Hyper-Compact Virtual Estimators for Big Network Data Based on Register Sharing

Qingjun Xiao[†‡] Shigang Chen[†] Min Chen[†] Yibei Ling[§]
[†]Department of Computer & Information Science & Engineering, University of Florida, Gainesville, FL, USA
[‡]Key Lab of Computer Network and Information Integration (Southeast University), Ministry of Education, China
[§]Telcordia Technologies & Applied Research, Ericsson, USA
Email: csqjxiao@seu.edu.cn {sgchen, min}@cise.ufl.edu yibei.ling@gmail.com

ABSTRACT

Cardinality estimation over big network data consisting of numerous flows is a fundamental problem with many practical applications. Traditionally the research on this problem focused on using a small amount of memory to estimate each flow's cardinality from a large range (up to 10^9). However, although the memory needed for each flow has been greatly compressed, when there is an extremely large number of flows, the overall memory demand can still be very high, exceeding the availability under some important scenarios, such as implementing online measurement modules in network processors using only on-chip cache memory. In this paper, instead of allocating a separated data structure (called *estimator*) for each flow, we take a different path by viewing all the flows together as a whole: Each flow is allocated with a virtual estimator, and these virtual estimators share a common memory space. We discover that sharing at the register (multi-bit) level is superior than sharing at the bit level. We propose a framework of virtual estimators that allows us to apply the idea of sharing to an array of cardinality estimation solutions, achieving far better memory efficiency than the best existing work. Our experiment shows that the new solution can work in a tight memory space of less than 1 bit per flow or even one tenth of a bit per flow — a quest that has never been realized before.

Categories and Subject Descriptors

C.4 [**Performance of Systems**]: [Measurement Techniques]; C.2.3 [**Computer Communication Networks**]: Network Operations – Network Management

Keywords

Big Network Data, Flow Monitoring, Cardinality Estimation

1. INTRODUCTION

Cardinality estimation is one of the fundamental problems in network traffic measurement [9,10,14,17,25,26]. In a general definition, it is to estimate the number of *distinct* elements in each flow during a measurement period. The *flows* under measurement may be per-source flows, per-destination flows, per-source/destination flows, TCP flows, WWW flows, P2P flows, or abstract flows. The *elements* may be destination addresses, source addresses, ports, values in other header fields, or even keywords that appear in the payload of packets in the flow.

Practical Importance: The cardinality problem has many practical applications. For example, if we treat all packets sent from the same *source address* as a flow (per-source flow), we may use a cardinality estimation module at a gateway or firewall to detect scanners by measuring the number of *distinct destination addresses* in each flow. In this case, packets belonging to a flow are identified by their common source address (also called *flow label*). The elements under measurement are the destination addresses in the headers of the packets. In the opposite example, we may treat all packets to a common destination as a flow and count the number of distinct source addresses in each flow. When we observe the cardinality of a certain flow suddenly surges, it may signal a DDoS attack against the destination address of the flow. For other applications, a large server farm may learn the popularity of its content by tracking the number of distinct users that access each file, where all accesses to a file form an (abstract) flow; an institutional gateway may determine the popularity of external web content for caching priority by tracking the number of outbound web requests for each web content, where all requests from different users to a common URL form a flow.

In yet another example, if Google treats all client IPs that query a keyword as a flow, the cardinality of the flow suggests the popularity of the keyword being searched. In this case, the flow label is the keyword under query. The estimator that works on per-keyword flows may be implemented as a function module at the web server. According to a recent paper [14], various data analysis systems at Google, such as Sawzall, Dremel and PowerDrill, estimate the cardinalities of very large data sets on a daily basis. As pointed out in [14], cardinality estimation over large data sets presents a challenge in terms of computational resources, and memory in particular; for the PowerDrill system, a non-negligible fraction of queries historically could not be computed because they exceeded the available memory.

State of the Art: To deal with big data consisting of a very large number of flows, we must conserve memory space when designing a cardinality estimation module. For this purpose, a series of solutions were developed in the past, including PCSA [12], MultiresolutionBitmap [10] (which is a generalization of LinearCounting [22]), MinCount [3], LogLog [8], and HyperLogLog [11]. They all allocate a separate data structure, called *estimator*, for every flow. Each estimator contains a certain number of registers, bitmaps or other elementary data structures. The most compact estimator in [11] requires hundreds of bytes to ensure a large estimation range and a good estimation accuracy.

Challenges: However, as the Internet moves into the era of big network data, hundreds of bytes per flow can be too much in some important scenarios — Modern high-speed routers forward packets at the speed of hundreds of Gigabits or even hundreds of Terabits per second [13]. The number of data flows that traverse a core router can be in tens of millions. Simultaneous tracking of such a large number of flows (each of which needs hundreds of bytes) brings a great challenge. The reason is that, in order to sustain high throughput, routers forward packets from incoming ports to outgoing ports via switching fabric, bypassing main memory and CPU. If one wants to apply cardinality estimation as an online module to process packets in real time, one way is to implement it on network processors at the incoming/outgoing ports and use on-chip cache memory. However, the commonly-used cache on processor chips is SRAM, typically a few megabytes, which may have to be shared among multiple functions for routing, performance, measurement, and/or security purposes. In such a context, the memory that can be allocated for the function of cardinality estimation may be even less than 1 bit per flow.

In another scenario, suppose a major web search company wants to know how many different users have searched the same phrase (question or sentence) each day, which provides information on phrase popularity, useful in optimizing search performance or studying social trends on the Internet [2]. This is a cardinality estimation problem, where all search records for a given phrase form a flow. The number of flows (phrases, questions, sentences) can be in billions. Of course, we can resort to a data center for such big data, but it will certainly be welcome if one can find a novel solution that deals with an extremely large number of flows in the memory of a cheap commodity computer.

Our Contribution: After decades of development [3, 8, 10–12, 22], it appears to be very difficult to further compress the size of an individual estimator much below hundreds of bits, without sacrificing estimation range or accuracy. Recently, an interesting idea was proposed to let different estimators (each for one flow) share bits [16, 17, 25], so that bits unused by one can be picked up by another. Along this line, we make three new contributions: First, we discover that sharing bits is actually inefficient because of too much noise introduced between estimators. Sharing space is good, but it should be done differently at the register level, not at the bit level, where a register is a multi-bit data structure that will be introduced later. Second, sharing has only been applied to bitmap and PCSA [12], an early work dated back to 1985. We develop a framework of virtual estimators which enables memory sharing for the recent cardinality estimation solutions, including LogLog [8] and HyperLogLog [11], with the latter being the best existing work. Third, we fully develop the virtual HyperLogLog solution, with a new procedure for recording per-flow information in the shared space, a set of formulas for estimating per-flow cardinality with noise removal, and the analytical results for estimation error under register sharing. We show that the new solution can work in a tight memory space of less than 1 bit per flow or even one tenth of a bit per flow — a quest that has never been realized before.

The rest of the paper is organized as follows: Section 2 discusses the related work. Section 3 introduces our new design of register sharing. Section 4 proposes a framework for constructing virtual estimators based on register sharing. Section 5 presents the detailed design of a memory-efficient cardinality estimation solution under the framework, and Section 6 analyzes the new solution's mean and variance. Section 7 evaluates the performance of the proposed estimation solution through experiments based on real traffic traces. Section 8 draws the conclusion.

2. RELATED WORK

Cardinality estimation is different from the related problem of *flow-size estimation* [15], which measures the number of elements (e.g., packets or bytes) in each flow through CountMin Sketches (a generalized tool for estimating the frequency of each element in a multiset) [5], Counter Braids [18, 19], Probabilistic Lossy Counting [7], Randomized Counter Sharing [15], etc., with the goal of learning flow distribution or identifying heavy hitters. Consider all packets from a source address as a flow. Suppose the source sends 10,000 packets to a single destination address. The flow size is 10,000 when we measure the number of packets, but the flow cardinality is just one if we measure the *distinct* number of destination addresses in this flow. In short, cardinality estimation needs to remove duplicates, which makes it a more difficult problem because it has to somehow "remember" the observed elements for duplicate removal, while measuring a flow size only needs a counter.

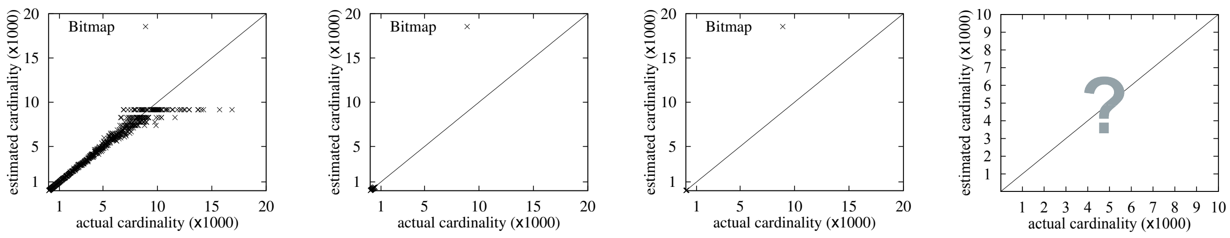
Hash table and Bitmap: It is too costly to design an estimator based on a hash table that stores all elements to remove duplicates. Instead, we may use a bitmap [22]: Initially all bits are zeros. Each arrival element is hashed to a bit which is then set to one. Duplicates are automatically filtered out since they are mapped to the same bit. At the end of a measurement period, the cardinality estimation is $\hat{n} = -b \ln V$ [22], where b is the number of bits used, V is the fraction of bits whose values remain zeros, and $\hat{n}$ is the estimated flow cardinality.

