Wireless Networks

Shigang Chen
Min Chen
Qingjun Xiao

Traffic Measurement for Big Network Data

Springer

Shigang Chen • Min Chen • Qingjun Xiao

Traffic Measurement for Big Network Data

Springer

Shigang Chen
Department of Computer &
Information Science
University of Florida
Gainesville, FL, USA

Min Chen
Department of Computer &
Information Science
University of Florida
Gainesville, FL, USA

Qingjun Xiao
School of Computer Science and
Engineering
Southeast University of China
Nanjing, Jiangsu, China

This work is supported in part by the National Science Foundation under grants CNS-1409797 and STC-1562485.

ISSN 2366-1186 ISSN 2366-1445 (electronic)
Wireless Networks
ISBN 978-3-319-47339-0 ISBN 978-3-319-47340-6 (eBook)
DOI 10.1007/978-3-319-47340-6

Library of Congress Control Number: 2016954314

Printed on acid-free paper

This Springer imprint is published by Springer Nature
The registered company is Springer International Publishing AG
The registered company address is: Gewerbestrasse 11, 6330 Cham, Switzerland

Contents

Chapter 1
Introduction

1.1 Big Network Data

There is hardly any other data set whose size can rival the big network data that flows on the Internet. The annual global IP traffic is expected to pass zettabyte by 2016 [6], which is a billion terabytes. High-speed routers can now forward network traffic at hundreds of terabits per second (Cisco CRS-3). In large enterprise networks, traffic records (such as NetFlow) and logs from routers, switches, intrusion detection systems, and firewalls overwhelm storage space as they are continuously produced; typically, such traffic records are kept only for a limited time frame before being deleted to free space for new information. Big data also happens at the network edge. For a few examples, Google handles over 40,000 search queries every second [27], 500 million tweets are produced per day [32], and major online retailers such as Alibaba process over a billion sales annually. As these data accumulate day by day and year by year, mining them for knowledge becomes a daunting task that requires tremendous resources. This book aims to develop new compact and fast online measurement methods that reduce big network data to measurement summaries orders-of-magnitude smaller than what the traditional methods can do. The new methods hold the promise of allowing routers to perform measurement on large network traffic in real time using small cache memory on network processors, allowing enterprise systems to store their traffic records (in the form of summaries) over a far longer time frame, and allowing users with ordinary computing resources to perform analysis on big network data.

S. Chen et al., *Traffic Measurement for Big Network Data*, Wireless Networks,
DOI 10.1007/978-3-319-47340-6_1

1.2 Online Challenge

Modern routers forward packets from incoming ports to outgoing ports via switching fabric. To process packets in real time, online modules for traffic measurement, packet scheduling, access control, and quality of service are implemented on network processors, bypassing main memory, and CPU almost entirely [20, 24, 28, 37]. Commonly used cache memory on network processor chips is SRAM, typically a few megabytes. Increasing on-chip memory to more than 10 MB is technically feasible, but it comes with a much higher price tag and access time is longer. There is a huge incentive to keep on-chip memory small because smaller memory can be made faster and cheaper. Off-chip SRAM or embedded DRAM (built on 3-D stacking interconnect or packaging on the same module) can be made larger. However, it is slower to access, and the bandwidth between a network processor and its off-chip memory can be a performance bottleneck. Hence, on-chip memory remains the first choice for online network functions that are designed to match the line speed.

To make the matter more challenging, limited on-chip memory may have to be shared among routing/performance/measurement/security functions that are implemented on the same chip. Each function can only use a fraction of the available space. Depending on their relative importance, some functions may be allocated tiny portions of the on-chip memory, whereas the amount of data they have to process and store can be extremely large in high-speed networks. The great disparity in memory demand and supply requires us to implement online functions, including real-time traffic measurement, as compact as possible. As an example, if the amount of on-chip memory allocated to a traffic measurement function is 1 Mb but there are 1M concurrent flows, with 1 bit per flow, can we still perform per-flow traffic measurement? How about 10M concurrent flows with the same memory allocation? This is what we want to achieve through this book.

1.3 Offline Challenge

The space problem also exists offline where disks are used to store network traffic data over time for long-term analysis. As such data are constantly produced, there is a limit on how long they can be stored. With a given amount of disk space, the smaller we can reduce the traffic data, the longer we can keep the data before it has to be removed.

The space issue also arises when we analyze big data. Suppose an analyst who has access to web search records wants to profile the number of searches for each keyword/phrase/question/sentence. This information is useful to online social/economical/opinion trend studies [12]. The profiling may require billions of counters, and more sophisticated data structures will be needed if the analyst wants to remove duplicate searches by the same user—this is called cardinality measurement (or estimation) as we will explain shortly. According to [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, which presents a challenge in 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.

As another example, let's consider an analyst with access to billions of sale records from an online retailer. Suppose she wants to analyze purchase associations. Each association is defined as the purchase of one product followed by the purchase of another product from the same client. Profiling the frequency of each association helps the retailer follow up with product recommendations to its clients after they make purchases. However, such analysis requires pairing up the sale records. The multiplicative effort of pairing may result in an extraordinary number of purchase associations, much larger than the number of sale records. Although the analyst may resort to a datacenter for needed resources, it would certainly be welcome if we can make the same job doable on a regular laptop, even when the number of available memory bits on the laptop is far fewer than the number of purchase associations. (The same is true for the previous example of profiling search record.) This is what we want to achieve through this project.

1.4 Fundamental Primitives

In this book, we model network data as a set of flows, each of which is the abstraction of a data subset defined based on the measurement requirement. For example, we may treat all packets from the same source address as a flow, i.e., per-source flow. In this case, the *flow identifier* is the source address in the packet header. Similarly, we may define per-destination flows, per-source/destination flows, TCP flows, WWW flows, P2P flows, or other application-specific flows. We also need to define *elements* in the flows to be measured. Depending on the application needs, the elements may be destination addresses, source addresses, ports, or even keywords that appear in the packets of a flow.

Big network data consists of millions or even billions of flows. We may measure the *flow size*—which is what NetFlow [3] does—in number of bytes or packets; here, each byte (or packet) is considered as an element to be counted. We may measure the *flow cardinality*—which is what firewalls often do—in number of *distinct* elements in each flow. This is a harder problem because in order to remove the duplicate elements in the flow, we need a way to remember which elements we have seen in the past. Or we may measure the *persistent spread* of a flow: For a certain number of consecutive periods, if an element of a flow appears in each period, we call it a persistent element. The persistent spread of the flow over a given number of periods is defined as the distinct number of persistent elements in the flow. This book presents three important fundamental online functions: per-flow size measurement, persistent spread measurement, and per-flow cardinality measurement.

