of which ones contribute to the corresponding counter. Instead, such information is retrieved from the bitmap B. Besides, the bitmap removes false positives and does not require peeling to retrieve entries that can derived by solving a system of linear equations $A \cdot \mathbf{x} = \mathbf{b}$, where $a_{ij} \in \{0, 1\}$ indicates whether key_j contributed to counter b_i. We detail the new `ListEntries` procedure[3] in Algorithm 1. The *Sketch Data* implemented using pIBLT is illustrated in Figure 4c. We use a pIBLT with $d = 4$ tables, each indexed using a different hash function and featuring $n_s \cdot n_u/4$ locations. Each location simply stores the sum of the counters. At each update, the d counters associated with the `<SketchID, index>` pair are incremented, and the corresponding bit in the bitmap is set.

[2]Sparsity factors extracted from real traffic traces used in the evaluation of Section 5 are between 0.003 and 0.16.
[3]Note that the `ListEntries` procedure is only executed in the control plane after a measurement epoch. Therefore, the time needed to solve the linear system does not introduce any overhead in the data plane monitoring pipeline.

Algorithm 1 pIBLT ListEntries

```
Require: $B[], T_0[] \dots T_{d-1}[]$, Initialize: all $a_{ij} = 0$; initialize $b_i$ values from counters in $T_0[] \dots T_{d-1}[]$
1: for $j \in [0 \dots 2^u - 1]$ do
2:     if B[j]==1 then
3:         save key $key_j$ (i.e., a pair <SketchID, index>)
4:         for all $i \in \{h_0(key_j), \dots, h_{d-1}(key_j)\}$, set $a_{ij} = 1$
5: solve the linear system $A \cdot \mathbf{x} = \mathbf{b}$
6: return pairs $key_j, x_j$ for all saved keys
```

It can be proven [19, 20, 46] that $A \cdot \mathbf{x} = \mathbf{b}$ has a unique solution with high probability (cf. Appendix A.2) when the load factor is below a threshold c_k. For $d = 4$, $c_k = 0.97$, which is higher than the IBLT peeling threshold i.e., 0.82. Even if the probability that $A \cdot \mathbf{x} = \mathbf{b}$ does not have a unique solution is small, we remark that in this case, it is possible to derive an approximate resolution. For example, in PR-Sketch [53] an iterative method provides the solution with minimum ℓ_2-norm. FlowLidar [42] presents an alternative method based on an initial peeling phase removing dependent rows from A.

Memory footprint. For the pIBLT, the memory requirements can be expressed as:

$$Memory = n_s \cdot (Row_{FM-CHT} + n_u \cdot s_c) + n_s \cdot m \tag{3}$$

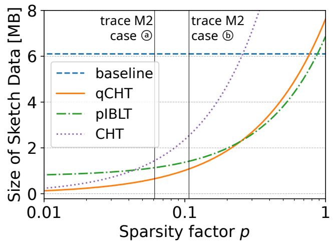
where the space occupied by the CHT *Flow Map* is the same as the baseline, $n_u \cdot s_c$ is the space occupied to store the counters of a single sketch, and $n_s \cdot m$ is the size of the bitmap B. Similarly to the qCHT version, the advantages of using a pIBLT *Sketch Data* in place of the baseline diminishes with increasing values of p, i.e., with an increasing number of non-zero counters. In Section 3.4, we detail the trade-offs of using SPADA in place of the baseline, comparing the two proposed solutions based on qCHT and pIBLT in multiple settings providing insights on memory sizing practices.

Example. We now highlight the SPADA-pIBLT memory footprint referring to the same use case ⓑ as in Section 3.3.1. As the *Flow Map* is the same, its required memory remains 1.5MB. Concerning the *Sketch Data*, we need 800KB for the bitmap, that is, one bit for each possible `<SketchID, index>` pair, while the 4 tables require $n_s \cdot n_u \cdot s_c = 960$KB. Therefore the total memory is around 3.26MB, which is similar to the case of the qCHT, i.e., 60% smaller compared to the baseline.

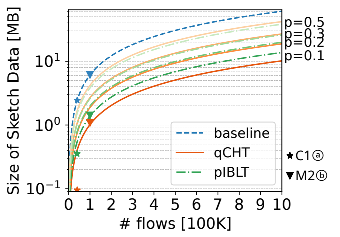
Limitations. We remark that pIBLT supports only additive counters e.g., used in CMS, DDSketch, hence it is not suitable for HLL since it requires reading values at update time. Furthermore, with a qCHT *Sketch Data*, increasing m, i.e., the number of buckets of each sketch, has negligible impact on the memory size. This is not true for the pIBLT *Sketch Data*, as the size of the bitmap storing the non-zero `<SketchID, index>` pairs grows linearly with the number of buckets.

3.4 Analysis of memory sizing

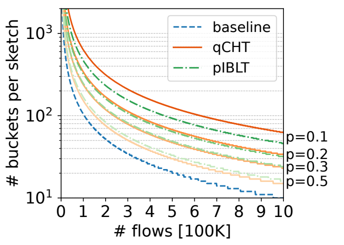
To benefit from SPADA, it is paramount to accurately size the *Sketch Data* based on a worst-case sparsity factor p. Despite allowing flexible bucket assignments across different flows, our solution requires fixing the total number of non-zero buckets in practice. If the average p of the data is higher than the expected one, the *Sketch Data* reaches its critical load factor, making new insertions impossible. In this section, we analyze the relationships between sparsity, the number of flows to be monitored, and the required memory size. This analysis breaks down the advantages that SPADA can bring in practice and provides an insight into proper system configuration based on the available memory and the number of flows to monitor. The baseline *Sketch Data* memory footprint depends on *(i)* the number of flows n_s to monitor, i.e., number of sketches in most use cases, and *(ii)* the desired accuracy, i.e., buckets per sketch m. SPADA *Sketch Data* memory footprint instead, depends on a worst-case sparsity assumption, i.e., average ratio of non-zero counters per sketch p.



(a) Memory vs data sparsity factor (n_s=100K, m=64).



(b) Memory vs number of flows for various values of p (m=64).



(c) Max number of buckets per sketch m in 10 MB of *Sketch Data*.

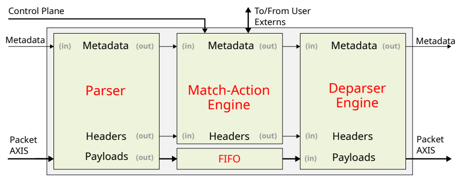
Fig. 5. Relationship between data sparsity p and memory requirements of SPADA data structures.

Figure 5a depicts the *Sketch Data* memory footprint for the three implementations varying p, assuming $n_s = 100K$ sketches with $m = 64$ buckets each, and a bucket size $s_c = 8$ bits. The plot also reports the sparsity factors for two reference use cases, i.e., ⓐ super spreader and ⓑ IAT quantiles estimation, extracted from the worst-case trace used in our evaluation (cf. Table 4 for further details). We remark that the pIBLT bitmap introduces a constant cost that is relatively high when p is small, i.e., data is highly sparse; this disadvantage diminishes for higher values of p, as the pIBLT does not use extra memory to store the key of each bucket like the qCHT. For $p > 0.25$ the disadvantage of the bitmap disappears and the qCHT requires slightly more memory. In general, both solutions greatly outperform the baseline even for very conservative sparsity assumptions (up to $p \approx 0.75$). Finally, we remark that the vanilla CHT is not a good solution for the *Sketch Data* implementation in most cases, as it occupies more memory than the baseline when $p \approx 0.3$.

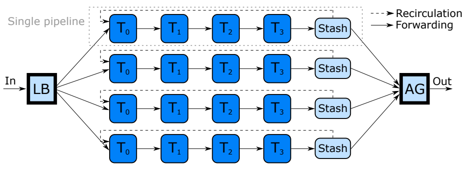
Figure 5b contrasts the required *Sketch Data* memory with the number of monitored flows providing an insight on how to properly set up SPADA based on the system requirements. For instance, assuming that 10 MB are pre-allocated for the *Sketch Data*, a baseline implementation would allow to monitor $\approx 150K$ flows, while SPADA can monitor $\approx 400K$ assuming a worst-case $p = 0.3$. Finally, Figure 5c shows the trade-off between the number of monitored flows and desired monitoring accuracy, i.e., number of buckets per sketch m, for 10 MB of available memory. Based on the number of desired flows, one can adjust the expected sparsity factor p to reach the desired monitoring precision: with 500K flows, setting $p = 0.3$ only grants $m \approx 50$ counters per sketch, while $p = 0.2$ brings m to ≈ 70. We highlight that properly selecting p, hence sizing the system accordingly, requires additional considerations based on historical knowledge, traffic predictions, and the purpose of the monitoring system itself. In extreme cases when worst-case p exceeds expectations, SPADA can stop adding new counters or start allowing sharing them across different flows at the cost of losing accuracy with respect to the original per-flow sketch, as it would happen for non-SPADA structures with more memory. Nevertheless, we remark that, as highlighted in Figure 5a, the memory trade-off provided by SPADA is better when compared to a non-sparse implementation, even for very conservative sparsity assumptions.