The problem of bitmap is that the estimation range is bounded by $b \ln b$. Hence, the bitmap has to be huge to handle a very large flow. Fig. 1 shows the simulation results, where the bitmap size is 1280 bits per flow in the leftmost plot, 96 bits per flow in the second plot, and 32 bits per flow in the third, respectively. Each flow is represented by a point, whose x-coordinate is the true cardinality and y-coordinate is the estimated cardinality. The equality line is also shown. The closer a point is to the line, the more accurate the estimation is. The leftmost plot clearly shows a limited estimation range. As the bitmap size shrinks, the range shrinks quickly, as shown by plots (b)-(d). Note that "less than 1 bit" per flow will not work for the bitmap approach. Variants of the bitmap approach also have the problem of limited estimation range [23–26].

MultiResolutionBitmap and PCSA: Sampling is one of the main methods in the literature for dealing with the estimation range problem. MultiResolutionBitmap [10] is essentially the concatenation of multiple bitmaps, each having a different sampling probability. If we let the sampling probabilities be $\frac{1}{2}$, $(\frac{1}{2})^2$, ..., $(\frac{1}{2})^w$ and set each bitmap to its minimum size (a single bit), then we have the smallest MultiResolutionBitmap, equivalent to an FM sketch of the earlier PCSA [12]. An FM sketch, also referred to as a *register* in the literature, can give an estimation up to 2^w, where w is the number of bits in the register. For example, $w = 32$ for an estimation range up to 2^{32}.

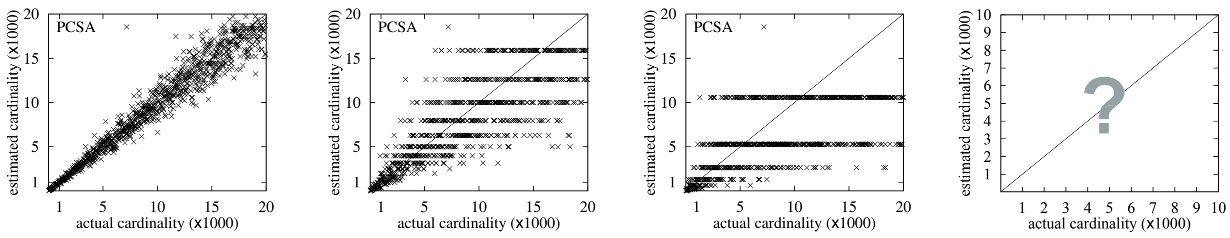
However, the estimation result from a single register is very inaccurate. To improve accuracy, FM uses multiple registers and returns the average of their estimations. Fig. 2 presents the simulation results of FM. It clearly has a larger estimation range, but its estimation accuracy is low even when there are 40 registers in the first plot. The estimation results are discrete when there are just a few registers in the second and third plots.

LogLog and HyperLogLog: LogLog [8] and HyperLogLog [11] were proposed to compress the size of each register from 32 bits to



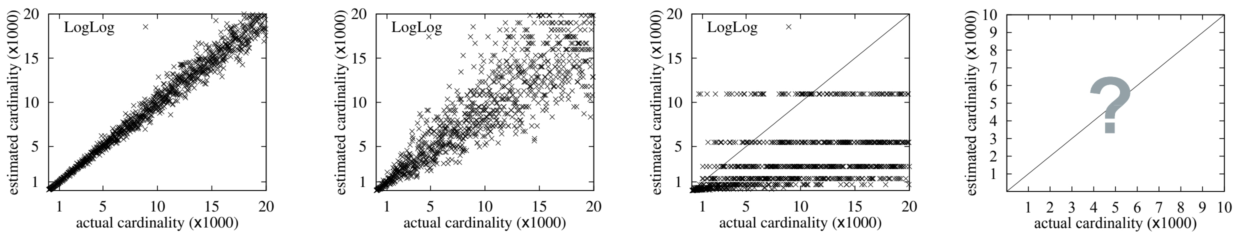
(a) 1280 bits per flow (b) 96 bits per flow (c) 32 bits per flow (d) less than 1 bit per flow

Figure 1: Measurement results of the bitmap approach, whose estimation range is limited. Each flow is represented by one point. The x-coordinate is the true cardinality, and the y-coordinate is the estimated cardinality. The closer a point is to the equality line, the more accurate the estimation is.



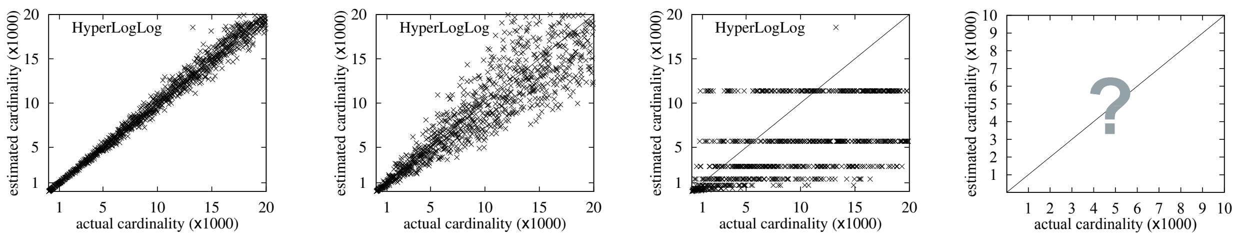
(a) 1280 bits per flow, 40 registers of 32 bits each, 13% error (b) 96 bits per flow, 3 registers of 32 bits each (c) 32 bits per flow, 1 register of 32 bits (d) less than 1 bit per flow

Figure 2: Measurement results of FM or PCSA.



(a) 1280 bits per flow, 256 registers of 5 bits each, 8.1% error. (b) 80 bits per flow, 16 registers of 5 bits each, 33% error. (c) 5 bits per flow, 1 register of 5 bits (d) less than 1 bit per flow

Figure 3: Measurement results of LogLog.



(a) 1280 bits per flow, 256 registers of 5 bits each, 6.5% error (b) 80 bits per flow, 16 registers of 5 bits each, 26% error (c) 5 bits per flow, 1 register of 5 bits (d) less than 1 bit per flow

Figure 4: Measurement results of HyperLogLog.

5 bits for the same estimation range of 2^{32}. Their performance is presented in Fig. 3 and Fig. 4. The estimation accuracy of LogLog and HyperLogLog (HLL) is much improved as compared with PCSA, because smaller registers mean there are more of them under the same memory constraint, which drives the estimation variance down. However, they do not work well for 80 bits in the second plot of Fig. 3 and Fig. 4 (with the relative standard error being 33% for LogLog and 26% for HLL), let alone less than one bit per flow. The accuracy of HLL is a little better than that of LogLog.

Performance Summary: The performance of the traditional cardinality estimators is summarized in Table 1, where MinCount [3, 4] takes a different approach by hashing each arrival element and keeping a number of smallest hash values, from which the estimation is made (using the range of the smallest hash values). In the second column, m is the number of smallest hash values kept by MinCount for each flow, the number of bits used by MultiResolutionBitmap, or the number of registers used by other approaches. The total memory cost is m multiplied by the size of each memory unit (hash value, bit or register).

Table 1: Comparison of the prior art.

Solution	Std. Err. (σ)	Mem Units	Mem (σ=5%)
MinCount	$1.00/\sqrt{m}$	≤32 bits	1600 bytes †
MultiResBitmap	$\approx 4.4/\sqrt{m}$	1 bit	968 bytes
PCSA	$0.78/\sqrt{m}$	32-bit registers	974 bytes
LogLog	$1.30/\sqrt{m}$	5-bit registers	423 bytes
HyperLogLog	$1.04/\sqrt{m}$	5-bit registers	271 bytes

† For MinCount, we assume the size of its memory units is 32 bits, and each unit stores the 32-bit hash value of a stream element.

For a single flow, the memory needed to control the standard error within 5% of the actual cardinality is given in the last column, which shows the progress in memory saving over the past decades: If we use PCSA as the initial benchmark, the seminal work of LogLog cuts the memory requirement by more than half. The followup HyperLogLog cuts the memory further by more than 30%. HyperLogLog has made great impact on IT industry and was adopted by Google [14], PostgreSQL, file-sharing P2P systems [21], and DDoS attack detection systems [11].

3. OUR NEW APPROACH OF REGISTER SHARING AND VIRTUAL ESTIMATORS

3.1 Motivation: Waste of Space

The traditional solutions allocate one estimator for each flow, which is however a serious waste of space. As an example, we download traffic traces from CAIDA (Cooperative Association for Internet Data Analysis) [1]. Consider per-source flows. The cardinality of each flow is the number of distinct destination addresses contacted by a source. We illustrate the distribution of the flow cardinalities in Fig. 5, where the measurement period is 10 minutes and each point shows the number (y-coordinate) of flows that have a certain cardinality (x-coordinate). A roughly straight line on a log-log plot is often considered as the signature of a power law distribution. In this figure, the line is roughly $y = 3 \cdot 10^4 \cdot x^{-1.7}$. This log-scale figure demonstrates that the vast majority of flows have small cardinalities, while a small number of flows have large cardinalities.