1.5 Scalable Counter Architectures for Per-Flow Size Measurement

Measuring flow size has many important applications. We may measure the number of packets in each TCP flow, the data rate of each voice-over-IP session, the number of bytes that each host downloads, the number of SYN packets from each source address, or the number of ACK packets sent to each address. Such information is very useful to service provision, capacity planning, accounting and billing, and anomaly detection [13, 16]. For instance, measuring the number of SYN/ACK packets provides a means for detecting SYN attacks [34]. For another example, if we use client addresses as flow identifiers, per-flow size measurement provides each client's traffic volume, which serves as the basis for usage-based billing [20] and graceful service differentiation, where a client's service priority gracefully drops as he over-spends his resource quota. Studying per-flow statistics over consecutive measurement periods may help us discover network access patterns and, together with user profiling, reveal geographic/demographic traffic distributions among users. Such information will assist Internet service providers and application developers to align network resource allocation with the majority's needs [17]. In the event of a botnet attack where there is a sudden surge of small flows, a security administrator may analyze the change in the flow size distribution [7, 15] and use per-flow information to compile the list of candidate bots that contribute to the change, helping to narrow down the scope for further investigation.

In this book, we first present a novel SRAM-only counter architecture called Counter Tree. The contributions are summarized as follows:

1. We design a two-dimensional counter sharing scheme, where each counter can be shared not only by different flows, but also among different virtual counters. Thanks to this scheme, a significant memory save can be achieved, making Counter Tree work well under a tight memory where existing schemes do not work.
2. Counter Tree reserves more significant bits for larger flows, which dramatically extends the estimation range when compared with existing schemes.
3. Counter Tree has a very high processing speed. Encoding a packet only requires a little more than 2 memory accesses on average, which is asymptotically optimal.
4. Counter Tree supports an instantaneous query of the size of an arbitrary flow. Two offline decoding methods are designed to estimate flow sizes. The extensive experiments with real network trace demonstrate both methods can generate accurate results even under extremely tight memory, e.g., 2 bits per flow.

1.6 Hyper-Compact Virtual Estimators for Per-Flow Cardinality Measurement

Measuring flow cardinality also has many applications. Address-scan detection is to measure the number of distinct destination addresses (elements) in each per-source flow. If a source is found to contact too many destinations, it is flagged as a potential scanner. In case of random scanning in worm attacks, such cardinality measurement provides the infection rate of a worm [4, 23, 30]. Similarly, port-scan detection [29] is to measure the number of distinct destination ports in each per-source/destination flow. In another application example, we treat all packets sent to a common destination as a per-destination flow and count the number of distinct source addresses in each flow. If we observe the cardinality of a certain flow suddenly surges, it may signal a DDoS attack [21, 22, 25, 26, 33] 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 [36], where all requests from different users to a common URL form a flow. Flow cardinality can also help identify P2P hosts [2, 9, 33, 37].

For the big data cases at network edge in the introduction, if we treat all search records that query the same phrase as a flow, we may define the cardinality of each flow as the number of distinct source addresses that have performed the search, which suggests the popularity of the phrase and is therefore useful in social/economical/disease trend studies [12]. If we treat each online purchase association as a flow, we may define the flow cardinality as the number of distinct costumers who have purchased the two products in the association, which provides information for targeted advertisement.

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 [10], MultiresolutionBitmap [9] (which is a generalization of LinearCounting [35]), MinCount [1], 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, which is still too much for measuring big network data.

After decades of development [1, 8–11, 35], 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 is to let different estimators (each for one flow) share bits [18, 19, 36], so that bits unused by one can be picked up by another. 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. Moreover, sharing has only been applied to bitmap and PCSA [10], 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. Finally, 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.

1.7 Memory-Efficient Estimators for Persistent Spread Measurement

The traditional *superspreader* detector, which is designed to identify flows with abnormally large spreads, has many applications in monitoring network anomalies [33, 38]. For example, if a spread estimator can measure the number of distinct destinations in each per-source flow, then it can be used to detect network scanners (or infected hosts), which probe a large number of different destinations. Another example is the spread estimator of per-destination flows, which can be applied to detecting the well-known DDoS attack, in which a malicious party uses an army of compromised hosts to overwhelm a destination server.

However, the superspreader detector may fail to discover malicious activities, if the attackers deliberately suppress their traffic volumes and spreads to escape the detection. In these cases, measuring persistent flow cardinality may find its applications. Consider two examples where flow cardinality alone is insufficient. First, scanners can be identified if they send probes to too many destination addresses, i.e., the cardinalities of per-source flows are large. However, a stealthy scanner may intentionally reduce its probing rate to control its flow cardinality in order to evade detection. Even with a reduced probing rate, after enough time, the scanner can discover systems with vulnerabilities to exploit. Measuring persistent flow cardinality can help identify this type of scanners. As a scanner probes different destination addresses over time, its persistent flow cardinality is zero. Therefore, modest flow cardinality but usually low persistent cardinality signals a low-rate scanner that wanders in the destination address space.

In the second example, DDoS attacks may be identified if too many clients send requests to a server, i.e., the cardinality of a per-destination flow is too high. However, with a smaller number of attacking machines, stealthy denial-of-quality attacks do not attempt to overwhelm the target server with excessive requests, but to degrade its performance. If the number of attacking machines is similar to the number of legitimate users, we will not see unusual flow cardinality. In this case, measuring persistent flow cardinality may help. According to our analysis of real-world network traces from CAIDA [31], the continuous interaction between legitimate users and their target servers is normally shorter than 20 min. For stealthy denial-of-quality attacks, since their objective is to degrade the performance of target server over a long period, the attacking machines will send requests persistently to the target server, resulting in a significant persistent cardinality over time that is higher than the usual value.

This book provides an implementation of the persistent spread estimator based on a data structure called *multi-virtual bitmaps*. The size of on-chip SRAM space it requires does not relate with the number of time periods t, but depends on the number of flow elements that pass through a router in just one time period. More precisely, in each time period, its required size of SRAM is less than one bit per-flow element.

Even given such limited space, our algorithm is able to deliver high estimation accuracy. The evaluation results show that our estimator is 90 % more accurate than a continuous variant of Flajolet–Martin sketches [5]. Such an improvement comes from our observation that in real network traffic traces, the continuous interaction of legitimate users with a HTTP/HTTPs server is pretty short in time duration, less than 20 min typically. Hence, it is possible to filter the traffic from the short-term behaviors of legitimate users, and retain the long-term persistent traffic that is suspicious to link with stealthy DDoS attacks or network scanning. Moreover, the estimation accuracy of our algorithm improves when the number of measurement periods t grows, because it is able to filter the short-term traffic of legitimate users more effectively. The accuracy gain as t grows is a useful feature that allows a network administrator to increase t arbitrarily to distinguish persistent elements from normal transient traffic.

Our algorithm provides another advantage that extends the operating range of producing effective measurements by hundreds of times, as compared with the traditional bitmap method, which allocates each flow with an equal-sized and separated bitmap. In contrast, our method allows different flows to share bits from a common memory pool. By drawing bits randomly from the pool, an individual flow constructs a virtual bitmap, for the estimation of its persistent spread. Through bit sharing, large flows can "borrow" bits from small flows to extend their effective operating range. We have evaluated the performance of our algorithm, including memory expense, estimation accuracy, and operating range, by experiments based on real network traffic traces.