4 IMPLEMENTATION

We implement SPADA using the Xilinx Vitis Networking P4 [4]. The framework translates P4 code into Intellectual Property blocks for AMD-Xilinx FPGAs. Such blocks can then be integrated into a wrapper, as the AMD Xilinx OpenNIC Shell [2] or the NetFPGA-PLUS [43], which provide basic networking functionalities, and deployed in an FPGA-based SmartNIC. Our implementation exploits the NetFPGA-PLUS reference NIC using the Xilinx Alveo U280 (2×100Gbps QSFP, 8GB DDR, 41MB SRAM, 1M Look-up Tables, and 2M Flip-Flops) as target board. All the processing pipelines are clocked at 180 MHz, which is the standard frequency for the NetFPGA-PLUS datapath.



(a) Vitis Networking P4 Architecture.



(b) P4 pipeline for CHT and qCHT.

Fig. 6. Details of P4 FPGA programming framework and CHT and qCHT implementation pipelines.

In the following, after a brief Vitis Networking P4 architecture overview, we detail basic SPADA data structures implementations, and two full-fledged SPADA monitoring pipelines synthesized for the Xilinx Alveo U280 FPGA contrasting them with baseline designs.

4.1 The Vitis Networking P4 architecture

The Vitis Networking P4 architecture (cf. Figure 6a) is composed of three main blocks: *(i)* a parser, *(ii)* a match action engine, and *(iii)* a deparser. Extern blocks are directly implemented in Verilog HDL and can be added to the P4 pipeline using a configurable interface. In our prototype, we use externs to deploy data plane accessible registers, similar to those in Banzai [54] or Tofino [3].

4.2 SPADA building blocks

SPADA monitoring pipelines can be built using two of the following building blocks: MAT, (q)CHT, and pIBLT. Here, we detail their implementation in Vitis Networking P4 while the required hardware resources with the Xilinx Alveo U280 FPGA-based Smart-NIC as target are deferred to Appendix C.

Match Action Table. The MAT is the basic programming unit offered by the P4 language abstraction. Each MAT is defined by: what data to match on, a list of possible actions, and an optional number of properties, e.g., size, default action, etc. In SPADA, the MAT can be used to implement the *Flow Map* when `FlowToSketch` mappings are statically pushed from the control plane. Its implementation in the Vitis P4 framework is directly supported by the compiler and requires specifying the matching key, the algorithm (exact or ternary, the action), and the table length. The resulting block offers an AXI4 interface that can be wired to the control plane to populate the MAT. We implement two different MATs: one matching on source IP for cardinality estimation (use case ⓐ), and one matching on TCP/IP 5-tuple for IAT quantile estimation (use case ⓑ). Both MAT actions consist of writing in the packet metadata an integer `SketchID`.

Cuckoo Hash Table with quotient. In SPADA the CHT, optionally with quotienting when needed, can be used for the *Flow Map* or the *Sketch Data*. Its P4 implementation requires d hash functions (implemented with externs in the Vitis P4 framework) to compute the table indexes. Unlike the MAT, implementing a CHT in P4 is challenging due to its non-constant insertion time. Indeed, when all designated CHT indexes for an item are occupied, one of them is randomly chosen, replaced, and reinserted elsewhere. This would require accessing the same memory multiple times in a pipeline, which is not allowed in P4. Hence, CHT insertion may force recirculating keys and values (as we are monitoring, packets can be forwarded normally), potentially impacting the processing rate of the forwarding pipeline. To mitigate this problem, we implement a CHT with four tables, i.e., $d = 4$, augmented with a small memory called *stash* [36] that enables fixed insertion time via lazy recirculation. The high-level architecture of the simple P4 pipeline is depicted in Figure 6b (top). At key insertion, the packet first traverses the different pipeline modules one by one, each implementing a single table T_r. If a free position is found, then the item is inserted[4] there;

[4]Our system is based on a per-epoch measurement approach in which elements are deleted only at the end of an epoch. Hence it is safe to insert a key in the first available slot without checking if the key is already present in a subsequent table.

Table 3. Prototypes resources utilization.

UC	Pipeline	**LUT** [K - %]	**LUTRAM** [K - %]	**FF** [K - %]	**BRAM** [# - %]
-	NetFPGA	125.6 - 9.6	17.1 - 2.8	201 - 7.7	279 - 13.8
ⓐ	HLL Basline	401.7 - 30.8	186.1 - 30.9	225.2 - 8.6	354 - 17.5
	SPADA-HLL (static, 4 datapaths)	206.9 - 15.8	67.3 - 11.1	329 - 12.61	358 - 17.7
ⓑ	DDSketch Baseline	388.7 - 30	187 - 31	237.5 - 9.1	390 - 19.3
	SPADA-DDSketch (static)	183.1 - 14	57.4 - 9.5	241.7 - 9.3	390 - 19.3
ⓒ	ES Baseline	387.2 - 29.7	193.8 - 32.2	227.3 - 8.7	282 - 13.9
	SPADA-ES	185.9 - 14.2	66.8 - 12	248.1 - 9.5	282 - 13.9

otherwise, the key is stored in the next free slot in the stash[5]. The recirculation process is triggered whenever a stash insertion makes it exceeds a predefined threshold. In this case, we randomly pick a table T_r and insert the evicted item into it, replacing any previously stored item. The replaced item then traverses the pipeline starting from block r, looking for an empty memory slot among its other designated positions in other tables. If an empty slot is found, the item is inserted there, and the recirculation process ends; otherwise, it is stored in the stash, triggering another recirculation.

This solution has the drawback of only recirculating one item at a time. To overcome this limitation, we stack up to four (smaller) CHT that operate on parallel datapaths fed by a hash-based load balancer. Each datapath features its own stash, and recirculation is triggered on all datapaths at the same time. We call this mechanism "batch recirculation" since a single recirculation step moves more than one item at a time. Note that different recirculation policies might be implemented, e.g., recirculate when one (aggressive) or all (conservative) stashes reach the threshold. Finally, an output aggregator is responsible for recomposing a single output stream of packets.

Perfect IBLT. Due to its simple insertion routine, implementing the pIBLT in P4 does not present particular challenges. Similarly to the CHT, the P4 code employs four externs for hashes used to identify the indexes within each table, and another one for both the bitmap and the buckets.

4.3 SPADA-enabled monitoring pipelines

In this section, we use the building blocks described above to implement pipelines for use cases ⓐ, ⓑ, and ⓒ and wrap them in the NetFPGA-Plus architecture [43]. The pipelines feature three P4 blocks: *(i)* a MAT or CHT for the *Flow Map*, *(ii)* a middle block that computes the sketch index, and *(iii)* a pIBLT or qCHT for storing the sparse *Sketch Data*. Table 3 contrasts the basic NetFPGA-Plus reference NIC hardware requirements to the ones for HLL, DDSketch, and ElasticSketch (with $m = 32$ and $s_c = 16$ bits) wrapped in this reference NIC. It also details resource usage for static and dynamic versions as well as monitoring system baselines described in Section 3.2. In general, we remark that the additional hardware requirements to deploy our monitoring data plane on top of the NetFPGA-Plus reference NIC architecture are marginal, i.e., $\approx +10\%$. Additionally, we note that baseline implementations require $\approx 2\times$ hardware resources with respect to their SPADA counterparts and would be able to accurately monitor fewer flows in the data plane.

HLL. We implement the HLL sketching algorithm for super spreader detection (use case ⓐ) in two steps within the second and third P4 blocks above. The middle block relies on a single extern to compute the hash of the destination IP and uses the output to *(i)* identify the index of the HLL sketch bucket, and *(ii)* compute the value to be stored in the bucket, i.e., the number of consecutive leading zeroes. The bucket index is concatenated to the `SketchID` to build the key `<SketchID, index>`, and both key and value are attached to the packet as metadata. Finally, the block implementing the *Sketch Data* is responsible for checking whether the newly computed value is greater than the one currently stored, and replacing it if so. For HLL we use smaller $s_c = 8$ bits counters.