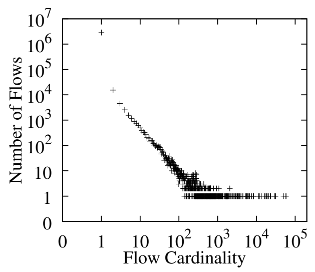


Figure 5: Flow distribution: each point shows the number (y-coordinate) of flows having a certain cardinality (x-coordinate). The average cardinality of all flows is about two.

Without knowing the flows' cardinalities beforehand (which are in fact what we want to figure out), the estimators of all flows are set according to the maximum range of cardinality, requiring hundreds of bits even for the best estimator. However, if a flow turns out to be small, e.g., with a cardinality of 1, most of the bits will be wasted.

3.2 Sharing at bit level?

One way to make use of unused bits is to share bits among the estimators. Two solutions were proposed for sharing among bitmaps [25] and FM sketches [17]. In the compact spread estimator (CSE) [25], a bitmap is allocated for each flow, but all bitmaps share their bits from a common bit pool. The problem is that it is difficult to extend the estimation range of bitmaps without incurring large overhead or causing estimation inaccuracy.

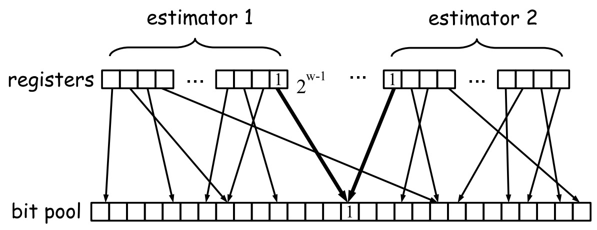


Figure 6: Bit sharing in [17], where the FM sketches (registers) share their individual bits from a common bit pool.

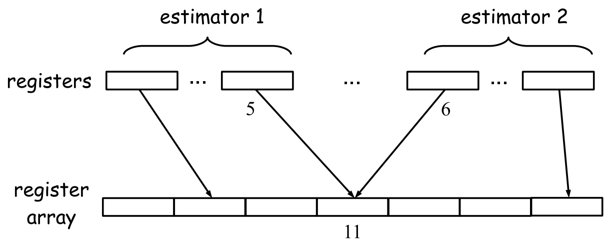


Figure 7: Register sharing, where the estimators share their registers from a common register pool.

In the probabilistic multiplicity counting solution (PMC) [17], an estimator with multiple FM sketches is allocated to each flow. In fact, PMC was originally designed for estimating the flow size (i.e., the number of packets in each flow), but it can be easily modified for estimating the flow cardinality, which is not commonly true for other flow-size estimators. As illustrated in Figure 6, the FM sketches (called registers) of all estimators share their *bits* from a common bit pool uniformly at random, so that mostly unused higher-order bits in the registers can be utilized. However, sharing introduces noise across estimators, which we explain through an example: Without going into too much technical details which can be found in the original paper [17], roughly speaking, when the ith bit in a register (FM sketch) is set to one, it means 2^i packets are recorded by the register on average, where $0 \leq i < w$ and w is the number of bits in each register. In the Figure 6, suppose estimator 1 is for a small flow. So the high-order bits in its sketches should be zeros. If a high-order bit in estimator 1 happens to share the same bit in the common pool with a low-order bit in estimator 2, when the low-order bit is set to one by estimator 2, the high-order bit in estimator 1 will also become one. The low-order bit in estimator 2 only represents one element, but the noise it induces represents 2^{w-1} elements for estimator 1. Although novel statistical methods can be used to remove noise, the noise of bit-level sharing is too high to take the full potential of the sharing idea, which we will demonstrate through experiments. Sharing high-order bits only with high-order bits will not work well either because the underutilized high-order bits will remain underutilized.

3.3 Register Sharing and Virtual Estimators

Our idea is to share at the register level, as illustrated in Figure 7. The estimators of different flows share their registers from a common register array M. Given a fixed register array, we dynamically create an estimator for a new flow by randomly drawing a number of registers from the array. In a sense, the array of registers are physical, but the estimators are logical because they are created on the fly without additional memory allocation. Hence, they are called *virtual estimators*.

Suppose a system allocates a certain amount of physical memory to the function of cardinality estimation. The number of bits available may be smaller than the number of flows. If this is the case, the number of registers in M will certainly be even smaller. Each register is thus shared by many virtual estimators, ensuring that the register is fully utilized.

Consider the virtual estimator of an arbitrary flow. What it estimates is actually the cardinality of the flow plus the noise introduced by other flows that share its registers. Refer to Figure 7 where estimator 1 and estimator 2 share a common register. If the register records 5 elements from the flow of estimator 1 and 6 elements from the flow of estimator 2, the final result will be 11 elements recorded. From the viewpoint of estimator 1, the register carries its flow's information as well as noise from other flows. The same is true from the viewpoint of estimator 2.

Because the registers in all virtual estimators are randomly picked, there is an equal opportunity for any two registers from different estimators to be mapped to the same physical register in M. Hence, as one virtual estimator records an element of its flow into one of its registers, the probability for this operation to cause noise to any other virtual estimator is the same. When there are a large number of virtual estimators and each of them randomly chooses a large number of registers, the noise that they cause to each other will be roughly uniform. Such uniform noise can be measured and removed.

One may argue that similar noise also exists for register-level sharing. An estimator of a small flow may share a register with an estimator of a large flow. First, the elements of the large flow will be spread among its hundreds of registers. Each register carries much smaller noise than a single bit in PMC can do. Second, the number of large flows is often exponentially fewer than the number of small flows; see Fig. 5 for example. That means the number of registers that carry large noise account for a small fraction of all registers in M. If the estimator of a small flow contains one or a few registers of large noise, the technique of harmonic averaging can be used to remove the effect of such outliers (which is already done by [8, 11]). On the contrary, for PMC, all bits that are set to ones in V can cause large noise.

3.4 Counter Sharing for Flow Size — A Different Problem

We have explained in the previous section that flow-size estimation is a different problem than cardinality estimation. Sharing counters has been applied to reduce memory overhead for estimating the sizes of a large number of flows [5, 15, 18]. Take CountMin [5] as example, which resembles a segmented counting Bloom filter (organized in a two-dimensional matrix), where the arrival of each packet of a flow causes the k counters of the flow to increase by one. The minimal value of the k counters is used as an estimation of the flow size. This approach cannot solve the problem of cardinality estimation because the counter does not "remember" the elements that it has seen for duplicate removal. As a minor note, although the minimal value of the k counters has the least noise, it does have noise, which can be significant when the number of bits is smaller than the number of flows — Because each counter has multiple bits, the number of counters will be far smaller than the number of flows. Therefore it is highly probable that most counters are shared by multiple flows, and thus even the minimal value of the k counters carries the combined size of multiple flows.

4. A FRAMEWORK FOR VIRTUAL-ESTIMATOR SOLUTIONS

We propose a unified framework for developing virtual-estimator solutions that enable register-level sharing for mainstream sketches such as PCSA [12], LogLog [8], and HyperLogLog [11]. In the next section, we will show as an example how to apply the framework to HLL for a virtual-estimator solution denoted as vHLL. The notations used are summarized in Table 2 for quick reference.

Table 2: Notations

M	a physical array of registers
m	number of registers in M
M_f	a subset of registers from M used by the virtual estimator of flow f
s	number of registers used by a virtual estimator
$H_i(f)$	a hash function that maps the ith register of M_f to a physical register in M
n_f	true cardinality of flow f
$\hat{n_f}$	estimated cardinality of flow f
$\hat{n_s}$	an estimation made based on M_f, which record both elements of flow f and elements from other flows as noise
n	true combined cardinality of all flows
$\hat{n}$	an estimated value of n

In the framework, we use a single array M of m registers to store the cardinality information of all flows. The ith register in the array is denoted by $M[i]$, $0 \leq i < m$. The size of the registers is set based on the type of estimators used [8, 11, 12] and the maximum range of cardinality to be estimated. For example, in the vHLL solution, the size of registers is five bits, in order to measure big cardinalities up to 4×10^9. Each flow has s virtual registers that are randomly selected from M through hash functions. These registers logically form a virtual estimator, denoted as M_f, where f is the label of the flow. The ith register of the virtual estimator, denoted as $M_f[i]$, $0 \leq i < s$, is selected from M as follows:

$$M_f[i] = M[H_i(f)], \tag{1}$$

where $H_i(...)$ is a hash function whose range is $[0, m)$. We want to stress that M_f is not a separate data structure. It is merely a logical construction based on registers selected from M, and it is not explicitly constructed during online operation. In all our later formulas, one should treat the notation $M_f[i]$ simply as $M[H_i(f)]$, referring to a register in M.