1.8 Outline of the Book

The rest of the book is organized as follows. Chapter 2 presents a scalable counter architecture for per-flow size measurement based on two-dimensional counter sharing. In this chapter, we provide a novel tree structure of counters for packet recording/decoding, which mixes per-flow information randomly in a tight SRAM space for compactness. Chapter 3 presents a framework of virtual estimators based on register (multi-bit) level sharing of a common memory space. We can apply the framework to various solutions of cardinality estimation, achieving far better memory efficiency than the best existing work. Chapter 4 presents an efficient scheme for persistent spread measurement based on bit sharing in physical bitmap, which records the elements of all flows during a measurement period to a single bitmap. The bitmaps built in different measurement periods can be combined for estimating the persistent spread of any flow.

References

1. Bar-yossef, Z., Jayram, T.S., Kumar, R., Sivakumar, D., Trevisan, L.: Luca: counting distinct elements in a data stream. In: Proceedings of RANDOM: Workshop on Randomization and Approximation (2002)
2. Cao, J., Jin, Y., Chen, A., Bu, T., Zhang, Z.: Identifying high cardinality internet hosts. In: Proceedings of IEEE INFOCOM (2009)
3. Cisco IOS NetFlow: Available at http://www.cisco.com/c/en/us/products/ios-nx-os-software/ios-netflow/index.html (2005)
4. Chen, S., Tang, Y.: Slowing down internet worms. In: Proceedings of IEEE ICDCS (2004)
5. Chen, A., Cao, J., Bu, T.: A simple and efficient estimation method for stream expression cardinalities. In: Proceedings of the 33rd International Conference on Very Large Data Bases, VLDB '07, pp. 171–182 (2007)
6. Cisco: The Zettabyte EraTrends and Analysis (2015). Available at http://www.cisco.com/c/en/us/solutions/collateral/service-provider/visual-networking-index-vni/VNI_Hyperconnectivity_WP.html
7. Duffield, N., Lund, C., Thorup, M.: Estimating flow distributions from sampled flow statistics. In: Proceedings of ACM SIGCOMM (2003)
8. Durand, M., Flajolet, P.: Loglog counting of large cardinalities. In: ESA: European Symposia on Algorithms, pp. 605–617 (2003)
9. Estan, C., Varghese, G., Fish, M.: Bitmap algorithms for counting active flows on high-speed links. IEEE/ACM Trans. Netw. **14**(5), 925–937 (2006)
10. Flajolet, P., Martin, G.N.: Probabilistic counting algorithms for database applications. J. Comput. Syst. Sci. **31**(2), 182–209 (1985)
11. Flajolet, P., Fusy, E., Gandouet, O., Meunier., F.: HyperLogLog: the analysis of a near-optimal cardinality estimation algorithm. In: Proceedings of AOFA: International Conference on Analysis of Algorithms (2007)
12. Google Trends: Available at https://www.google.com/trends/
13. Hao, F., Kodialam, M., Lakshman, T.V.: ACCEL-RATE: a faster mechanism for memory efficient per-flow traffic estimation. In: Proceedings of ACM SIGMETRICS/Performance (2004)
14. Heule, S., Nunkesser, M., Hall, A.: HyperLogLog in practice: algorithmic engineering of a state-of-the-art cardinality estimation algorithm. In: Proceedings of EDBT (2013)
15. Kumar, A., Sung, M., Xu, J., Wang, J.: Data streaming algorithms for efficient and accurate estimation of flow size distribution. In: Proceedings of ACM SIGMETRICS (2004)
16. Kumar, A., Xu, J., Wang, J., Spatschek, O., Li, L.: Space-code bloom filter for efficient per-flow traffic measurement. In: Proceedings of IEEE INFOCOM (2004). IEEE JSAC **24**(12), 2327–2339 (2006)
17. Li, T., Chen, S., Ling, Y.: Fast and compact per-flow traffic measurement through randomized counter sharing. In: Proceedings of IEEE INFOCOM, pp. 1799–1807 (2011)
18. Li, T., Chen, S., Luo, W., Zhang, M., Qiao, Y.: Spreader classification based on optimal dynamic bit sharing. IEEE/ACM Trans. Netw. **21**(3), 817–830 (2013)
19. Lieven, P., Scheuermann, B.: High-speed per-flow traffic measurement with probabilistic multiplicity counting. In: Proceedings of IEEE INFOCOM, pp. 1–9 (2010). doi:10.1109/INFCOM.2010.5461921
20. Lu, Y., Montanari, A., Prabhakar, B., Dharmapurikar, S., Kabbani, A.: Counter Braids: a novel counter architecture for per-flow measurement. In: Proceedings of ACM SIGMETRICS (2008)
21. Mahajan, P., Bellovin, S.M., Floyd, S., Ioannidis, J., Paxson, V., Shenker, S.: Controlling high bandwidth aggregates in the network. Comput. Commun. Rev. **32**(3), 62–73 (2002)
22. Moore, D., Voelker, G., Savage, S.: Inferring internet denial of service activity. In: Proceedings of USENIX Security Symposium' 2001 (2001)
23. Moore, D., Shannon, C., Voelker, G.M., Savage, S.: Internet quarantine: requirements for containing self-propagating code. In: Proceedings of IEEE INFOCOM (2003)

24. Moshref, M., Yu, M., Govindan, R., Vahdat, A.: Dream: dynamic resource allocation for software-defined measurement. In: Proceedings of ACM SIGCOMM, pp. 419–430 (2014)
25. Park, K., Lee, H.: On the effectiveness of route-based packet filtering for distributed DoS attack prevention in power-law internets. In: Proceedings of ACM SIGCOMM'2001 (2001)
26. Plonka, D.: FlowScan: a network traffic flow reporting and visualization tool. In: Proceedings of USENIX LISA (2000)
27. Smith, G.: By the numbers: 80+ amazing google search statistics and facts (2015). Available at http://expandedramblings.com/index.php/by-the-numbers-a-gigantic-list-of-google-stats-and-facts/
28. Song, H., Hao, F., Kodialam, M., Lakshman, T.: IPv6 lookups using distributed and load balanced bloom filters for 100Gbps core router line cards. In: Proceedings of IEEE INFOCOM (2009)
29. Staniford, S., Hoagland, J., McAlerney, J.: Practical automated detection of stealthy portscans. J. Comput. Secur. **10**, 105–136 (2002)
30. Staniford, S., Paxson, V., Weaver, N.: How to 0wn the internet in your spare time. In: Proceedings of USENIX Security Symposium (2002)
31. The CAIDA UCSD Anonymized 2013 Internet Traces - January 17: http://www.caida.org/data/passive/passive_2013_dataset.xml (2013)
32. Twitter Usage Statistics: Available at http://www.internetlivestats.com/twitter-statistics/ (2013)
33. Venkatataman, S., Song, D., Gibbons, P., Blum, A.: New streaming algorithms for fast detection of superspreaders. In: Proceedings of NDSS (2005)
34. Wang, H., Zhang, D., Shin, K.G.: SYN-dog: sniffing SYN Flooding Sources. In: Proceedings of 22nd International Conference on Distributed Computing Systems (ICDCS'02) (2002)
35. Whang, K.Y., Vander-Zanden, B.T., Taylor, H.M.: A linear-time probabilistic counting algorithm for database applications. ACM Trans. Database Syst. **15**(2), 208–229 (1990)
36. Yoon, M., Li, T., Chen, S., Peir, J.K.: Fit a spread estimator in small memory. In: Proceedings of IEEE INFOCOM (2009)
37. Zhao, Q., Xu, J., Kumar, A.: Detection of super sources and destinations in high-speed networks: algorithms, analysis and evaluation. IEEE JASC **24**(10), 1840–1852 (2006)
38. Zhao, Q., Xu, J., Kumar, A.: Detection of super sources and destinations in high-speed networks: algorithms, analysis and evaluation. IEEE JSAC **24**(10) (2006)