[5]The same key might be stored twice in the stash, hence possibly recirculated in two separate tables. However, at the end of the measurement epoch, any duplicated value is reconciled by the control plane resolving potential conflicts. We expect this phenomenon to be limited since CHTs are reset at each epoch and packets of the same flow may fall in separate counters.

DDSketch. For IAT quantile estimation (use case ⓑ), the *Flow Map* stores the `SketchID` and the timestamp of the last packet for each flow. The latter is used to compute the IAT upon reception of a new packet. The IAT value is attached to the packet as metadata and used by the middle block to identify the relevant DDSketch bin `index` through a small MAT table. In particular, we select the longest prefix matching between the current IAT value and precomputed delimiters of DDSketch buckets. The key is built using the `<SketchID, index>` pair and written in the packet metadata. The last block is responsible for incrementing the counter stored in the corresponding bucket.

ElasticSketch. For flow size estimation with ElasticSketch [63] (use case ⓒ), the first P4 block implements the *Flow Map*: four hash tables constituting the heavy part that stores elephant flows. In particular, every flow key is associated with an exact packet counter and positive and negative votes to implement the ostracism mechanism. Flows identified as mice by the ostracism are dynamically evicted. Mice flows are stored in a separate *Sketch Data*: a single-row Count-Min Sketch (CMS). The second P4 block computes a hash on the flow key for mice flows, thus identifying the corresponding bucket within the CMS, and attaches the `index` as user-defined metadata. Finally, the last block is responsible for incrementing the relevant CMS bucket. Note that, as there is only one sketch, in this case, the key for the *Sketch Data* is composed of the sole bucket `index`.

5 EVALUATION

In this section, we first evaluate SPADA on reference use cases using CAIDA 2016 [13] and MAWI 2019 datasets [41] and a custom software simulator (available at https://github.com/cpt-harlock/SPADA). We compare SPADA against state-of-the-art approaches in terms of memory requirements and accuracy (when affected). Second, we evaluate the (q)CHT recirculation overhead and discuss its feasibility in real systems. Finally, we also evaluate latency and throughput of our FPGA implementation, both via real prototype experiments and with accurate Verilog simulations.

5.1 Use cases evaluation

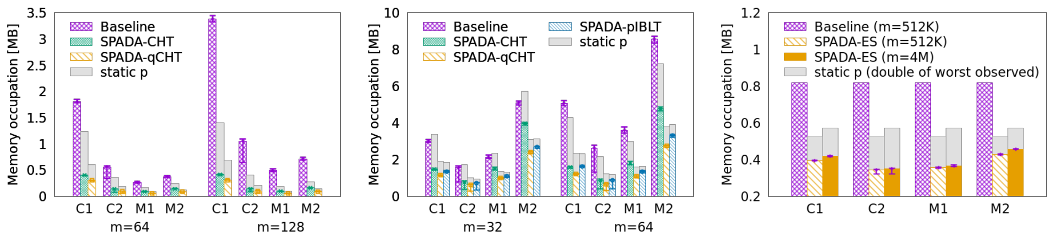
To evaluate the efficiency of SPADA in a realistic scenario, we feed the simulator with 1-hour CAIDA traces and 5-minute MAWI traces. In particular, we consider: (C1) 21/01/2016 13:00 – 14:00, (C2) 17/03/2016 14:00 – 15:00, (M1) 09/04/2019 22:15 – 22:20, and (M2) 09/04/2019 13:15 – 13:20. For CAIDA traces, we consider 1-second epochs, while for MAWI traces we use 1-minute epochs since they feature fewer packets. We consider TCP traffic only from all traces. For ⓐ super spreader detection, we use source IPs as flow keys and $m = 64$ or $m = 128$ counters for the HLL sketches (error 13% and 9% respectively). For ⓑ IAT quantile estimation, we use 5-tuples as flow keys and DDSketches with $m = 32$ or $m = 64$ counters each (relative error $\alpha = 0.28$ and $\alpha = 0.14$ respectively). Finally, for ⓒ flow size estimation we build a baseline ElasticSketch (ES) of 818KB, divided into a heavy part of 318KB and a light part of 500KB (i.e., a CMS composed of $m = 512$K 8-bit counters). SPADA-ES allocates the same amount of memory to the heavy part (*Flow Map*), uses a qCHT to store the light part (*Sketch Data*), and is tested using ES open source simulator [22]. Additionally, we evaluate a more accurate SPADA-ES "virtually" increasing the CMS size ($m = 4$M).

Table 4. Data structure parameters for each use case and their sparsity values for traces C1, C2 and M1, M2.

Use	Parameters						Sparsity factor			
case	s_k	m	$n_{s\,\mathrm{C1}}$	$n_{s\,\mathrm{C2}}$	$n_{s\,\mathrm{M1}}$	$n_{s\,\mathrm{M2}}$	p_{C1}	p_{C2}	p_{M1}	p_{M2}
ⓐ	32	64	36K	11K	5.4K	7.6K	0.020	0.028	0.051	0.061
	32	128	36K	11K	5.4K	7.6K	0.010	0.015	0.028	0.034
ⓑ	104	32	59K	31K	41K	100K	0.068	0.078	0.139	0.162
	104	64	59K	31K	41K	100K	0.040	0.048	0.092	0.107
ⓒ	104	0.5M	59K	31K	41K	100K	0.085	0.025	0.039	0.124
	104	4M	59K	31K	41K	100K	0.011	0.003	0.005	0.016

Table 5. ES and SPADA-ES accuracy for traces C1, C2, and M1, M2.

Metric		m	C1	C2	M1	M2
ARE	ES	0.5M	0.10	0.01	0.05	0.30
	SPADA-ES	4M	0.01	2E-3	6E-3	0.03
AAE	ES	0.5M	0.16	0.02	0.12	0.97
	SPADA-ES	4M	0.02	3E-3	5E-3	0.12
ER	ES	0.5M	0.93	0.98	0.97	0.89
	SPADA-ES	4M	0.99	0.99	0.99	0.98



(a) Super spreader detection. (b) IAT quantile estimation. (c) Flow size estimation.

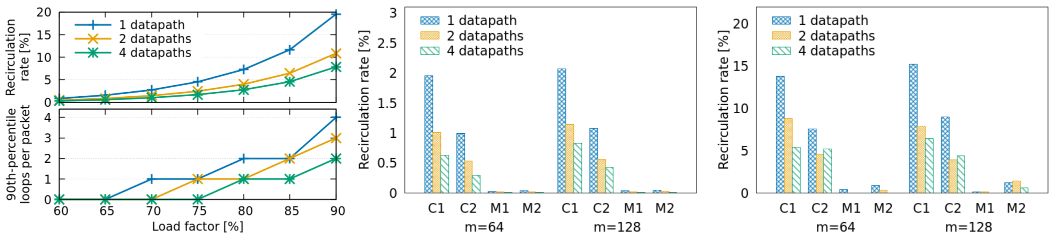
Fig. 7. Memory footprint for the reference use cases comparing SPADA with SOTA baselines.

In our simulations both *Flow Map* and *Sketch Data* are sized to keep the load factor below 90%, and all (q)CHTs feature a stash with 16 additional buckets. Table 4 lists the main SPADA parameters for uses cases ⓐ, ⓑ and ⓒ, also reporting the sparsity recorded for the various traces, expressed as the average ratio of non-zero sketch counters p_{Ci}, ranging from 0.003 to 0.162.

Memory occupancy. Figures 7a, 7b, and 7c contrast the memory occupied by the baseline and SPADA monitoring system. Colored histograms average values across the epochs, with error bars for min and max values when available. Overall, SPADA reduces memory occupancy from 2× for ⓑ (DDSketch), 2.5× for ⓒ (ElasticSketch), to 11× for ⓐ (HLL). More precisely, the sparser the baseline sketches, the higher the memory saving. As analyzed in Section 3.4, SPADA-pIBLT requires more memory than SPADA-qCHT due to the additional bitmap. However, the memory saving with respect to the baseline is still significant with the advantage of avoiding recirculation in the *Sketch Data* component. Figures 7a, 7b, and 7c also show over-dimensioned SPADA memory footprint using a conservative fixed value of p, grey bars (we set p to the double of the highest value observed in each experiment — cf. Table 4). We remark that with qCHT and pIBLT *Sketch Data*, memory reduction is still sizeable despite conservative settings. Conversely, as highlighted in Section 3.4, for the CHT the per-counter overhead is too high, and hence its memory occupancy is in practice similar, sometimes even higher, than the baseline (for $p \approx 0.3$).