The hash function H_i, $0 \leq i < s$, can be implemented from a master function $H(...)$ as follows:

$$\begin{aligned} H_i(f) &= H(f \oplus R[i]) \quad \text{or} \\ H_i(f) &= H(f \,|\, i), \end{aligned} \tag{2}$$

where '$|$' is the concatenation operator, '$\oplus$' is the XOR operator, and $R[i]$ is a constant whose bits differ randomly for different indexes i. The master hash function H we have adopted in our experiments is 64-bit MURMUR3 hash. According to an online technical document, MURMUR3 performs better than many other hash functions, including JENKINS' LOOKUP3, CITY, and SPOOKY [20].

At the beginning of each measurement period, all registers are reset to zeros. The arrival stream of elements is abstracted as a sequence of $\langle f, e \rangle$ pairs, where f is a flow label and e is an element of the flow. For example, if a router measures per-source flows for their numbers of distinct destination addresses, it extracts the source address of each arrival packet as the flow label and the destination address from the IP header as the element to be recorded. For each pair $\langle f, e\rangle$, we record e in one of the registers of M_f based on the methods in [12], [8] or [11], depending on which one is used.

At the end of a measurement period, the register array M is offloaded by a server for long-term storage. Given a flow label f in offline query, we reconstruct its virtual estimator M_f by copying s registers from M at indices $H_i(f), 0 \le i < s$. Let n_s be the number of distinct elements recorded by M_f, which is the flow's cardinality plus the noise introduced by other flows due to register sharing. Let n_f be the actual cardinality of flow f. The noise term is $n_s - n_f$. We use the estimation formula from [12], [8] or [11] (depending on which one is used) to give an estimation $\hat{n_s}$ of n_s. Below we focus on noise estimation.

Let n be the sum of all flows' cardinalities. From the flow f's point of view, the elements of all other flows, $(n - n_f)$ of them, are noise. Let Y be a random variable for the number of noise elements recorded by an arbitrary register in M. When the number of flows and the number of registers per estimator are both sufficiently large and the cardinality of any flow is negligibly small when comparing with n, Y approximately follows the binomial distribution $Bino(n - n_f, \frac{1}{m})$, because each noise element has approximately an equal chance to be recorded by any register due to the random selection of registers by virtual estimators. Hence,

$$E(Y) \quad = \quad \frac{n - n_f}{m}.$$

The total noise, $n_s - n_f$, is the sum of individual noises in the s registers of M_f. Hence, $n_s - n_f$ can be considered as the sum of s independent random variables of $Bino(n - n_f, \frac{1}{m})$.

$$E(n_s - n_f) = s\,E(Y) = s\,\frac{n - n_f}{m} \tag{3}$$

By the law of large numbers in the probability theory, the relative variance $Var(\frac{n_s - n_f}{E(n_s - n_f)})$ approaches to zero when s is large. In this case, $E(n_s - n_f)$ can be approximated by an instance value, $n_s - n_f$. We have

$$\begin{aligned} n_s - n_f &\approx \frac{n - n_f}{m} s \\ n_f &\approx \frac{ms}{m - s}\left(\frac{n_s}{s} - \frac{n}{m}\right). \end{aligned} \tag{4}$$

We define a *grand flow* as the combination of all flows. With a few hundreds of extra bytes and applying the HyperLogLog, we can obtain an accurate estimation $\hat{n}$ for n (see Table 1), while the additional memory overhead is negligible when comparing with the memory space M. Alternatively, since the elements of the grand flow distribute approximately in uniform over M, we can use the entire register array M as an estimator to give an estimation for n (using HyperLogLog, for example).

Let $\hat{n_f}$ be our estimation of n_f. We have the following estimation formula from (4).

$$\hat{n_f} \quad = \quad \frac{ms}{m - s} \cdot \left(\frac{\hat{n_s}}{s} - \frac{\hat{n}}{m}\right) \tag{5}$$

In the next section, we will select vHLL, i.e., virtual HyperLogLog, to discuss its operations and performance in details.

5. VIRTUAL HYPERLOGLOG ESTIMATOR

In this section, as an example, we apply the framework of virtual estimators on HyperLogLog for a new solution, vHLL, based on register-level sharing. This solution consists of two components: one for recording the stream of packets in the virtual HyperLogLog estimators, and the other for estimating the cardinality of an arbitrary flow f.

5.1 Record Flow Elements in Virtual Estimator

Consider a flow f. When a measurement period begins, all registers in its virtual estimator M_f are reset to zeros. For each arrival element e of flow f, we perform the hashing below:

$$\begin{aligned} H(e) &= \langle x_1 x_2 ...\rangle \\ p &= \langle x_1 x_2 \ldots x_b\rangle \\ q &= \langle x_{b+1} x_{b+2}, \ldots\rangle, \end{aligned} \tag{6}$$

where $\langle x_1 x_2 ...\rangle$ is binary format of the hash output $H(e)$, p denotes the leading b bits with b equal to $\log_2 s$, and q represents the remaining bits. Using the value of p, we can map e pseudo-randomly to a register $M_f[p \bmod s]$. For clarity, we will breviate $M_f[p \bmod s]$ simply as $M_f[p]$ afterwards.

The operation of recording e is simple: Let $\rho(q)$ be the number of leading zeros in q plus one; for example, if $q = 001...$, then $\rho(q) = 3$. Clearly, the probability of $\rho(q) = i$ is $(\frac{1}{2})^i$, for $\forall i > 0$. We update $M_f[p]$ if its current value is smaller than $\rho(q)$. Namely,

$$M_f[p] := \max\big(M_f[p],\ \rho(q)\big), \tag{7}$$

where := is assignment operator. Hence, $M_f[p]$ has recorded (one plus) the longest run of leading zeros from any element mapped to the register. Suppose $M_f[p] = M[H_p(f)]$ as in (1), and $H_p(f) = H(f \,|\, p)$ as in (2). Combining (7), (1) and (2), we have

$$M[H(f \,|\, p)] := \max\big(M[H(f \,|\, p)],\ \rho(q)\big). \tag{8}$$

This assignment requires two hash operations: $H(e)$ for p and q in (6), and $H(f \,|\, p)$. It also requires at most two memory accesses, reading $M[H(f \,|\, p)]$ and writing $M[H(f \,|\, p)]$ back if its value changes. Note that the writing operation happens rarely since the likelihood for $\rho(q) > M[H(f \,|\, p)]$ to happen will decrease exponentially as the register's value increases.

Eq. (8) shows that the operations are actually performed on the physical register array M, and the virtual estimator is logical in the online recording phase.

5.2 Flow Cardinality Estimation

Given a flow label f for offline query, we construct M_f from the stored M. Consider an arbitrary register $M_f[i], 0 \le i < s$. Any element mapped to this register had a probability of $\frac{1}{2^{M_f[i]}}$ to set the register to its current value. Hence, the estimation for the number of elements mapped to this register is $2^{M_f[i]}$ [11].

Recall that n_s is the total number of distinct elements that have been recorded by the estimator M_f, including both elements in flow f and those in other flows that share registers in M_f. In order to estimate n_s, the normalized harmonic mean is applied to aggregate the estimations from all registers in M_f:

$$\hat{n_s} = \alpha_s \cdot s^2 \cdot \Big(\sum_{j=0}^{s-1} 2^{-M_f[j]}\Big)^{-1}, \tag{9}$$

where α_s is a bias correction constant that equals

$$\alpha_s = \Big(s \int_0^\infty \Big(\log_2\Big(\frac{2+u}{1+u}\Big)\Big)^m du\Big)^{-1}. \tag{10}$$

The above equation for constant α_s is complicated. Numerical values are often used in practice: $\alpha_{16} = 0.673$, $\alpha_{32} = 0.697$, $\alpha_{64} = 0.709$, and $\alpha_s = 0.7213/(1 + 1.079/s)$ for $s \ge 128$.

The estimator in (9) is good for large cardinalities, but it is severely biased when dealing with small cardinalities [11]. For a small cardinality, we treat M_f as a bitmap of s bits, with each register $M_f[i]$ converted to one bit, whose value is 1 when $M_f[i] > 0$ or zero otherwise. The estimation formula is $\hat{n_s} = -s \log_2 V$, where V is the fraction of bits in the bitmap that are zeros [22]. This formula is used when the cardinality estimation by (9) is smaller than $2.5s$.

Recall that we can estimate the sum $\hat{n}$ of all flow cardinalities based on a separate estimator or simply from the whole array M using (10) where $\hat{n_s}$ is replaced with $\hat{n}$, s is replaced with m, and M_f is replaced with M. After computing both $\hat{n_s}$ and $\hat{n}$, we use (5) to compute the estimated flow cardinality $\hat{n_f}$.

6. ESTIMATION BIAS AND VARIANCE

This section analyzes the bias and standard error of our vHLL estimator. From [11], we have the following theorem.