Chapter 2
Per-Flow Size Measurement

Per-flow size measurement, which is to count the number of packets for each active flow during a certain measurement period, has many applications in usage accounting, traffic engineering, service provision and anomaly detection. In order to maintain the high throughput of routers or switchers, the per-flow size measurement module should use high-bandwidth SRAM that allows fast memory accesses. Due to the limited SRAM space, exact counting, which requires to keep a counter for each flow, does not scale to measure big network data consisting of numerous flows. Some recent work takes a different design path to accurately estimate the flow sizes using counter architectures that can fit into tight SRAM. However, existing counter architectures have some limitations, either still requiring considerable SRAM space, or having a very small estimation range. This chapter presents a scalable counter architecture, called Counter Tree, which leverages a two-dimensional counter sharing technique to achieve far better memory efficiency and significantly extend estimation range. Furthermore, we improve the performance of Counter Tree by adding a status bit to each counter. The extensive experiments with real network trace demonstrate that the new architecture can produce accurate estimates for flows of all sizes even under a very tight memory space, e.g., 1 bit per flow.

2.1 Problem Statement

Per-flow size measurement is one of the fundamental problems in network traffic measurement [8, 10–13, 15–17]. It is to count the number of packets (or called flow size) for each active flow during a measurement period, e.g., 1 min or the time for processing 10 million packets. The flows under measurement can be per-source flows, per-destination flows, per-source/destination flows, TCP flows, http flows, or any user-defined logical flows. Each flow is uniquely identified by its flow label—for example, the flow labels for per-source flows are the source addresses.

Three metrics are used to evaluate the performance of different per-flow size measurement schemes:

1. Memory requirement: Due to the constraint of SRAM space, we want to use as small memory as possible for per-flow size measurement. Here we focus on the memory requirement for implementing the counter architectures, while the memory overhead for flow label collection is not our concern. Some memory-efficient approaches [12] for flow label collection can be found in literature.
2. Processing time: To keep up with the line speeds, the processing time for encoding a packet should be small, such that the implementation of the measurement module will not deteriorate throughput. In most counter architectures [8, 10, 13], the processing time for encoding a packet mainly results from the memory accesses and hash computations.
3. Measurement accuracy: Given a particular memory space, the measurement results of flow sizes should be as accurate as possible. Suppose the true size of a flow is s, and the measured size is $\hat{s}$. We use the relative bias $Bias(\frac{\hat{s}}{s})$ and relative standard error $StdErr(\frac{\hat{s}}{s})$ to evaluate the measurement accuracy, which are defined as follows:

$$Bias(\frac{\hat{s}}{s}) = E(\frac{\hat{s}}{s}) - 1, \tag{2.1}$$

$$StdErr(\frac{\hat{s}}{s}) = \sqrt{Var(\frac{\hat{s}}{s})} = \frac{\sqrt{Var(\hat{s})}}{s}. \tag{2.2}$$

Notations used in the chapter are given in Table 2.1 for quick reference.

Table 2.1 Notations

Symbols	Descriptions
s	True size of a flow
$\hat{s}$	Estimated size of a flow
M	Available memory space in bits
b	Number of bits in each physical counter
m	Number of leaf nodes in the Counter Tree
d	Degree of each non-leaf node in the Counter Tree
r	Number of virtual counters for each flow
h	Height of the Counter Tree
$C[i]$	The ith physical counter
$V[i]$	The ith virtual counter
k	Number of leaf nodes in a subtree counter
n	Total number of packets in the measurement period

2.2 Prior Art

With tremendous number of flows under measurement for big network data, it is impossible to keep an individual counter for each flow in SRAM. Therefore, exact counting generally adopts a hybrid SRAM–DRAM architecture [15–17], where small counters in SRAM are incremented at high speed, and occasionally written back to larger counters in DRAM. However, the hybrid architecture incurs very costly SRAM-to-DRAM updates. Furthermore, the flow-to-counter association requires considerable SRAM [13].

To fit the measurement module in tight SRAM, some schemes only give the distribution of flow sizes [3, 7], or measure the sizes of large flows [5, 6]. Some recent work takes a different design path to accurately estimate the flow sizes instead of counting their exact sizes, thereby reducing memory overhead. The state-of-the-art estimation approaches include bitmap-based MSCBF, and counter-based Counter Braids and randomized counter sharing scheme.

The *Multiresolution Space-Code Bloom Filter* (MSCBF) [8] employs multiple Bloom filters to encode packets with different sampling probabilities. Filters with high sampling probabilities can keep track of small flows, while filters with low sampling probabilities can track large flows. However, the bitmap nature of MRSCBF determines that it is not memory efficient for counting [10]. TinyTable [4] is a novel hash table-based data structure that represents multiset membership. It improves the query and update efficiency of Bloom filters. However, for per-flow traffic measurement, it still requires a high memory cost, which is tens of bits per flow [4].

The *Counter Braids* (CB) [12, 13] is a counter architecture for flow size measurement. It avoids the storage of flow-to-counter association by hashing flows to counters on the fly, and it reduces memory requirement by sharing counters among flows. A typical implementation of CB consists of two layers of counters, and employs three hash functions. To encode a packet, it is hashed to three counters based on its flow label, which are all incremented by one. If any of the first-layer counter overflows, another three second-layer counters will be used. Since each counter is shared by multiple flows, it counts all associated flows. Therefore, the counters essentially form a set of linear equations of the flow sizes. A message passing reconstruction algorithm was used to estimate the flow sizes in an iterative way. CB can recover the exact flow sizes when sufficient memory is available, e.g., 10 bits per flow. However, CB has three limitations. First, it performs 6 (occasionally 12) memory accesses to encode one packet. Second, it yields very biased or even meaningless estimates under a tight memory, e.g., less than 4 pits per flow. In fact, we find that the estimation results of CB do not converge even with 8 bits per flow, though it may occasionally produce very accurate results if we manually terminate the process after some iterations. Third, CB does not support instantaneous queries of flow sizes. All flow sizes must be decoded together at the end of a measurement period.