Processing time for pIBLT lookup. At the end of the measurement epoch, the control plane dumps the pIBLT content and solves the linear system associated with it. Note that, even though this computation does not affect the data plane, the system needs to be solved before the end of the next epoch to avoid overloading the resolution system. Since the resolution time of a linear system is superlinear with respect to the number of equations, we use a first-level hash function to split the rows of the pIBLT into a set of smaller disjoint linear systems. This reduces the computation time and enables the use of multiple cores. In particular, with two threads of an Intel i7-10700K CPU clocked at 3.80 GHz, the linear system of the pIBLT is solved in less than one second (960 ms). We achieved this result by splitting the 128K rows of the linear system into 128 independent linear systems of 1K rows each. The time needed for solving each of these systems is approx. 15 ms.

Flow size estimation accuracy. While for use cases ⓐ (super spreader) and ⓑ (IAT quantiles), the estimation accuracy is not impacted, use case ⓒ (flow size estimation) requires additional consideration. As shown in Figure 7c, with SPADA we can increase the CMS size m from 512K to 4M with minimal impact on memory. We now compare the accuracy of the baseline ElasticSketch (ES) with our enhanced SPADA-ES. Table 5 reports Average Relative Error (ARE), Average Absolute Error (AAE), and the fraction of flows for which the sketch provides the exact result (ER). We note that, despite the much smaller memory footprint, SPADA-ES always provides better accuracy than the baseline ES. In particular, SPADA-ES achieves one order of magnitude better AAE and ARE than ES. Furthermore, SPADA-ES provides the exact count for more than 98% of flows in *all* traces, whereas the standard ES tops at 98% only in C2 which is the trace featuring fewer flows.



(a) w/ uniform random keys. (b) Overall — w/ real traffic. (c) Worst case — w/ real traffic.

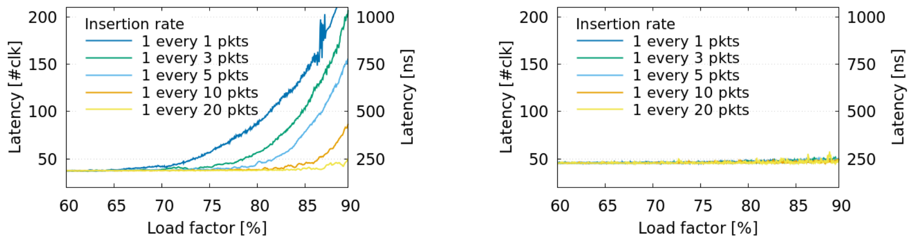
Fig. 8. **(a)** CHT recirculation rate (top) and 90th percentile loop length (bottom) vs. load factor with random keys. **(b)** Overall and **(c)** worst-case recirculation rate on real traces for super spreader detection.

5.2 Cuckoo Hash Table recirculation overhead

In this section, we evaluate the computational overhead of SPADA-qCHT due to CHT recirculations. For simplicity, we assume asynchronous recirculation loops in our simulations, hence we never experience insertion failures due to stashes overload. In this set of tests, we fix the number of CHT slots to 2^{16}, with 16 additional slots for the stash, equally split among the available datapaths and trigger recirculation when all stashes reach a threshold of 50%, unless otherwise specified.

Synthetic keys. We first run a stress test that consists of inserting random keys in the CHT until a target load factor is reached. Figure 8a (top) shows the average recirculation rate when starting from an empty CHT. We observe that when using a single datapath, the recirculation rate to reach a load factor of 90% is 20%. This drastically improves when using four datapaths thanks to batch recirculation. For what concerns a more aggressive policy, i.e., recirculate when at least one stash reaches the threshold, when using 4 datapaths we report up to 26% more recirculations w.r.t. the conservative policy (not shown for lack of space). This is due to recirculations triggered when some stashes are still empty, thus wasting available datapaths. Figure 8a (bottom) provides the 90th percentiles loop length when starting from non-empty CHT, i.e., we count recirculations that occur when the CHT is at the target load in order to evaluate the recirculation overhead in the worst case. We observe that, at 90% load, most packets trigger less than 4 recirculations (90th percentile) when using a single datapath. Recirculation is halved when using 4 datapaths, so that 90% of the packets at 90% load trigger at most 2 recirculations. Not shown for lack of space, average recirculation is much lower: at 90% load, we measure 1.6 (resp. 0.6) loops per packet with 1 (resp. 4) datapath(s), meaning that every insertion triggers less than one recirculation on average. Thus, in most cases, the recirculation overhead is negligible and does not affect the system performance.

Real traces. We then evaluate SPADA-qCHT recirculation overhead on real traffic. Unlike the previous analysis, we remark that on real traffic a large fraction of packets only update already occupied `<SketchID, index>` pairs, thus not triggering any recirculation. Figure 8b reports the recirculation rate with respect to the overall number of packets. This set of results refers to the HLL use case ⓐ with $m = 64$ and $m = 128$. Simulations are performed using 1, 2, and 4 datapaths and the CHT is dimensioned to reach 90% load factor. We observe an overall recirculation rate below 2% for a single datapath and around 0.5% when using 4 datapaths with CAIDA traces. Recirculation rate drops drastically with MAWI traces as they feature much fewer flows, i.e., sIP in the HLL use case ⓐ. Our findings assess the feasibility of the recirculation approach in real scenarios as the extra bandwidth required to recirculate is negligible. Figure 8c reports worst-case values, which correspond to a fully loaded table (around 90%). In this case, the recirculation rate is much higher but stays below 5% when 4 datapaths are used. Other data plane applications that exploit recirculation exhibit similar (Lucid [55], average 2%) or higher (Dart [51], worst-case 16%) recirculation rate.



(a) Single datapath. (b) 2 datapaths.

Fig. 9. FPGA insertion latency vs. load factor at different insertion rates (average over 100 runs).

5.3 FPGA prototype evaluation

Finally, we evaluate throughput and latency of our SPADA-qCHT FPGA implementation through real FPGA experiments and clock cycle-accurate Verilog simulations. The latter enables better visibility of latency degradation since it usually corresponds to a few clock cycles over the 960 ns of the plain Open NIC Shell [2]. Note that SPADA is a passive monitoring system, meaning that monitoring logic does not delay incoming packets or influence forwarding decisions. Thus, any additional latency introduced by SPADA is due to *(i)* the arbiter that multiplexes input packets and *(ii)* processing of recirculated packets.

Figure 9 shows CHT per-packet latency of the SPADA-qCHT Verilog simulation at maximum throughput, varying the insertion rate, that is, how many packets trigger the insertion of a new key. Note that the latency in the figure does not take into account the latency overhead of the Xilinx Open NIC Shell used to host the system. With a single datapath, latency is $\approx$40 clock cycles (200 ns) for a low CHT load factor, i.e., $<$65%. At 90% load, latency increases to more than 200 clock cycles (1000 ns) in the worst case that every packet triggers a new CHT insertion. However, the latency increase is much more limited at lower insertion rates, and becomes negligible when performing 1 insertion every 20 packets: note that in CAIDA and MAWI traces, worst case insertion rate is 1 every 38 packets. Using 2 datapaths (Figure 9b) we do not observe latency increase even at very high load factors and insertion rates (4 datapaths not shown as results are similar).

Concerning throughput, our simulations show no throughput degradation and no-failure insertions up to 85% load factor for a single datapath, and up to 89.0% for 2 datapaths, and 4 datapaths also in the worst case of 1 insertion at every packet. This is mainly because we clock the system at 180 MHz, and the maximum throughput with minimum size packets (64B) at 100 Gbps is 144 Mpps thereby leaving 36 Mpps for packet recirculations. To confirm the Verilog simulations we tested our FPGA prototype with minimum-sized packets at maximum throughput. The experiments confirm no throughput degradation, while the measured latency is 1160ns, that is 200 ns latency due to SPADA-qCHT plus 960 ns overhead given by the Xilinx Open NIC Shell.

6 RELATED WORK AND DISCUSSION

Several Flow-to-ID mapping techniques have been proposed in the literature [8, 47, 67]. Since SPADA main goal is to compact the *Sketch Data*, we do not directly review them but we remark that such techniques could be used in SPADA to further reduce the memory footprint.