THEOREM 1. *Let n_s be the number of distinct elements that are mapped to a HyperLogLog estimator M_f. Suppose the number s of registers in M_f is more than 16.*

- *If n_s is sufficiently large, the estimate $\hat{n_s}$ by (9) is asymptotically almost unbiased in the sense that*
$$\frac{1}{n_s} E(\hat{n_s}) = 1 + \delta_1(n_s) + o(1),$$
where $|\delta_1(n_s)| < 5 \times 10^{-5}$ as soon as $s \geq 16$.
- *The standard error defined as $\frac{1}{n_s}\sqrt{Var(\hat{n_s})}$ satisfies*
$$\frac{1}{n_s}\sqrt{Var(\hat{n_s})} = \frac{\beta_s}{\sqrt{s}} + \delta_2(n_s) + o(1),$$
where $|\delta_2(n_s)| < 5 \times 10^{-4}$ as soon as $s \geq 16$. The constants β_s being bounded, with $\beta_{16} = 1.106$, $\beta_{32} = 1.070$, $\beta_{64} = 1.054$, $\beta_{128} = 1.046$, and $\beta_\infty = 1.039$.

As stated in the HyperLogLog paper [11], the functions δ_1 and δ_2 represent oscillating functions of a tiny amplitude, and they can be safely neglected for all practical purposes.

6.1 Estimation Bias

Given an arbitrary flow f, we know from Section 4 that n_s is the sum of the flow cardinality n_f and a noise random variable $n_s - n_f$ with a binomial distribution of $Bino(n - n_f, \frac{s}{m})$. For $\forall i \in [0, n - n_f]$, we have

$$Prob\{n_s - n_f = i\} = \binom{n - n_f}{i}(\frac{s}{m})^i(1 - \frac{s}{m})^{n-n_f-i}. \quad (11)$$

Under the condition of $n_s - n_f = i$, by Theorem 1, we have

$$\begin{aligned} E(\hat{n_s} \,|\, n_s - n_f = i) &= (n_f + i)\big(1 + \delta_1(n_f + i) + o(1)\big) \\ &\approx n_f + i, \end{aligned} \quad (12)$$

with a small error bounded by a ratio of 5×10^{-5}. Hence,

$$\begin{aligned} E(\hat{n_s}) &= \sum_{i=0}^{n-n_f} E(\hat{n_s} \,|\, n_s - n_f = i) \times Prob\{n_s - n_f = i\} \\ &\approx \sum_{i=0}^{n-n_f} (n_f + i) \times \binom{n - n_f}{i}(\frac{s}{m})^i(1 - \frac{s}{m})^{n-n_f-i} \\ &= n_f + (n - n_f)\frac{s}{m}. \end{aligned} \quad (13)$$

The value of $\hat{n}$ is estimated based on the entire array M or through a separate estimator with hundreds of bytes (i.e., much more than 16 registers). From Theorem 1, we have $E(\hat{n}) = n(1 + \delta_1(n) + o(1)) \approx n$, with a very small error bounded by a ratio of 5×10^{-5}. From the estimation formula (5), we have

$$\begin{aligned} E(\hat{n_f}) &= \frac{ms}{m - s}\Big(\frac{E(\hat{n_s})}{s} - \frac{E(\hat{n})}{m}\Big) \\ &\approx \frac{ms}{m - s}\Big(\frac{n_f + (n - n_f)\frac{s}{m}}{s} - \frac{n}{m}\Big) = n_f. \end{aligned} \quad (14)$$

Therefore, the vHLL estimator is approximately unbiased.

6.2 Estimation Variance

Next we derive the variance of $\hat{n_f}$.

$$\begin{aligned} Var(\hat{n_f}) &= (\frac{ms}{m - s})^2\Big(\frac{Var(\hat{n_s})}{s^2} + \frac{Var(\hat{n})}{m^2}\Big) \quad (15) \\ &= (\frac{ms}{m - s})^2\Big(\frac{E(\hat{n_s}^2) - \big(E(\hat{n_s})\big)^2}{s^2} + \frac{Var(\hat{n})}{m^2}\Big) \\ &= (\frac{m}{m - s})^2\Big(E(\hat{n_s}^2) - \big(E(\hat{n_s})\big)^2 + (\frac{s}{m})^2 Var(\hat{n})\Big) \end{aligned}$$

With $\forall i \in [0, n - n_f)$, under the condition of $n_s - n_f = i$, by Theorem 1, we have

$$\begin{aligned} \tfrac{1}{n_f + i}\sqrt{Var(\hat{n_s} \,|\, n_s - n_f = i)} &= \tfrac{\beta_s}{\sqrt{s}} + \delta_2(n_f + i) + o(1) \\ &= \frac{\beta_s}{\sqrt{s}} \approx \frac{1.04}{\sqrt{s}}, \end{aligned} \quad (16)$$

where we use 1.04 to approximate β_s, assuming $s \geq 128$, which is always the case in our experiments later. Hence,

$$Var(\hat{n_s} \,|\, n_s - n_f = i) \approx \frac{1.04^2}{s}(n_f + i)^2. \quad (17)$$

Similarly, we have

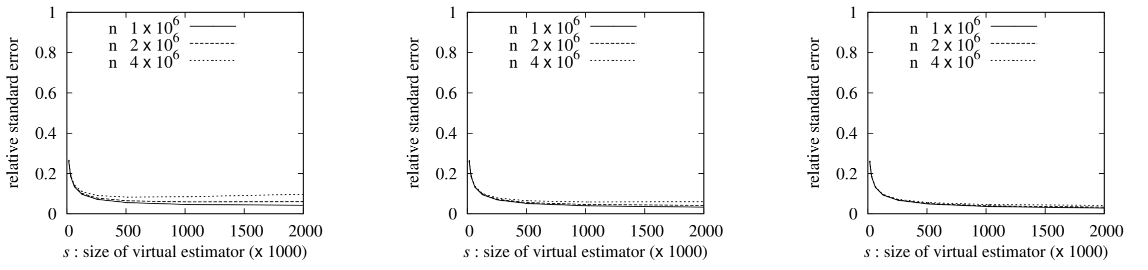
$$Var(\hat{n}) \approx \frac{1.04^2}{m} n^2, \quad (18)$$

where m is the number of registers in the physical estimator for n, and we let $m \geq 128$. Because $E(\hat{n_s}^2 \,|\, n_s = n_f + i) = Var(\hat{n_s} \,|\, n_s - n_f = i) + \big(E(\hat{n_s} \,|\, n_s - n_f = i)\big)^2$, from (12) and (17), when s is sufficiently large, we have

$$\begin{aligned} E(\hat{n_s}^2 \mid n_s = n_f + i) &\approx \frac{1.04^2(n_f + i)^2}{s} + (n_f + i)^2 \\ &= (\frac{1.04^2}{s} + 1)(n_f + i)^2. \end{aligned}$$

Combining (11) with the above equation, we have

$$\begin{aligned} E(\hat{n_s}^2) &= \sum_{i=0}^{n-n_f} E(\hat{n_s}^2 \,|\, n_s - n_f = i) Prob\{n_s - n_f = i\} \quad (19) \\ &\approx \sum_{i=0}^{n-n_f} (\frac{1.04^2}{s} + 1)(n_f + i)^2 \binom{n - n_f}{i}(\frac{s}{m})^i(1 - \frac{s}{m})^{n-n_f-i} \\ &= (\frac{1.04^2}{s} + 1)\big(n_f^2 + 2n_f E(n_s - n_f) + E((n_s - n_f)^2)\big) \\ &= (\frac{1.04^2}{s} + 1)\Big(n_f^2 + 2n_f(n - n_f)\frac{s}{m} \\ &\qquad + (n - n_f)\frac{s}{m}(1 - \frac{s}{m}) + (n - n_f)^2(\frac{s}{m})^2\Big) \\ &= (\frac{1.04^2}{s} + 1)\Big((n_f + (n - n_f)\frac{s}{m})^2 + (n - n_f)\frac{s}{m}(1 - \frac{s}{m})\Big). \end{aligned}$$



(a) Flow cardinality n_f=1 × 10^4 (b) Flow cardinality n_f=2 × 10^4 (c) Flow cardinality n_f=4 × 10^4

Figure 8: Relative standard error with respect to s, n, and n_f

Applying (13), (18) and (19) to (15), we have

$$Var(\hat{n_f}) \approx (\frac{m}{m-s})^2 \Big(\frac{1.04^2}{s}(n_f + (n-n_f)\frac{s}{m})^2 + (n-n_f)\frac{s}{m}(1-\frac{s}{m}) + (\frac{s}{m})^2\frac{1.04^2}{m}n^2\Big). \quad (20)$$

Consider the three terms between the parentheses after $\left(\frac{m}{m-s}\right)^2$. We know that the noise $n_s - n_f$ follows a binomial distribution $Bino(n-n_f, \frac{s}{m})$, whose mean is given by (3) as $(n-n_f)\frac{s}{m}$. Hence, the first term is the estimation variance for the flow cardinality plus the mean noise. The noise variance is $(n-n_f)\frac{s}{m}(1-\frac{s}{m})$, which is captured by the second term. The third term $(\frac{s}{m})^2\frac{1.04^2}{m}n^2$ is caused by the estimation $\hat{n}$ for the grand flow.

6.3 Relative Standard Error

We define the relative standard error as

$$StdErr(\frac{\hat{n_f}}{n_f}) = \frac{\sqrt{Var(\hat{n_f})}}{n_f}. \quad (21)$$

From (20) and (21), we also observe that the relative standard error (or error in short) increases as the cardinality of grand flow n increases, and it decreases as the cardinality of target flow n_f grows.