A new data encoding/decoding scheme, called *randomized counter sharing* [10], was designed to further reduce the memory requirement and processing time of per-flow size measurement. The idea is to split each flow among a number of counters (called the *storage vector* of the flow) that are randomly selected from a counter pool. When encoding a packet of a particular flow, it is randomly mapped to a counter of the flow's storage vector, and the counter is then incremented by one. This scheme requires only 2 memory accesses for encoding one packet, achieving the optimal processing speed. Moreover, it can still yield reasonably accurate estimates under a tight memory space where CB no longer works. Two estimation methods CSM and MLM are used to estimate flow sizes. The most serious problem of this scheme is that its estimation range is limited, e.g., a few thousands in a typical implementation. For large flows with sizes beyond the estimation range, the scheme leads to very negatively biased estimates since overflowed counters lose information. In the journal version [11], some approaches were provided to extend the estimation range, which, however, cannot address the issue fundamentally. The first approach is to increase the length of each counter or the size of the storage vector. However, this approach degrades estimation accuracy since fewer counters are available or each counter is shared by more flows. Following a reasonable parameter setting, the estimation range is still very limited. The second approach employs a sampling module. Each arriving packet is sampled with a probability p before being encoded to a counter. Aggressive sampling not only introduces significant error [8], but also fails to measure some small size or even moderate-size flows. For example, if we let $p = 0.001$, flows with sizes less than 1000 are hardly be captured. The final approach resorts to the hybrid SRAM/DRAM design, which requires costly SRAM-to-DRAM updates.

2.3 Design of Counter Tree Architecture

In this section, we provide a novel scalable counter architecture called Counter Tree [1].

2.3.1 Motivation

In spite of the large number of flows in networks, many studies reveal a common observation that a small percentage of large flows account for a high percentage of the traffic (also known as the heavy-tailed distribution). The study in [14] showed that the top 15 % of the destination prefixes (per-destination flows) account for over 95 % of the byte traffic. As an example, we use a network trace obtained from the main gateway of University of Florida, which contains about 68 million TCP flows and 750 million packets. The distribution of flow sizes is illustrated in Fig. 2.1, where each point represents the number (y coordinate) of flows that have a particular

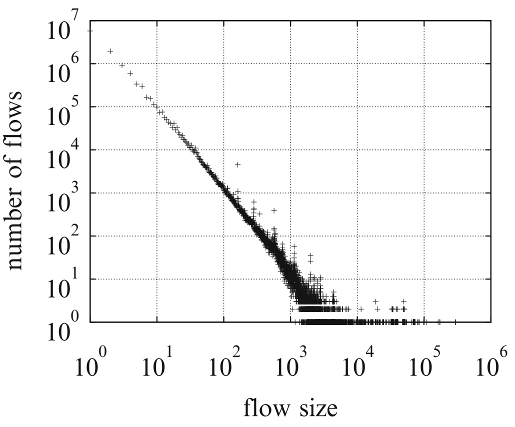


Fig. 2.1 Distribution of flow sizes, where each point represents the number (y coordinate) of flows that have a particular size (x coordinate)

size (x coordinate). This log-scale figure demonstrates that the vast majority of flows have small sizes, while only a small number of flows have large sizes. Without knowing the flow sizes beforehand (which are in fact what we want to measure), the length of counters should be set according to the maximum flow size, which may need to be as large as 64 bits [15]. However, if a flow turns out to be small, e.g., with a size of 1, most of the bits in its counter will be wasted. This observation motivates us to save memory by utilizing the waste.

2.3.2 Two-Dimensional Counter Sharing

To reduce the memory waste caused by small flows, we should enable counter sharing. Therefore, we develop a novel counter sharing technique called two-dimensional counter sharing, which includes horizontal counter sharing and vertical counter sharing. The available memory space is divided into small physical counters, which are logically organized on different layers. Horizontal counter sharing means that any counter on any layer can be shared by different flows, and vertical counter sharing means that each higher-layer counter can be shared by multiple lower-layer counters, acting as their high-order bits.

In horizontal counter sharing, each counter is shared by multiple flows. The rationale is to let large flows borrow memory from small or medium flows that will not fully use their counters. But each counter will have to store the combined size of multiple flows particularly if the number of counters is much smaller than the number of flows. To alleviate this problem, the CountMin approach [2] maps each flow to multiple counters and use the smallest counter value as the flow size. The problems are that each packet of a flow will result in multiple counter updates (which reduces the processing throughput multiple folds) and that the smallest counter may still be the sum of multiple flow sizes, causing positive bias in estimation. Our new design in the next subsection will keep the benefit of counter sharing, while ensuring that approximately one counter is updated per packet and in the meantime avoiding

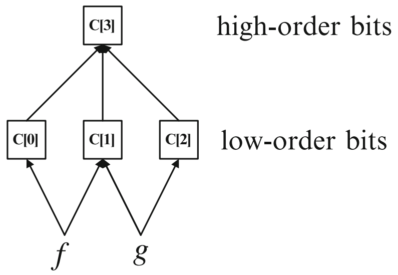


Fig. 2.2 An illustration of two-dimensional countering sharing, where $C[0]$, $C[1]$, $C[2]$, and $C[3]$ are four small physical counters, and f and g are two flows. Each low-order counter can be shared by multiple flows, and each high-order counter can be shared by multiple low-order counters as their high-order bits

the positive bias in estimation. The idea of our design is to split each flow to multiple counters through random mapping, and each counter can therefore be shared by different flows. In the example of Fig. 2.2, the counter $C[1]$ is shared by two flows f and g. If f is a small flow and g is a large flow, g can help make full use of $C[1]$.

Horizontal counter sharing allows some bits wasted by small or medium flows to be used by large flows. However, since small flows account for a dominant percentage of network flows, many bits in counters occupied only by small flows can still be wasted. This observation leads to the idea of *vertical counter sharing* below, which allows the more significant bits (i.e., higher-order bits) to be shared by more flows. As an example, the counter $C[3]$ in Fig. 2.2 serves as extra high-order bits for $C[0]$, $C[1]$, and $C[2]$, and any flows mapped to $C[0]$, $C[1]$, and $C[2]$ can use $C[3]$ when necessary.

We introduce the concept of virtual counters, each of which is the concatenation of multiple small physical counters, which are also called the *component counters* of the virtual counter. Physical counters do not share their bits, but virtual counters share bits by sharing their component counters, particularly those representing more significant bits in virtual counters. In essence, vertical counter sharing implements seamless dynamic memory allocation based on flow sizes, without having to allocate or deallocate physical counters of different sizes on the fly (which is too costly to implement on chip). Suppose we have three physical counters, each with 3 bits. The first counter is allocated to f, the second is for g, while the third is reserved for whoever needs it. As a result, g will use the third counter when the second counter overflows. These three component counters only require 9 bits to successfully record f and g.

The scheme of two-dimensional counter sharing not only contributes to significant memory saving, but it also introduces noise among virtual counters due to space sharing. Fortunately, we can employ statistical tools to remove such noise as we will show shortly. In the sequel, when we use the word “counter" without preceding it with “virtual," we mean a physical (component) counter by default unless the context clearly suggests otherwise.