In the context of cardinality estimation, HLL [24] and BeauCoup [15] are popular and efficient methodologies proposed in the literature. HLL enhancements [33, 61] try to exploit sparsity to reduce the data structure memory footprint. Unlike SPADA however, these approaches employ a counter-sharing mechanism that affects the measurement accuracy. BeauCoup instead, performs distinct counting through "coupons" collection in bit-vectors whose size trades off memory occupancy and accuracy e.g., for $m=256$ bits for each vector. BeauCoup error is comparable with HLL using the same memory while $m = 32$ increases the error by $\approx 3\times$ for $\approx 70\%$ less memory. Unlike BeauCoup, SPADA provides significant memory saving without sacrificing accuracy. It is worth mentioning that SPADA can also be applied to BeauCoup, as the bit-vectors are affected by sparsity ($p \approx 0.1$). Finally, SPADA can reduce the size of a series of other data structures in the context of cardinality estimation, e.g., PCSA [25], KVM [7], and Fast-AGMS [17]. Similar considerations apply to the quantile estimation use case. In particular, Circllhist [29] and KLL [35] feature sparse arrays similar to DDSketch and can be implemented using SPADA representations. Other approaches rely on a compact data representation by restricting the data collection to a subset of more relevant flows. For instance, SQUAD [52] combines the problem of quantile estimation with the one of heavy hitter detection, hence dynamically restricting quantile estimation only to the most frequent flows. Instead, SPADA does not need to rely on a prediction mechanism and significantly reduces the memory footprint without the drawback of restricting monitoring to a few flows. The key difference between SPADA and the aforementioned approaches is that it relies on *sparse data representation* to mitigate memory footprint. This leads to a generic technique that can be applied to a variety of other use cases as summarized in Table 1.

Existing sparse representations are typically restricted to static data structures and include well-known techniques such as Compressed Sparse Row, Coordinate Format, etc. Dynamic sparse representations such as STINGER [21], AIM [60], and HORNET [12] have been recently proposed in the context of graph and matrix representations for parallel processing units, e.g., GPU. However, such solutions are deployed on hardware where memory access is not constrained, unlike programmable switches. Finally, sparse representation directly relates to Compressive Sensing [26]. In the context of sketching, NZE [31] uses Compressive Sensing to design specific sketches that can approximately reconstruct the desired measurement, e.g., flow size estimation. Although NZE goal is similar to SPADA, we note that the reconstruction complexity in our case is negligible, whereas for NZE it exponentially increases with the number of flows.

Finally, while in this paper we focus on an FPGA-based implementation of SPADA, here we briefly discuss its applicability to other systems. We argue that sparse representation memory savings in CPU-based servers are limited (due to the amount of DRAM in such systems) but might help reduce cache misses. The reduced memory footprint provided by SPADA is beneficial in systems with plain memory hierarchy or stringent latency requirements, e.g., DRAM/HBM memories are not allowed. SmartNICs and P4 programmable ASIC switches can benefit from SPADA to better exploit their limited SRAM memory. The main issue of porting SPADA is *Flow Map* and *Sketch Data* insertion. This can be realized in SmartNICs P4 targets that support the `add_on_miss` feature proposed in the P4 Portable NIC Architecture [1], in programmable switches such as the NVIDIA Spectrum series that provides stateful tables with data plane flow insertions and removals [57, 62] or exploiting the stash with the batch recirculation mechanism proposed in this paper.

7 CONCLUSIONS

This paper presented SPADA, a Sparse Approximate Data Structure representation. SPADA is a method to reduce the data plane memory occupancy of per-flow monitoring systems without affecting accuracy. SPADA exploits the observation that, due to the skewed nature of network traffic, only a handful of sketch counters are used for most flows. This leads to heavily underutilized data structures in a number of monitoring use cases. SPADA has been designed to efficiently represent such sparse data by only storing non-zero sketch buckets and relies on (q)CHT and on a novel data structure pIBLT. We implemented SPADA on an FPGA-based SmartNIC using P4 and performed extensive simulations and prototype experiments on real traces and synthetic workloads for three popular monitoring use cases, achieving a memory reduction between $2\times$ and $11\times$ while maintaining the same accuracy and introducing limited computational overhead.

REFERENCES

[1] 2021. P4 Portable NIC Architecture (PNA) version 0.5. https://p4.org/p4-spec/docs/PNA.html.

[2] 2023. AMD OpenNIC Project. https://github.com/Xilinx/open-nic.

[3] Anurag Agrawal and Changhoon Kim. 2020. Intel Tofino2 – A 12.9Tbps P4-Programmable Ethernet Switch. In *2020 IEEE Hot Chips 32 Symposium (HCS)*. 1–32. https://doi.org/10.1109/HCS49909.2020.9220636

[4] AMD. 2023. *Xilinx Vitis Networking P4, https://www.xilinx.com/products/intellectual-property/ef-di-vitisnetp4.html.* AMD Inc.

[5] Arvind Arasu and Gurmeet Singh Manku. 2004. Approximate Counts and Quantiles over Sliding Windows. In *Proceedings of the Twenty-Third ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems* (Paris, France) *(PODS '04)*. Association for Computing Machinery, New York, NY, USA, 286–296. https://doi.org/10.1145/1055558.1055598

[6] Behnaz Arzani, Selim Ciraci, Luiz Chamon, Yibo Zhu, Hongqiang (Harry) Liu, Jitu Padhye, Boon Thau Loo, and Geoff Outhred. 2018. 007: Democratically Finding the Cause of Packet Drops. In *15th USENIX Symposium on Networked Systems Design and Implementation (NSDI 18)*. USENIX Association, Renton, WA, 419–435. https://www.usenix.org/conference/nsdi18/presentation/arzani

[7] Ziv Bar-Yossef, T. S. Jayram, Ravi Kumar, D. Sivakumar, and Luca Trevisan. 2002. Counting Distinct Elements in a Data Stream. In *Randomization and Approximation Techniques in Computer Science (RANDOM 2002)*, José D. P. Rolim and Salil Vadhan (Eds.). Springer, Berlin, Heidelberg, 1–10.

[8] Tom Barbette, Chen Tang, Haoran Yao, Dejan Kostić, Gerald Q. Maguire Jr., Panagiotis Papadimitratos, and Marco Chiesa. 2020. A High-Speed Load-Balancer Design with Guaranteed Per-Connection-Consistency. In *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI 20)*. USENIX Association, Santa Clara, CA, 667–683. https://www.usenix.org/conference/nsdi20/presentation/barbette

[9] Ran Ben Basat, Xiaoqi Chen, Gil Einziger, and Ori Rottenstreich. 2020. Designing heavy-hitter detection algorithms for programmable switches. *IEEE/ACM Transactions on Networking* 28, 3 (2020), 1172–1185.

[10] Ran Ben Basat, Sivaramakrishnan Ramanathan, Yuliang Li, Gianni Antichi, Minian Yu, and Michael Mitzenmacher. 2020. PINT: Probabilistic In-Band Network Telemetry. In *Proceedings of the Annual Conference of the ACM Special Interest Group on Data Communication on the Applications, Technologies, Architectures, and Protocols for Computer Communication* (Virtual Event, USA) *(SIGCOMM '20)*. Association for Computing Machinery, New York, NY, USA, 662–680. https://doi.org/10.1145/3387514.3405894

[11] Theophilus Benson, Ashok Anand, Aditya Akella, and Ming Zhang. 2011. MicroTE: Fine Grained Traffic Engineering for Data Centers. In *Proceedings of the Seventh COnference on Emerging Networking EXperiments and Technologies* (Tokyo, Japan) *(CoNEXT '11)*. Association for Computing Machinery, New York, NY, USA, Article 8, 12 pages. https://doi.org/10.1145/2079296.2079304

[12] Federico Busato, Oded Green, Nicola Bombieri, and David A. Bader. 2018. Hornet: An Efficient Data Structure for Dynamic Sparse Graphs and Matrices on GPUs. In *2018 IEEE High Performance extreme Computing Conference (HPEC)*. 1–7. https://doi.org/10.1109/HPEC.2018.8547541

[13] CAIDA 2019. The CAIDA Anonymized Internet Traces Dataset. https://www.caida.org/catalog/datasets/passive_dataset/.

[14] Haoxian Chen, Nate Foster, Jake Silverman, Michael Whittaker, Brandon Zhang, and Rene Zhang. 2016. Felix: Implementing Traffic Measurement on End Hosts Using Program Analysis. In *Proceedings of the Symposium on SDN Research* (Santa Clara, CA, USA) *(SOSR '16)*. Association for Computing Machinery, New York, NY, USA, Article 14, 12 pages. https://doi.org/10.1145/2890955.2890971

[15] Xiaoqi Chen, Shir Landau-Feibish, Mark Braverman, and Jennifer Rexford. 2020. BeauCoup: Answering Many Network Traffic Queries, One Memory Update at a Time. In *Proceedings of the Annual Conference of the ACM Special Interest Group on Data Communication on the Applications, Technologies, Architectures, and Protocols for Computer Communication* (Virtual Event, USA) *(SIGCOMM '20)*. Association for Computing Machinery, New York, NY, USA, 226–239. https://doi.org/10.1145/3387514.3405865

[16] Baek-Young Choi, Sue Moon, Rene Cruz, Zhi-Li Zhang, and Christophe Diot. 2007. Quantile sampling for practical delay monitoring in Internet backbone networks. *Computer Networks* 51, 10 (2007), 2701–2716.