Below we use some numerical examples to illustrate the above observations and the interplay between different sources of estimation error. Suppose the allocated memory is $m = 256K$. Consider a target flow cardinality of $n_f = 10^4$. Figure 8(a) shows the numerically computed estimation error by (21) with respect to s (the number of registers per virtual estimator) along the horizontal axis and n (the combined cardinality of all flows) for different curves. Starting from 16, as s increases, the error drops quickly, thanks to improved estimation accuracy from the virtual estimator M_f as predicted by Theorem 1. However, when s becomes further larger (more than 256 in the figure), the rate of improvement drops significantly, which can also be predicted by Theorem 1 with its factor of improvement being $\frac{1}{\sqrt{s}}$. Moreover, as s increases, the error caused by noise increases. Combining these two factors, we observe that when s is relatively large (for a wide range from 500 to 2000 in the figure), its impact on the error becomes more or less stabilized.

From Figure 8(a) to Figure 8(c), we increase n_f and observe that the error decreases, which means that the *relative* standard error is smaller for flows of larger cardinalities (although their *absolute* errors can still be larger). When n increases, the error increases, as predicted.

7. EXPERIMENTAL EVALUATION

We have implemented the vHLL solution and the most related work PMC [17]. vHLL is based on register-sharing, while PMC is based on bit-level sharing. We compare their performance through experiments using real network traces downloaded from CAIDA [1]. The traces are captured by a high-speed monitor named equinix-sanjose (located in San Jose, CA, US), which is connected to a 10-Gbit/s Ethernet backbone link. Each trace file captures the packets in 1 minute. In order to create larger traces for our experiments, we download 60 traces and combine them into 6 larger ones, each for 10 consecutive minutes. The statistics of the large traces can be found in Table 3.

Table 3: Trace Statistics

time(min)	num of flows	total cardinality	mean flow cardinality
1-10	1473306	2675506	1.8
11-20	1013517	1856676	1.8
21-30	1648779	3005649	1.8
31-40	1562288	2881330	1.8
41-50	1612709	3280242	2.0
51-60	1612605	3280138	2.0

We consider per-source flows and measure the number of distinct destinations that each source sends packets to. The distribution of the flows with respect to the cardinality has been shown previously in Figure 5. We stress that the purpose of our experiments is primarily technical — evaluating how accurate our vHLL is on cardinality estimation, while the case study of measuring per-source flows may find use in profiling scanners, identifying popular hosts on the Internet (server sources send data to a large number of clients), and detecting anomaly based on measurement over consecutive periods, such as the detection of a worm-infected host by observing that it suddenly deviates from normal behavior by probing a large number of different destination addresses [6,27].

7.1 Estimation Accuracy in Tight Memory

We evaluate the impact of memory space on the accuracy of cardinality estimation for vHLL and PMC. To make a fair comparison between the two, they are allocated with the same memory to process the CAIDA traces. For the proposed vHLL, we configure the value of s to 512 by default, but will vary its value in later experiments. Recall that m is the total number of registers in the common pool. Its value depends on the overall available memory. The average number of flows in all six traces is about 1.5 millions. We vary the available memory space from 1.5 Mb to 0.75 Mb to 0.375 Mb to 0.15 Mb, such that the average memory per flow is about 1 bit, 0.5 bit, 0.25 bit, and 0.1 bit, respectively. The corresponding experimental results are presented in Figures 9, 10, 11, and 12, respectively. Again, each flow is represented by a point, whose x-coordinate is the true cardinality and y-coordinate is the estimated cardinality. The equality line is also shown. The closer a point is to the line, the more accurate the estimation is.

In Figure 9, plot (a) shows the performance of vHLL with average memory of 1 bit per flow. The points are clustered around the equality line ($y = x$), indicating good accuracy. Plot (b) shows the performance of PMC with the points scattering away from the equality line. Plot (c) compares the two solutions in terms of estimation bias. The vertical axis is the relative bias defined as $E(\frac{n_f - \hat{n_f}}{n_f})$. Since there are too few flows for some cardinalities (especially the large ones) in our Internet trace, we divide the horizontal axis into measurement bins of width from 5000 on the high end in the plots to 1000 in the low end to ensure that each bin has a sufficient number of flows 25, and measure the bias and standard deviation in each bin. In general, PMC has larger bias than vHLL. Plot (d) compares the two solutions in terms of accuracy. The vertical axis is the relative standard error of the estimation results, which is defined as $\frac{\sqrt{Var(\hat{n_f})}}{n_f}$. The measurement also uses the bin method as previously explained. vHLL has much smaller error than PMC. This result is expected because according to [17], the performance of PMC is related to the so-called fill rate, i.e., the fraction of bits that are set to ones in the common bit pool. The intended fill rate for PMC to perform well is in the range of $(0, 0.5)$. When the memory is 0.5 bit per flow, the fill rate is about 0.76 in our experiment, which explains why PMC performs relatively poor. Specifically, when the actual cardinality is 10000, 20000 and 30000, the measured errors by PMC are 0.22, 0.28 and 0.23, respectively, while those by vHLL are 0.055, 0.043 and 0.044, respectively.

Figure 10 makes the same set of comparison with 0.5 bit per flow. The performance of vHLL remains good, whereas PMC no longer works as its fill rate becomes 0.9. For example, when the actual cardinality is 10000, 20000 and 30000, the measured errors by PMC are 0.74, 0.67 and 0.87, respectively, while those by vHLL are 0.073, 0.065 and 0.049, respectively.

As the average memory per flow decreases to 0.25 bit and further to 0.1 bit, Figures 11-12 show that vHLL still works with gradually deteriorating accuracy. For 0.25 bit per flow, when the actual cardinality is 10000, 20000 and 30000, the measured errors by vHLL are 0.10, 0.095 and 0.096, respectively. For 0.25 bit per flow, when the actual cardinality is 10000, 20000 and 30000, the measured errors by vHLL are 0.15, 0.13 and 0.10, respectively. We also point out that although the relative standard errors for small flows are higher, it does not entirely diminish the usefulness of these estimations because the absolute errors for small flows are in fact much smaller than those of large ones. For example, by examining the first plot of each figure, one will not mistaken a small flow for a large one due to the modest absolute error.

7.2 Impact of Value s on vHLL

Our second set of experiments evaluate the impact of s (number of registers per virtual estimator) on estimation accuracy. We repeat the experiment in Figure 10(a) with average memory of 0.5 bit per flow, but change s from 512 to values: 128, 256 and 1024. The results are shown in Fig. 13(a)-(c), respectively. Corresponding relative standard errors are shown in Figure 14(a)-(c), respectively.

We observe that when s is relatively small at 128, the estimation accuracy in Figure 13(a) is noticeably worse than that in Figure 10(a), which is evident from the fact that the points of the former surround the equality line less tightly. Quantitatively, the errors in Figure 14(a) with $s = 128$ are larger than those in Figure 14(d) for vHLL with the default $s = 512$. For example, when the actual cardinality is 20000, the relative standard error under $s = 128$ is 10.9%, while that under $s = 512$ is 6.5%.

However, when s becomes large enough (more than 256), for a wide range of values, the impact of s on the estimation accuracy stabilizes, which is evident when comparing Figure 13(b), Figure 13(c), and Figure 10(a), whose s values are 256, 512 and 1024, respectively. For example, when the actual cardinality is 20000, their errors are 8.1%, 6.5% and 5.2%, based on from Figure 14(b), Figure 14(c) and Figure 10(d), respectively.

The above observations are consistent with our analysis in Section 6 and the numerical results in Figure 8 (which has different parameters though). The reasons for these observations have been explained in Section 6.3 and will not be repeated here.

7.3 Impact of Overall Traffic

Our third set of experiments investigate how the overall traffic volume affects estimation accuracy. The overall traffic volume is characterized by n, the sum of all flows' cardinalities, because duplicates in the traffic must be removed in our context. The greater the value of n is, the larger the average noise level on each register will be, which will in turn negatively affect the estimation accuracy of a virtual estimator consisting of s registers.

We artificially increase the cardinality of each flow by a factor randomly chosen from the range of $[1, 3]$, which doubles the cardinality on average. The value of n is thus expected to be doubled. We then repeat the experiment in Figure 10(a) with average memory of 0.5 bit per flow. The results are presented in Figure 15, where plot (a) shows raw estimated cardinalities, plot (b) shows the estimation bias, and plot (c) shows the relative standard error. The bias remains close to zero, particularly for large flows. The error is modest, but larger than that in Figure 10(d) where the value of n is half, which confirms our prediction above.