2.3.3 Counter Tree Architecture

We design a Counter Tree architecture to implement two-dimensional counter sharing. The architecture can be used to record flows of all sizes (small, medium, or large). Given a memory space of M bits, we divide it into small counters, each consisting of b bits. We organize those counters into a tree structure from the bottom up. Let the leaf nodes be the layer 0 which consists of m counters. The degree of each non-leaf node is d. Therefore, the number of counters on a upper layer is $\frac{1}{d}$ of it lower layer. Denote the height of the tree by h, with layers indexed from 0 to $h-1$. We have the following constraint:

$$\sum_{j=0}^{h-1} \frac{m}{d^j} \times b \le M. \tag{2.3}$$

Therefore,

$$m \le \frac{d^{h-1}M}{(d^h-1)b}. \tag{2.4}$$

If the $(h-1)$th layer contains more than one counter, namely $\frac{m}{d^{h-1}} > 1$, we put an extra node as the root of the tree, called *virtual root*. Figure 2.3 gives an example of organizing the 14 counters into a binary tree with three layers, where $m = 8$ and $d = 2$. Starting from a leaf node $C[i]$ $(0 \le i < m)$, the h counters along the path to the root form a virtual counter, denoted as $V[i]$. As a result, there will be m virtual counters in total, denoted as an array V.

Starting from $C[i]$ at layer 0, the counter at layer j $(0 \le j < h)$ that $V[i]$ will include, denoted by $V[i][j]$, is

$$V[i][j] = C[\lfloor \frac{i}{d^j} \rfloor + \sum_{t=1}^{j} \frac{m}{d^{t-1}}]. \tag{2.5}$$

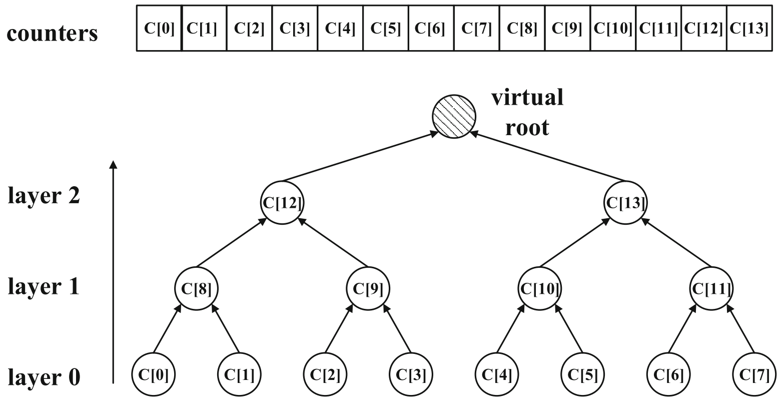


Fig. 2.3 An example of organizing counters into a binary tree

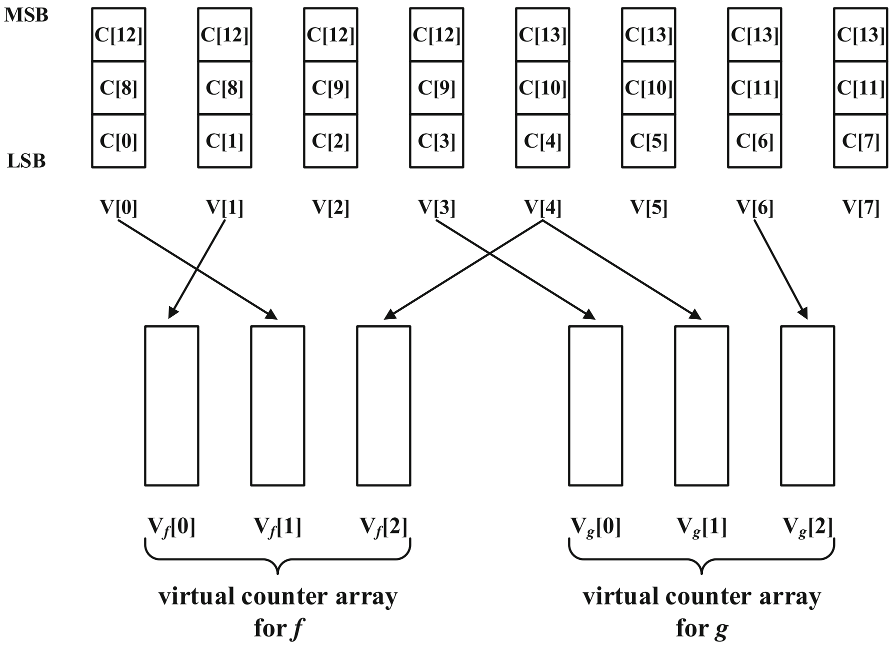


Fig. 2.4 Virtual counters and virtual counter arrays for flows. The virtual counter array for a particular flow consists of multiple counters pseudo-randomly chosen from the counters

Therefore, the Counter Tree in Fig. 2.3 can yield 8 virtual counters as shown in the upper half of Fig. 2.4. We can see that a counter located at a higher layer (which corresponds to more significant bits in a virtual counter) is shared by more virtual counters but will be used only by large flows. This embodies the idea of vertical counter sharing. Later we will show that each virtual counter can be shared by multiple flows, corresponding to the idea of horizontal counter sharing.

We note that multiple layers of counters are also used by Counter Braids [12, 13] under a different design. For Counter Braids, each flow is randomly mapped to u counters at the bottom level, where u may be 3. Any packet of a flow will cause all u counters of the flow to increase by one. Hence, each counter records the total number of packets for all flows that are mapped to it. That is, each counter represents a linear equation on the sizes of some flows (that are mapped to the counter). When there are enough counters, there will be enough linear equations, which we can solve (by the method of message passing in [13]) for the sizes of all flows. But this approach requires updating u counters per packet and it needs a sufficient number of counters. What about the memory space is too tight and thus the number of counters is insufficient? Our method can work under such tight space where Counter Braids no longer work. In Counter Tree, each flow is randomly mapped to r counters at the bottom level, where r should be large, e.g., in hundreds. Any packet of a flow will cause one of the r counters to increase by one. Therefore, each counter no longer represents a linear equation on the sizes of some flows, which also means that the estimation method of Counter Braids cannot be applied here. Instead, we view the r counters of a flow f as a whole, whose sum carries both the size of flow f and the noise from other flows due to counter sharing. We do not know exactly who those other flows are, but nevertheless the noise can be statistically measured and removed.