[17] Graham Cormode and Minos Garofalakis. 2005. Sketching Streams through the Net: Distributed Approximate Query Tracking. In *Proceedings of the 31st International Conference on Very Large Data Bases* (Trondheim, Norway) *(VLDB '05)*. VLDB Endowment, 13–24.

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

[19] Martin Dietzfelbinger, Andreas Goerdt, Michael Mitzenmacher, Andrea Montanari, Rasmus Pagh, and Michael Rink. 2010. Tight Thresholds for Cuckoo Hashing via XORSAT. In *Automata, Languages and Programming*, Samson Abramsky, Cyril Gavoille, Claude Kirchner, Friedhelm Meyer auf der Heide, and Paul G. Spirakis (Eds.). Springer, Berlin, Heidelberg,

213–225.

[20] O. Dubois and J. Mandler. 2002. The 3-XORSAT threshold. In *The 43rd Annual IEEE Symposium on Foundations of Computer Science, 2002. Proceedings.* IEEE, 769–778. https://doi.org/10.1109/SFCS.2002.1182002

[21] David Ediger, Rob McColl, Jason Riedy, and David A. Bader. 2012. STINGER: High performance data structure for streaming graphs. In *2012 IEEE Conference on High Performance Extreme Computing*. IEEE, 1–5. https://doi.org/10.1109/HPEC.2012.6408680

[22] ES-code 2023. Elastc Sketch source code. https://github.com/BlockLiu/ElasticSketchCode.

[23] Cristian Estan and George Varghese. 2003. New Directions in Traffic Measurement and Accounting: Focusing on the Elephants, Ignoring the Mice. *ACM Trans. Comput. Syst.* 21, 3 (aug 2003), 270–313. https://doi.org/10.1145/859716.859719

[24] Philippe Flajolet, Éric Fusy, Olivier Gandouet, and Frédéric Meunier. 2007. HyperLogLog: the analysis of a near-optimal cardinality estimation algorithm. *AofA: Analysis of Algorithms* DMTCS Proceedings vol. AH, 2007 Conference on Analysis of Algorithms (AofA 07) (June 2007), 137–156. https://doi.org/10.46298/dmtcs.3545

[25] Philippe Flajolet and G Nigel Martin. 1985. Probabilistic counting algorithms for data base applications. *Journal of computer and system sciences* 31, 2 (1985), 182–209.

[26] Massimo Fornasier and Holger Rauhut. 2015. Compressive Sensing. *Handbook of mathematical methods in imaging* 1 (2015), 187–229.

[27] Michael T. Goodrich and Michael Mitzenmacher. 2011. Invertible bloom lookup tables. In *2011 49th Annual Allerton Conference on Communication, Control, and Computing (Allerton)*. IEEE, 792–799. https://doi.org/10.1109/Allerton.2011.6120248

[28] Hui Han, Zheng Yan, Xuyang Jing, and Witold Pedrycz. 2022. Applications of sketches in network traffic measurement: A survey. *Information Fusion* 82 (2022), 58–85.

[29] Heinrich Hartmann and Theo Schlossnagle. 2020. Circllhist – A Log-Linear Histogram Data Structure for IT Infrastructure Monitoring. https://doi.org/10.48550/ARXIV.2001.06561

[30] Chen-Yu Hsu, Piotr Indyk, Dina Katabi, and Ali Vakilian. 2019. Learning-Based Frequency Estimation Algorithms. In *International Conference on Learning Representations*. https://openreview.net/forum?id=r1lohoCqY7

[31] Qun Huang, Siyuan Sheng, Xiang Chen, Yungang Bao, Rui Zhang, Yanwei Xu, and Gong Zhang. 2021. Toward Nearly-Zero-Error Sketching via Compressive Sensing. In *18th USENIX Symposium on Networked Systems Design and Implementation (NSDI 21)*. USENIX Association, 1027–1044. https://www.usenix.org/conference/nsdi21/presentation/huang

[32] Nikita Ivkin, Zhuolong Yu, Vladimir Braverman, and Xin Jin. 2019. QPipe: Quantiles Sketch Fully in the Data Plane. In *Proceedings of the 15th International Conference on Emerging Networking Experiments And Technologies* (Orlando, Florida) *(CoNEXT '19)*. Association for Computing Machinery, New York, NY, USA, 285–291. https://doi.org/10.1145/3359989.3365433

[33] Peng Jia, Pinghui Wang, Yuchao Zhang, Xiangliang Zhang, Jing Tao, Jianwei Ding, Xiaohong Guan, and Don Towsley. 2020. Accurately Estimating User Cardinalities and Detecting Super Spreaders over Time. *IEEE Transactions on Knowledge and Data Engineering* 34, 1 (2020), 92–106.

[34] Zohar Karnin, Kevin Lang, and Edo Liberty. 2016. Optimal quantile approximation in streams. In *2016 ieee 57th annual symposium on foundations of computer science (focs)*. IEEE, 71–78.

[35] Zohar Karnin, Kevin Lang, and Edo Liberty. 2016. Optimal quantile approximation in streams. In *Annual Symposium on Foundations of Computer Science (FOCS)*. IEEE, 71–78.

[36] Adam Kirsch, Michael Mitzenmacher, and Udi Wieder. 2010. More Robust Hashing: Cuckoo Hashing with a Stash. *SIAM J. Comput.* 39, 4 (2010), 1543–1561. https://doi.org/10.1137/080728743 arXiv:https://doi.org/10.1137/080728743

[37] Donald Ervin Knuth. 1973. *The art of computer programming: sorting and searching.* 723–723 pages.

[38] Ashwin Lall, Vyas Sekar, Mitsunori Ogihara, Jun Xu, and Hui Zhang. 2006. Data streaming algorithms for estimating entropy of network traffic. *ACM SIGMETRICS Performance Evaluation Review* 34, 1 (2006), 145–156.

[39] Yuliang Li, Rui Miao, Changhoon Kim, and Minlan Yu. 2016. LossRadar: Fast Detection of Lost Packets in Data Center Networks. In *Proceedings of the 12th International on Conference on Emerging Networking EXperiments and Technologies* (Irvine, California, USA) *(CoNEXT '16)*. Association for Computing Machinery, New York, NY, USA, 481–495. https://doi.org/10.1145/2999572.2999609

[40] Charles Masson, Jee E. Rim, and Homin K. Lee. 2019. DDSketch: A Fast and Fully-Mergeable Quantile Sketch with Relative-Error Guarantees. *Proc. VLDB Endow.* 12, 12 (aug 2019), 2195–2205. https://doi.org/10.14778/3352063.3352135

[41] MAWI 2023. MAWI Working Group Traffic Archive. https://mawi.wide.ad.jp/mawi/.

[42] Andrea Monterubbiano, Jonatan Langlet, Gianni Walzer, Stefan Antichi, Pedro Reviriego, and Salvatore Pontarelli. 2023. Lightweight Acquisition and Ranging of Flows in the Data Plane. In *Proceedings of the ACM SIGMETRICS/IFIP PERFORMANCE Joint International Conference on Measurement and Modeling of Computer Systems, Vol. 7, No. 3, Article 44, December 2023 (SIGMETRICS '24)*. Association for Computing Machinery, New York, NY, USA. https://doi.org/10.1145/3626775

[43] NetFPGA publisher. 2023. *NetFPGA-PLUS, https://netfpga.org/NetFPGA-PLUS.html.*

[44] Rasmus Pagh and Flemming Friche Rodler. 2004. Cuckoo hashing. *Journal of Algorithms* 51, 2 (2004), 122–144. https://doi.org/10.1016/j.jalgor.2003.12.002

[45] Odysseas Papapetrou, Minos Garofalakis, and Antonios Deligiannakis. 2015. Sketching Distributed Sliding-Window Data Streams. *The VLDB Journal* 24, 3 (jun 2015), 345–368. https://doi.org/10.1007/s00778-015-0380-7

[46] Boris Pittel and Gregory B. Sorkin. 2016. The Satisfiability Threshold for k-XORSAT. *Combinatorics, Probability & Computing* 25, 2 (2016), 236–268. https://doi.org/10.1017/S0963548315000097

[47] Salvatore Pontarelli, Roberto Bifulco, Marco Bonola, Carmelo Cascone, Marco Spaziani, Valerio Bruschi, Davide Sanvito, Giuseppe Siracusano, Antonio Capone, Michio Honda, Felipe Huici, and Giuseppe Siracusano. 2019. FlowBlaze: Stateful Packet Processing in Hardware. In *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19)*. USENIX Association, Boston, MA, 531–548. https://www.usenix.org/conference/nsdi19/presentation/pontarelli