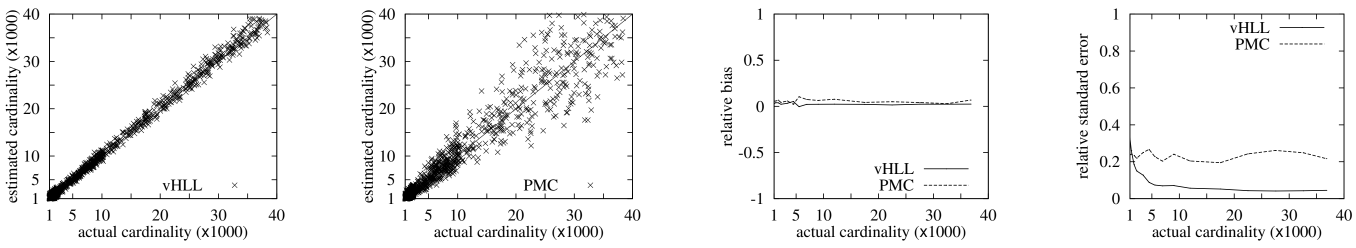
We further enlarge n by increasing the cardinality of each flow with a factor randomly chosen from the range of $[1, 7]$. The value of n is expected to be increased by four folds. The results are presented in Figure 16. Again the bias is close to one, but the error increases.

7.4 A Case Study: Detect Super Destinations

Our last set of experiments compare vHLL and PMC based on a hypothetical application for detecting so-called super destinations. In this case study, we consider per-destination flows and measure the number of distinct sources that access a destination address in each measurement period, using the same Internet traces. Suppose the policy is to report all the destinations that have been accessed by 5,000 or more sources within a measurement period. These *super destinations* may be used for profiling the popular servers (or services) in the network or triggering anomaly warnings (such as potential DDoS attacks) if they were never reported as super destinations before.

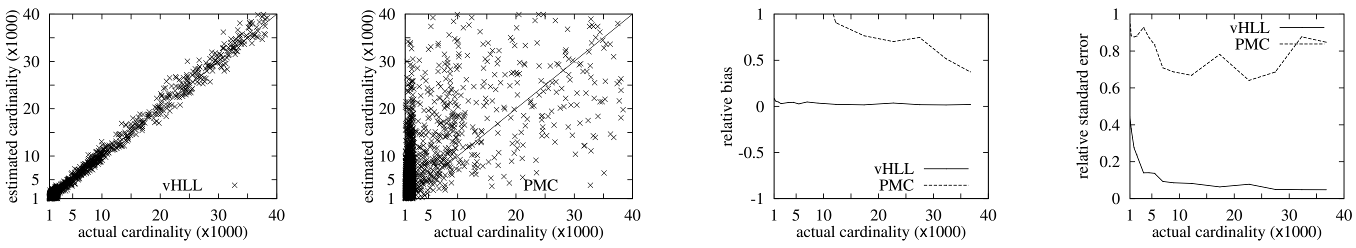
If a destination with a cardinality less than 5,000 is reported, it is called a *false positive*. If a destination with a cardinality 5,000 or above is not reported, it is called a *false negative*. We define the false positive ratio (FPR) as the number of false positives divided by the total number of destinations reported. Based on this definition, if FRP is 0.1, it means 10% of the reported destinations should not have been reported. We define the false negative ratio (FNR) as the number of false negatives divided by the number of destinations whose cardinalities are 5,000 or more.

The experimental results are shown in Table 4. Clearly, vHLL outperforms PMC by a wide margin when we take both FPR and FNR into consideration. The FNR is close to zero for PMC when the memory is 0.5 bit per flow or less. That is because PMC becomes a positively biased estimator in such a small memory as depicted in Fig. 10(b). Its FPR is 73.7% for 0.5 bit per flow and 99.2% for 0.25 bit per flow.



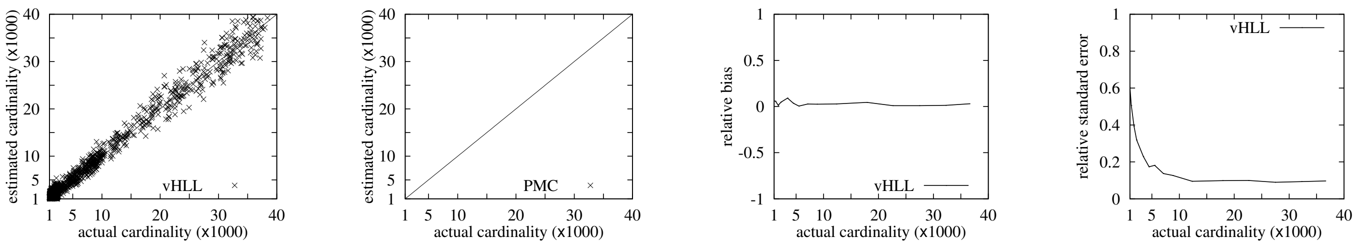
(a) vHLL with 1 bit per flow (b) PMC with 1 bit per flow (c) Estimation bias (d) Estimation accuracy

Figure 9: Compare vHLL and PMC with 1 bit per flow.



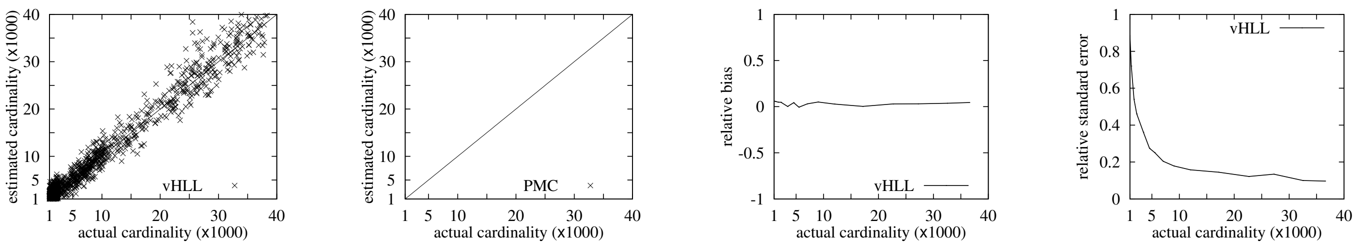
(a) vHLL with 0.5 bit per flow (b) PMC with 0.5 bit per flow (c) Estimation bias (d) Estimation accuracy

Figure 10: Compare vHLL and PMC with 0.5 bit per flow.



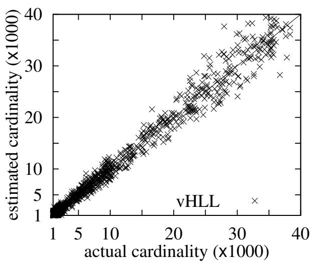
(a) vHLL with 0.25 bit per flow (b) PMC with 0.25 bit per flow (c) Estimation bias (d) Estimation accuracy

Figure 11: Compare vHLL and PMC with 0.25 bit per flow

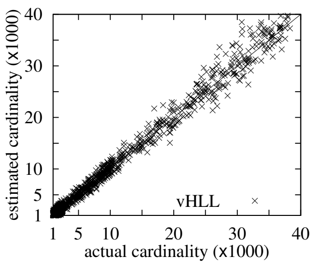


(a) vHLL with 0.1 bit per flow (b) PMC with 0.1 bit per flow (c) Estimation bias (d) Estimation accuracy

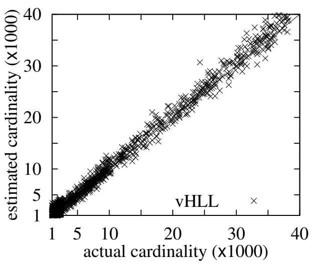
Figure 12: Compare vHLL and PMC with 0.1 bit per flow.



(a) $s = 128$.

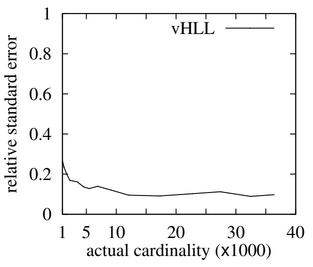


(b) $s = 256$

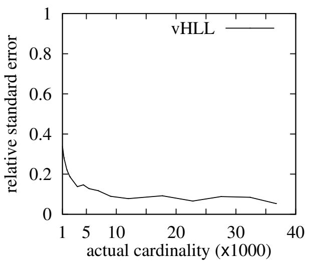


(c) $s = 1024$

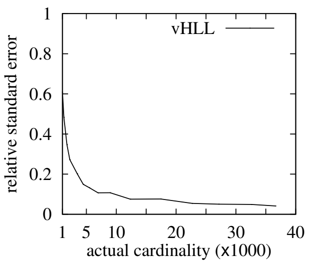
Figure 13: Cardinality estimation with different values of s under average memory of 0.5 bit per flow



(a) $s = 128$



(b) $s = 256$



(c) $s = 1024$

Figure 14: Relative standard errors of cardinality estimation with different values of s under average memory of 0.5 bit per flow

Table 4: False positive ratio and false negative ratio with respect to memory cost

Memory (bit per flow)	PMC		vHLL	
	FPR	FNR	FPR	FNR
0.25	0.992	0.048	0.039	0.026
0.5	0.737	0.045	0.034	0.013
1	0.039	0.044	0.012	0.014

Table 5: $\epsilon = 10\%$, False positive ratio and false negative ratio with respect to memory cost

Memory (bit per flow)	PMC		vHLL	
	FPR	FNR	FPR	FNR
0.25	0.992	0.010	0.014	0.010
0.5	0.846	0.029	0.003	0.003
1	0.013	0.017	0.003	0.002

Table 6: $\epsilon = 20\%$, False positive ratio and false negative ratio with respect to memory cost

Memory (bit per flow)	PMC		vHLL	
	FPR	FNR	FPR	FNR
0.25	0.991	0.003	0.007	0.006
0.5	0.953	0.021	0.0	0.0
1	0.010	0.002	0.0	0.0

vHLL also has non-negligible FPR and FNR since its estimated cardinality is not exactly the true cardinality. To confine impreciseness to a certain degree, the policy may be relaxed to report all destinations whose estimated cardinalities are $5000 \times (1 - \epsilon)$ or above, where $0 \leq \epsilon < 1$. If a destination less than $5000 \times (1 - 2\epsilon)$ gets reported, it is called an ϵ-*false positive*. If a destination with a true cardinality 5,000 or more is not reported, it is called an ϵ-*false negative*. The FPR and FNR are defined the same as before. The experimental results for $\epsilon = 10\%$ are shown in Table 5, and those for $\epsilon = 20\%$ are shown in Table 6, where the FPR and FNR for vHLL are merely 0.7% and 0.6%, respectively, when the memory is 0.25 bit per flow. In Table 6, when the memory grows to at least 0.5 bit per flow, FPR and FNR for vHLL become zeros.