2.3.4 Counting Range

The counting range of each virtual counter is 2^{bh}. To extend the counting range, one way is to increase b. However, larger b mean fewer counters are available, e.g., if we double b, the number of available counters will be reduced by half. Moreover, if b is set overly large, some bits in each counter will be wasted. Alternatively, we can increase h for the purpose of extending the counting range, which is more memory efficient as we will show shortly. Suppose M' bytes are allocated for layer-0 counters, which translates into $\frac{M'}{b}$ counters. Hence, the height of the Counter Tree can be up to $\log_d \frac{M'}{b} + 1$. Since the number of counters on the jth layers is reduced to $\frac{1}{d}$ of the $(j-1)$th layer, the following memory constraint should hold:

$$\sum_{j=0}^{h-1} \frac{M'}{d^j} \le M.$$

Since $\sum_{j=0}^{h-1} \frac{M'}{d^j} < \frac{d}{d-1}M'$, and we have

$$M' > \frac{d-1}{d}M, \tag{2.6}$$

which means that only $\frac{1}{d}$ of the memory needs to be reserved for non-leaf counters. Therefore, each virtual counter can have up to $b \times (\log_d \frac{M'}{b} + 1)$ bits, which translates to a counting range of $2^{b(\log_d \frac{M'}{b}+1)}$. In contrast, the counting range of each counter is only $2^b - 1$. This means that as long as we spare $\frac{1}{d}M$ memory for upper layers and set $M' = \frac{d-1}{d}M$, we can extend the counting range from $2^b - 1$ to $2^{b(\log_d \frac{M'}{b}+1)}$. The randomized counter sharing scheme [11] is a special case of Counter Tree with $h = 1$. Since it only records flows by one-layer physical counters, the estimation range of randomized counter sharing scheme is very limited, whereas the estimation range of Counter Tree with multiple layers is much larger. As an example, suppose $M = 1\,\text{MB}$, $b = 4$, $d = 2$, and M' is therefore $0.5\,\text{MB}$. Hence, the counting range of each counter is $2^4 - 1 = 15$, while each virtual counter can count up to $2^{4(\log_2 \frac{0.5\,\text{MB}}{4\text{bit}}+1)} = 2^{84}$ packets. Therefore, Counter Tree can significantly extend the estimation range to accommodate large flows.

2.3.5 Design Overview

Our traffic measurement function using Counter Tree consists of two modules. The online packet recording module stores the information of arriving packets in the Counter Tree. For each packet, it is mapped to a virtual counter by one hash computation and then the virtual counter will be updated, which needs approximately two memory accesses. At the end of each measurement period, the Counter Tree is stored to the disk and all counters are then reset to zeros. The offline data decoding module estimates the flow sizes. It is performed by a designated offline computer. Two methods are designed for separating the information about the size of a flow from the noise in the virtual counters. The first one is called Counter Tree base Estimation (CTE). The second one is based on the maximum likelihood estimation method (CTM). Both methods can yield accurate estimates for flow sizes.

2.4 Online Packet Recording

In this section, we show how to record a packet in the Counter Tree.

2.4.1 Recording

Consider an arbitrary flow f. We pseudo-randomly choose r ($r \ll m$) out of the m virtual counters to logically form a *virtual counter array* of f, denoted by V_f. The selection can be achieved by applying r independent hash functions to the flow label. The ith counter of V_f, denoted by $V_f[i]$, is chosen from V as follows:

$$V_f[i] = V[h_i(f)], \tag{2.7}$$

where $0 \le i < r$ and $h_i(\cdot)$ is a hash function $\in [0, m-1]$. To reduce the overhead of implementing r independent hash functions, we can use one master hash function H and a set S of random seeds, and let

$$h_i(f) = H(f \oplus S[i]), \tag{2.8}$$

where $\oplus$ is the XOR operator. Since $r \ll m$, the probability that r distinct virtual counters are selected by the hash functions to form the virtual counter array of f is $\frac{\binom{m}{r}}{m^r} \approx 1$. The bottom half of Fig. 2.4 illustrates the virtual counter arrays for f and g, where $r = 3$ and the virtual counter $V[4]$ is shared by both flows. We point out that our design does not limit the number of flows to be supported. There can be many more flows than the number of virtual vectors, m.

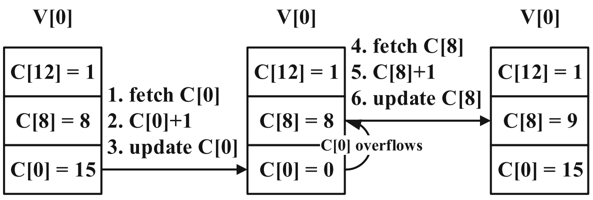


Fig. 2.5 The process for recording a packet to a virtual counter

At the beginning of each measurement period, all counters are initialized to 0s. When a packet of flow f arrives, the router extracts its flow label f, chooses a virtual counter from V_f uniformly at random, and increments that virtual counter by 1. More specifically, the router generates a random number $i \in [0,\ r-1]$, computes the hash value $u = h_i(f)$, and sets $V[u] \leftarrow V[u] + 1$. Note that the update of $V[u]$ may involve the updates of multiple counters. Based on (2.5), the router first fetches counter $C[u]$ from memory and increases it by 1. If $C[u]$ does not overflow, the recording for this packet is done. Otherwise, the router further fetches $C[\lfloor \frac{u}{d} \rfloor + m]$, and adds the overflowed 1 to $C[\lfloor \frac{u}{d} \rfloor + m]$. The process continues until no overflow happens or the counter on the root has been reached. In the latter case, the virtual counter is overflowed beyond the upper bound that it is designed to handle. Figure 2.5 gives an example of the online recording process for a packet of f. Suppose $b = 4$ (i.e., the counting range of each counter is 15), $h = 3$, and $V[0]$ is chosen for recording that packet. The router first fetches $C[0]$ whose value is 15. After adding 1 to $C[0]$, $C[0]$ becomes 0 and leads to an overflow. Hence, the router writes back $C[0] = 0$, further fetches $C[8]$ with value 9, and assigns $C[8] \leftarrow C[8] + 1$. Since $C[8] = 10$ does overflow, the router writes it back and the recording process terminates.

2.4.2 *Number of Memory Accesses*

To record a packet, the router at least needs to read and write 1 counter, which requires 2 memory accesses. Hence, the lower bound of the number of memory accesses for recording a packet is 2. In the worst case, the router needs to update h counters, which requires $2h$ memory accesses. The good thing is that the router needs to fetch another counter only when the current counter overflows, which happens after recording at least 2^b packets. Hence, the amortized number of memory accesses per packet is much smaller than the worst case. We have the following theorem:

Theorem 1. *A tight upper bound of the amortized number of memory accesses for recording a packet in the Counter Tree is* $2 + \frac{2}{2^b-1}$*, where b is length of each counter.*

Proof. Suppose n packets are recorded. Each packet causes one counter at layer 0 to be updated, requiring 2 memory accesses. In total, there are $2n$ memory accesses at layer 0. The number of counter overflows at layer 0 is at most $\lfloor \frac{n}{2^b} \rfloor \leq \frac{n}{2^b}$, which means that counters at layer 1 will be updated for no more than $\frac{n}{2^b}$ times, requiring

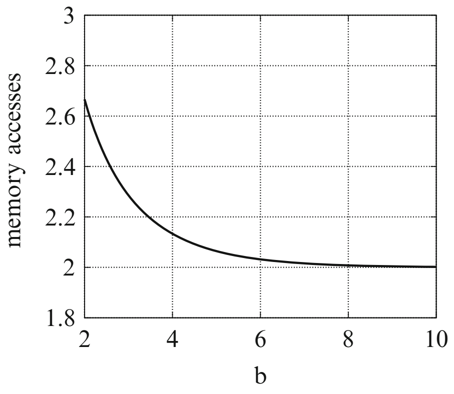


Fig. 2.6 Amortized number of memory accesses per packet with respect to b

no more than $\frac{2n}{2^b}$ memory accesses. By simple induction, we know that the number of memory accesses at layer j is no more than $\frac{2n}{2^{jb}}$. Hence, the total number of memory accesses over all layers is no more than

$$\sum_{j=0}^{h-1} \frac{2n}{2^{jb}} = \frac{2n(1-\frac{1}{2^{bh}})}{1-\frac{1}{2^b}} < 2n(1+\frac{1}{2^b-1}).$$

Hence, the amortized number of memory accesses per packet is no more than

$$\frac{2n(1+\frac{1}{2^b-1})}{n} = 2 + \frac{2}{2^b-1}. \tag{2.9}$$

The upper bound is tight when $n \equiv 0 \pmod{2^{(h-1)b}}$ and exactly $\frac{n}{2^{(j+1)b}}$ overflows happen at the jth layer, $0 \le j < h-1$.