[48] Tewarson Reginald P. 1973. *Sparse Matrices.* Academic Press. https://books.google.de/books?id=I-FQAAAAMAAJ

[49] Mariano Scazzariello, Tommaso Caiazzi, Hamid Ghasemirahni, Tom Barbette, Dejan Kostić, and Marco Chiesa. 2023. A High-Speed Stateful Packet Processing Approach for Tbps Programmable Switches. In *20th USENIX Symposium on Networked Systems Design and Implementation (NSDI 23)*. USENIX Association, Boston, MA, 1237–1255. https://www.usenix.org/conference/nsdi23/presentation/scazzariello

[50] Brandon Schlinker, Italo Cunha, Yi-Ching Chiu, Srikanth Sundaresan, and Ethan Katz-Bassett. 2019. Internet Performance from Facebook's Edge. In *Proceedings of the Internet Measurement Conference (IMC '19)*. Association for Computing Machinery, New York, NY, USA, 179–194. https://doi.org/10.1145/3355369.3355567

[51] Satadal Sengupta, Hyojoon Kim, and Jennifer Rexford. 2022. Continuous In-Network Round-Trip Time Monitoring. In *Proceedings of the ACM SIGCOMM 2022 Conference* (Amsterdam, Netherlands) *(SIGCOMM '22)*. Association for Computing Machinery, New York, NY, USA, 473–485. https://doi.org/10.1145/3544216.3544222

[52] Rana Shahout, Roy Friedman, and Ran Ben Basat. 2023. Together is Better: Heavy Hitters Quantile Estimation. *Proceedings of the ACM on Management of Data* 1, 1 (2023), 1–25.

[53] Siyuan Sheng, Qun Huang, Sa Wang, and Yungang Bao. 2021. PR-Sketch: Monitoring per-Key Aggregation of Streaming Data with Nearly Full Accuracy. *Proc. VLDB Endow.* 14, 10 (jun 2021), 1783–1796. https://doi.org/10.14778/3467861.3467868

[54] Anirudh Sivaraman, Alvin Cheung, Mihai Budiu, Changhoon Kim, Mohammad Alizadeh, Hari Balakrishnan, George Varghese, Nick McKeown, and Steve Licking. 2016. Packet Transactions: High-Level Programming for Line-Rate Switches. In *Proceedings of the 2016 ACM SIGCOMM Conference* (Florianopolis, Brazil) *(SIGCOMM '16)*. Association for Computing Machinery, New York, NY, USA, 15–28. https://doi.org/10.1145/2934872.2934900

[55] John Sonchack, Devon Loehr, Jennifer Rexford, and David Walker. 2021. Lucid: A Language for Control in the Data Plane. In *Proceedings of the 2021 ACM SIGCOMM 2021 Conference* (Virtual Event, USA) *(SIGCOMM '21)*. Association for Computing Machinery, New York, NY, USA, 731–747. https://doi.org/10.1145/3452296.3472903

[56] Cha Hwan Song, Pravein Govindan Kannan, Bryan Kian Hsiang Low, and Mun Choon Chan. 2020. FCM-Sketch: Generic Network Measurements with Data Plane Support. In *Proceedings of the 16th International Conference on Emerging Networking EXperiments and Technologies* (Barcelona, Spain) *(CoNEXT '20)*. Association for Computing Machinery, New York, NY, USA, 78–92. https://doi.org/10.1145/3386367.3432729

[57] Angelo Tulumello, Marco Bonola, Salvatore Pontarelli, Matty Kadosh, and Y Piasetzki. 2021. Extending p4 to realize a scalable flow caching mechanism. https://opennetworking.org/wp-content/uploads/2021/05/2021-P4-WS-Angelo-Tulumello-Slides.pdf.

[58] Stefan Walzer. 2021. Peeling Close to the Orientability Threshold: Spatial Coupling in Hashing-Based Data Structures. In *Proceedings of the Thirty-Second Annual ACM-SIAM Symposium on Discrete Algorithms* (Virtual Event, Virginia) *(SODA '21)*. Society for Industrial and Applied Mathematics, USA, 2194–2211.

[59] Haibo Wang, Chaoyi Ma, Olufemi O Odegbile, Shigang Chen, and Jih-Kwon Peir. 2021. Randomized error removal for online spread estimation in data streaming. *Proceedings of the VLDB Endowment* 14, 6 (2021), 1040–1052.

[60] Martin Winter, Rhaleb Zayer, and Markus Steinberger. 2017. Autonomous, independent management of dynamic graphs on GPUs. In *2017 IEEE High Performance Extreme Computing Conference (HPEC)*. 1–7. https://doi.org/10.1109/HPEC.2017.8091058

[61] Qingjun Xiao, Shigang Chen, Min Chen, and Yibei Ling. 2015. Hyper-Compact Virtual Estimators for Big Network Data Based on Register Sharing. *SIGMETRICS Perform. Eval. Rev.* 43 (jun 2015), 417–428.

[62] Jiarong Xing, Yiming Qiu, Kuo-Feng Hsu, Hongyi Liu, Matty Kadosh, Alan Lo, Aditya Akella, Thomas Anderson, Arvind Krishnamurthy, T. S. Eugene Ng, and Ang Chen. 2021. A Vision for Runtime Programmable Networks *(HotNets '21)*. Association for Computing Machinery, New York, NY, USA, 91–98. https://doi.org/10.1145/3484266.3487377

[63] Tong Yang, Jie Jiang, Peng Liu, Qun Huang, Junzhi Gong, Yang Zhou, Rui Miao, Xiaoming Li, and Steve Uhlig. 2018. Elastic Sketch: Adaptive and Fast Network-Wide Measurements. In *Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication* (Budapest, Hungary) *(SIGCOMM '18)*. Association for Computing

Machinery, New York, NY, USA, 561–575. https://doi.org/10.1145/3230543.3230544

[64] Raphael Yuster and Uri Zwick. 2005. Fast Sparse Matrix Multiplication. *ACM Transaction on Algorithms* 1, 1 (jul 2005), 2–13. https://doi.org/10.1145/1077464.1077466

[65] Ying Zhang. 2013. An Adaptive Flow Counting Method for Anomaly Detection in SDN *(CoNEXT '13)*. Association for Computing Machinery, New York, NY, USA, 25–30. https://doi.org/10.1145/2535372.2535411

[66] Zheng Zhang, Yong Xu, Jian Yang, Xuelong Li, and David Zhang. 2015. A survey of sparse representation: algorithms and applications. *IEEE access* 3 (2015), 490–530.

[67] Zongyi Zhao, Xingang Shi, Zhiliang Wang, Qing Li, Han Zhang, and Xia Yin. 2021. Efficient and Accurate Flow Record Collection With HashFlow. *IEEE Transactions on Parallel and Distributed Systems* 33, 5 (2021), 1069–1083.

A STORAGE DATA STRUCTURES

A.1 Cuckoo Hash Tables

A CHT [44] can be seen as a dictionary with constant lookup time. It is composed of d tables $T_0 \dots T_{d-1}$ with 2^r buckets each, where d hash functions $h_0 \dots h_{d-1}$ are used to map each key x to the range $[0, 2^r - 1]$ such that x is stored in one of the d buckets $T_i[h_i(x)]$. To insert an item x, first, an empty position is searched among the d designated buckets. If a free cell is found, x is inserted there and the procedure ends. If not, a bucket $T_i[h_i(x)]$ is randomly chosen. Item x is then inserted in this bucket, replacing the old item y previously stored there. If possible, y is reinserted in one of its other available d buckets, i.e., $T_j[h_j(y)]$ for $j \neq i$. Otherwise, the procedure is repeated until all elements are stored.

Quotienting is a technique, originally proposed by Knuth [37], to reduce the memory needed to store keys: consider a universe U comprising 2^u elements of u bits each and d bijective functions $m_i : U \longrightarrow U$; we use the r least significant bits of $m_i(x)$ as the hash function $h_i(x)$, and store as key in $T_i[h_i(x)]$ only the remaining $u - r$ bits of $m_i(x)$, i.e., the *quotient* of $m_i(x)$. This method enables unambiguous identification of the items stored in a cuckoo table while reducing the memory cost of each item from u to $u - r$. Note that quotienting provides significant savings when $u \approx r$, i.e., when a significant fraction of the universe is stored in the qCHT. This is the case of SPADA's *Sketch Data*, as shown in Section 3.3.1.