8. CONCLUSION

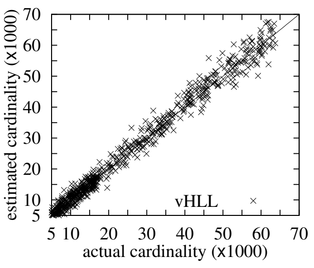
In this paper, we have proposed a unified framework for developing efficient solutions to the problem of estimating cardinalities for a very large number of streaming flows. From this framework, we examine a particularly powerful solution called virtual HyperLogLog (vHLL) in details. Through analysis and experimental evaluation, we show that vHLL can use a compact memory space (down to 0.1 bit per flow on average) to estimate the cardinalities of flows with wide range and reasonable accuracy. This new capability enables on-chip implementation of cardinality estimation needed for online applications that can keep up with the line speed of modern routers, or allow efficient processing of big data by using low-cost commodity computers instead of expensive high-performance computing systems.

9. ACKNOWLEDGMENTS

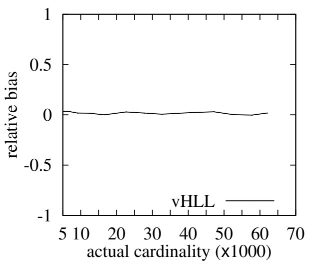
This work is supported in part by National Science Foundation of United States under grants CNS-1115548 and CNS-1409797. The authors also would like to thank the Key Laboratory of Computer Network and Information Integration (Southeast University), Education Ministry of China, for financial support.

10. REFERENCES

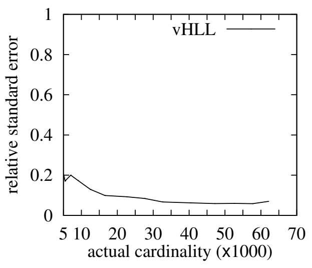
[1] CAIDA UCSD anonymized 2013 internet traces on Jan. 17. http://www.caida.org/data/passive/passive_2013_dataset.xml.
[2] Google trends. *http://www.google.com/trends/*.
[3] Z. Bar-yossef, T. S. Jayram, R. Kumar, D. Sivakumar, L. Trevisan, and Luca. Counting distinct elements in a data stream. *Proc. of RANDOM: Workshop on Randomization and Approximation*, 2002.



(a) n is doubled

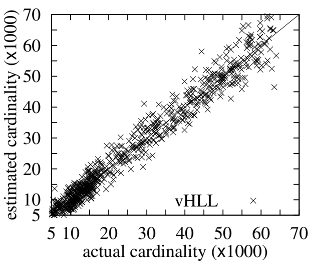


(b) Estimation bias

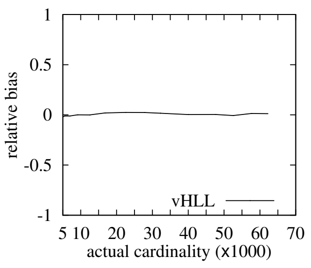


(c) Estimation accuracy

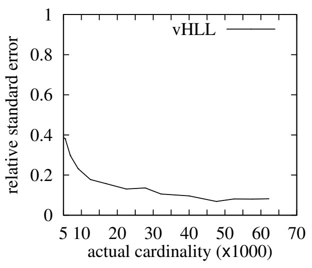
Figure 15: Cardinality estimation with n doubled under average memory of 0.5 bits per flow



(a) n is increased by four folds



(b) Estimation bias



(c) Estimation accuracy

Figure 16: Cardinality estimation with n increased four folds under average memory of 0.5 bits per flow

[4] K. Beyer, P. J. Haas, B. Reinwald, Y. Sismanis, and R. Gemulla. On synopses for distinct-value estimation under multiset operations. *Proc. of ACM SIGMOD*, 2007.

[5] G. Cormode and S. Muthukrishnan. An improved data stream summary: the Count-Min sketch and its applications. *Proc. of LATIN*, 2004.

[6] M. Costa, J. Crowcroft, M. Castro, A. Rowstron, L. Zhou, L. Zhang, and P. Barham. Vigilante: End-to-end containment of internet worms. *SIGOPS Operating Systems Review*, 39(5), October 2005.

[7] X. Dimitropoulos, P. Hurley, and A. Kind. Probabilistic lossy counting: An efficient algorithm for finding heavy hitters. *ACM SIGCOMM Computer Communication Review*, 38(1), 2008.

[8] M. Durand and P. Flajolet. Loglog counting of large cardinalities. *ESA: European Symposia on Algorithms*, pages 605–617, 2003.

[9] C. Estan and G. Varghese. New directions in traffic measurement and accounting. *Proc. of ACM SIGCOMM*, August 2002.

[10] C. Estan, G. Varghese, and M. Fish. Bitmap algorithms for counting active flows on high-speed links. *IEEE/ACM Transactions on Networking (TON)*, 14(5):925–937, 2006.

[11] P. Flajolet, E. Fusy, O. Gandouet, and F. Meunier. HyperLogLog: The analysis of a near-optimal cardinality estimation algorithm. *Proc. of AOFA: International Conference on Analysis Of Algorithms*, 2007.

[12] P. Flajolet and G. N. Martin. Probabilistic counting algorithms for database applications. *J. Comput. Syst. Sci.*, 31(2), 1985.

[13] W. D. Gardner. Researchers transmit optical data at 16.4 Tbps. *InformationWeek*, February 2008.

[14] S. Heule, M. Nunkesser, and A. Hall. HyperLogLog in practice: Algorithmic engineering of a state-of-the-art cardinality estimation algorithm. *Proc. of EDBT*, 2013.

[15] T. Li, S. Chen, and Y. Ling. Fast and compact per-flow traffic measurement through randomized counter sharing. *in Proc. of IEEE INFOCOM*, 2011.

[16] T. Li, S. Chen, W. Luo, M. Zhang, and Y. Qiao. Spreader classification based on optimal dynamic bit sharing. *IEEE/ACM Transactions on Networking*, 21(3):817–830, 2013.

[17] P. Lieven and B. Scheuermann. High-speed per-flow traffic measurement with probabilistic multiplicity counting. *Proc. of IEEE INFOCOM*, pages 1–9, 2010.

[18] Y. Lu, A. Montanari, B. Prabhakar, S. Dharmapurikar, and A. Kabbani. Counter braids: A novel counter architecture for per-flow measurement. *Proc. of ACM SIGMETRICS*, June 2008.

[19] Y. Lu and B. Prabhakar. Robust counting via counter braids: An error-resilient network measurement architecture. *Proc. of IEEE INFOCOM*, April 2009.

[20] Neustar.biz. How to choose a good hash function: Part 3. http://research.neustar.biz/2012/02/02/choosing-a-good-hash-function-part-3.

[21] N. Ntarmos, P. Triantafillou, and G. Weikum. Counting at large: Efficient cardinality estimation in internet-scale data networks. *Proc. of ICDE*, pages 40–40, 2006.

[22] K.-Y. Whang, B. T. Vander-Zanden, and H. M. Taylor. A linear-time probabilistic counting algorithm for database applications. *ACM Transactions on Database Systems*, 15(2):208–229, 1990.

[23] Q. Xiao, Y. Qiao, M. Zhen, and S. Chen. Estimating the persistent spreads in high-speed networks. *Proc. of IEEE ICNP*, pages 131–142, 2014.

[24] Q. Xiao, B. Xiao, and S. Chen. Differential estimation in dynamic RFID systems. In *Proc. of INFOCOM (mini-conference)*, pages 295–299, 2013.

[25] M. Yoon, T. Li, S. Chen, and J.-K. Peir. Fit a spread estimator in small memory. *Proc. of IEEE INFOCOM*, 2009.

[26] Q. Zhao, J. Xu, and A. Kumar. Detection of super sources and destinations in high-speed networks: Algorithms, analysis and evaluation. *IEEE JASC*, 24(10):1840–1852, 2006.

[27] C. C. Zou, L. Gao, W. Gong, and D. Towsley. Monitoring and early warning for internet worms. *Proc. of the 10th ACM Conference on Computer and Communications Security*, 2003.