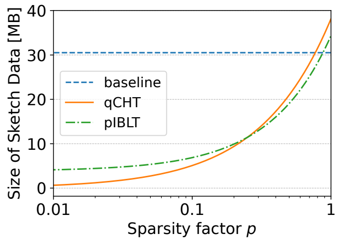
A.2 Invertible Bloom Lookup Tables

IBLT is a data structure that enables storing key-value pairs with constant insertion time. An IBLT [27] uses d hashes to identify d different buckets where to insert the keys. To solve key collisions, each entry also stores a counter that keeps track of how many keys are in the same bucket and the XOR of these keys. Hence, each bucket contains a field for the XORed keys, a key counter for the number of colliding keys, and a value-store to sum all the values associated with the XORed keys. Data inside the IBLT can be retrieved using a decoding procedure called `ListEntries` that performs what is called a "peeling process". Intuitively, the peeling process operates by identifying the buckets where the key counter is equal to 1, i.e., only one value was stored there, then removing that value from all the d associated buckets. Hopefully, then other buckets end up having their key counters equal to one, hence the process continues iteratively. This procedure can retrieve all the `<key,value>` pairs if the load factor (i.e., the ratio between the number of inserted keys and the number of overall available buckets) of the IBLT is below a certain "peeling threshold". In particular, the best achievable threshold according to [58] is around 0.82 (achieved with $d = 3$). On the other hand, the pIBLT structure we introduce in Section 3.3.2 supports a much higher threshold load of 0.97 (achieved with $d = 4$) up to which all the `<key,value>` pairs can be retrieved *with high probability*[6].

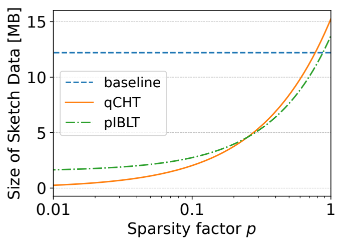
[6]The `<key,value>` pairs are extracted from the pIBLT solving a linear system $A \cdot \mathbf{x} = \mathbf{b}$. For a load factor up to 0.97, this system has a unique solution *with high probability*, i.e., A being an $m \times n$ matrix, the probability approaches 1 for $m, n \longrightarrow \infty$ [19, 20, 46].

B ADDITIONAL RESULTS ON MEMORY SIZING

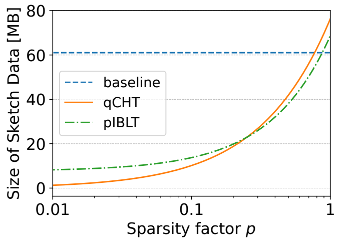
To complement the analysis in Section 3.4, we hereby provide additional plots showing the trade-offs between memory size of the *Sketch Data*, number of flows under monitoring, number of buckets and data sparsity for other configurations (Figure 10).



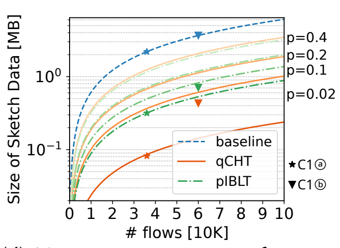
(a) Memory requirements varying sparsity factor p (n_s=500K, m=64).



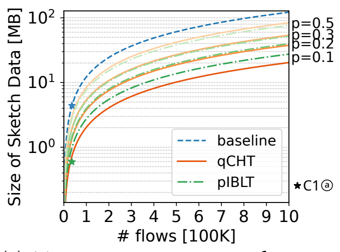
(b) Memory requirements varying sparsity factor p (n_s=100K, m=128).



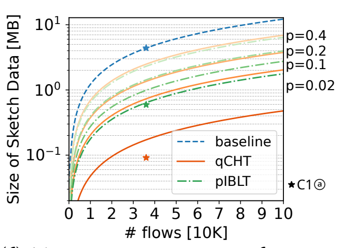
(c) Memory requirements varying sparsity factor p (n_s=500K, m=128).



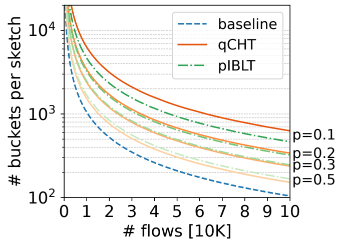
(d) Memory requirements for up to 100K flows varying p (m=64).



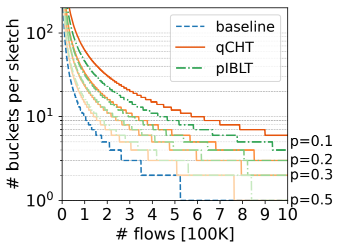
(e) Memory requirements for up to 1M flows varying p (m=128).



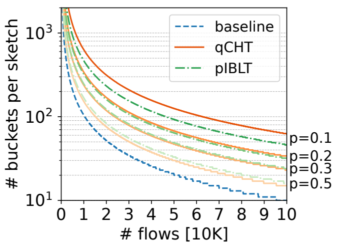
(f) Memory requirements for up to 100K flows varying p (m=128).



(g) Max. number of buckets m per sketch in 10 MB (up to 100K flows).



(h) Max. number of buckets m per sketch in 1 MB (up to 1M flows).



(i) Max. number of buckets m per sketch in 1 MB (up to 100K flows).

Fig. 10. Relationship between data sparsity and memory requirements of SPADA data structures. Only *Sketch Data* is shown as the size of the *Flow Map* does not change with sparsity and across solutions. All plots assume bucket size s_c = 8 bits. Additional configurations of all parameters are shown w.r.t. Figure 5.

C FPGA RESOURCE REQUIREMENTS

Table 6 reports the hardware resource requirements for SPADA building blocks namely MAT, CHT, qCHT, and pIBLT. Data structures are dimensioned for 2^{14} entries in the *Flow Map*, each with 2^{16} buckets of 16 bits in the *Sketch Data*, assuming "virtual sketches" with $m = 32$ buckets unless otherwise specified. The table reports the number of LUTs (expressed in thousands, K, and in percentage, %) used as logic, those used as distributed RAMs (LUTRAM), and the number of flip-flops and Block RAMs (BRAM). It is worth mentioning that due to the Vitis P4 framework architecture, the MAT Table 6(top) is mainly mapped to the BRAM memory element and that, as expected, the 5-tuple one requires more resources as it has a bigger flow key.

Table 6 (middle) details the FPGA hardware resources required by CHT and qCHT with 1, 2, and 4 datapaths, for $m = 32$. We recall that the qCHT is a CHT that stores a quotient instead of the full key, hence saving a significant amount of memory as the table highlights. The extra memory (mainly BRAM) for multiple datapaths is due to the load balancing and interconnection overhead. This overhead is fixed, i.e., does not depend on the hash table size, and is also related to the specific synthesizer optimizations. In particular, we observed that the memory required to synthesize a single datapath diminishes by increasing the number of datapaths. With multiple datapaths, such memory reduction may compensate for the extra memory required by the load balancer.

Table 6 (bottom) details the memory requirements for the pIBLT with 2^{16} counters and a bitmap B of size 2^{19} bits, assuming "virtual sketches" with $m = 32$. It is worth mentioning that in this case pIBLT occupies fewer resources with respect to qCHT. This is mainly due to the limited amount of sketch counters m. Increasing m would lead to a bigger bitmap B and hence bigger pIBLT.

Table 6. MAT, (q)CHT, and pIBLT resources utilization.

Datapaths	**Building Block**	**LUT** [K - %]	**LUTRAM** [K - %]	**FF** [K - %]	**BRAM** [# - %]
-	MAT (Src IP)	7.3 - 0.57	2.3 - 0.4	12.2 - 0.47	73 - 3.62
-	MAT (5-tuple)	10.3 - 0.8	2.8 - 0.5	18.4 - 0.71	109 - 5.41
1	CHT	53.2 - 4.09	43.7 - 7.27	41.6 - 1.6	1 - 0.05
	qCHT	37.3 - 2.87	29.3 - 4.88	32.2 - 1.24	1 - 0.05
2	CHT	57.8 - 4.44	43.3 - 7.21	58.2 - 2.32	3 - 0.15
	qCHT	48.3 - 3.71	34.9 - 5.81	57.8 - 2.22	3 - 0.15
4	CHT	80 - 6.14	53.8 - 8.96	109.5 - 4.20	5 - 0.25
	qCHT	70.9 - 5.44	46.2 - 7.70	108.8 - 4.17	5 - 0.25
-	pIBLT	40.5 - 3.11	35.4 - 5.9	8.8 - 0.34	1 - 0.05

Received 27 June 2023; revised 10 October 2023; accepted 18 October 2023