Figure 2.6 shows the upper bound of the amortized number of memory accesses for recording a packet with respect to b. We find that it quickly converges to 2 (the lower bound) with the increase of b. In contrast, Counter Braids need to perform at least $2u$ memory accesses since each packet is recorded by u counters on each layer, where u can be 3.

The non-deterministic counter access time per packet may introduce stalls into the datapath pipeline, resulting in reduced datapath bandwidth. Counter Braids [12, 13] and any other counter architectures with variable access time [17] face the same problem. The key issue is how frequent the variable access time will occur. In case that access time for most packets is constant but only varies for a tiny fraction of packets, the impact on datapath bandwidth will be limited, particularly if we are able to reduce the chance of variable access time to an arbitrarily small level through a system parameter. This is the case for Counter Tree. More specifically, the overflow of a layer-0 counter occurs once every 2^b packets. On average, one out of 2^b packets has longer access time because a counter at the higher layer is involved, but the other $2^b - 1$ packets have constant access time because they do not cause counter overflow. The fraction of all packets that have longer access time is $\frac{1}{2^b}$, which can be exponentially reduced by increasing the value of b, i.e., the number of bits in a counter. For example, when $b = 6$, only 1.6 % of all packets have variable access time and 98.4 % of the packets have constant access time. Suppose each counter overflow causes the pipeline to stall for 10× of the normal access time. The overall access time will increase by approximately 14.4 %, and the throughput therefore decreases by approximately 12.6 %.

2.5 Counter Tree-Based Estimation

After the measurement period, offline estimation should be performed to recover flow sizes from the Counter Tree. Counter Tree estimates flow sizes through some statistical methods, and it supports efficient query on the size of an arbitrary flow, without having to compute the sizes of other flows as Counter Braids do (the computation overhead is high). In this section, we first present and analyze the Counter Tree-based Estimation (CTE) method.

2.5.1 CTE Method

Consider the ith virtual counter $V_f[i]$ in the virtual counter array of flow f. According to (2.7), $V_f[i] = V[u]$, where $u = h_i(f)$. We know $V[u]$ records some of f's packets, as well as the noise introduced by other flows. There are two sources of noise: First, $V[u]$ is shared by other flows; Second, all component counters in $V[u]$ except $C[u]$ are shared by other virtual counters. To accurately recover the number of packets in f recorded by $V[u]$, we need to figure out how to remove such noise.

The component counter of $V[u]$ at the highest layer is $C[v]$, where $v = \lfloor \frac{u}{d^{h-1}} \rfloor + \sum_{t=1}^{h-1} \frac{m}{d^{t-1}}$ according to (2.5). Consider the subtree rooted at $C[v]$, denoted as T, consisting of d^{h-1} leaf nodes, which correspond to d^{h-1} virtual counters. Let $k = d^{h-1}$. Due to counter sharing, flows mapped to those k virtual counters may introduce noise to $V[u]$. We treat T as an aggregate, called a *subtree counter*, when dealing with noise. We denote the value of T by a random variable X_i. As an example, in Fig. 2.3, the value of the subtree counter rooted at $C[12]$ is $C[12] \times 2^{2b} + (C[8] + C[9]) \times 2^b + (C[0] + C[1] + C[2] + C[3])$. Note that the total number n of packets in all flows can be obtained from the entire Counter Tree in a similar way. Let random variable Y_i be the portion of X_i contributed by flow f, and Z_i be the portion of X_i contributed by all other flows. Hence, $X_i = Y_i + Z_i$.

Let s be the true flow size of f during the measurement period. Assume there is a large number of flows, n is large, the size of each flow is negligibly small when comparing with n, r is much larger than 1, and $r \ll m$.

Flow f has r virtual counters (including $V[u]$) in its virtual counter array. Each packet of f has a probability $\frac{1}{r}$ to be mapped to $V[u]$ and increment it by one. Therefore, Y_i follows a binomial distribution:

$$Y_i \sim B(s, \frac{1}{r}). \tag{2.10}$$

Consider an arbitrary packet belonging to a different flow g. The probability for the virtual counter array of g to include a particular virtual counter in T is approximately $1 - \frac{\binom{m-1}{r}}{\binom{m}{r}} = \frac{r}{m}$. When that happens, the conditional probability for this particular virtual counter to record the packet is approximately $\frac{1}{r}$. Combining the above analysis, the probability for this particular virtual counter to record the packet is approximately $\frac{r}{m} \times \frac{1}{r} = \frac{1}{m}$. Since there are k virtual counters in T, the probability that T records the packet is $(1 - \frac{1}{m})^k \approx \frac{k}{m}$ for $k \ll m$. There are $n - s$ packets outside of f. Since there are numerous flows under measurement, the total number n of packets can be much larger than the size s of any single flow, even for large flows, as we observe in our traffic traces. With $s \ll n$ and $n - s \approx n$, Z_i roughly follows a binomial distribution

$$Z_i \sim B(n, \frac{k}{m}). \tag{2.11}$$

Given the distributions of Y_i and Z_i, we know $E(Y_i) = \frac{s}{r}$, and $E(Z_i) = \frac{nk}{m}$. Therefore,

$$E(X_i) = E(Y_i + Z_i) = E(Y_i) + E(Z_i) = \frac{s}{r} + \frac{nk}{m}.$$

Hence, we have

$$s = rE(X_i) - \frac{nkr}{m}. \tag{2.12}$$

We do not know the exact value of $E(X_i)$, but we have the r instance values, also denoted as X_i, $0 \le i < r$, that can be directly counted from the Counter Tree as subtree counters. Replacing $E(X_i)$ with the measured average $\frac{\sum_{i=0}^{r-1} X_i}{r}$, we obtain an estimate of s, denoted as $\hat{s}$, as follows:

$$\hat{s} = \sum_{i=0}^{r-1} X_i - \frac{nkr}{m}, \tag{2.13}$$

where the first term is the number of all packets recorded by the r subtree counters and the second term captures the average noise presented in r counters.

2.5.2 Analysis of $\hat{s}$

Since Y_i and Z_i follow binomial distributions, we have

$$Var(Y_i) = \frac{s}{r}(1 - \frac{1}{r}), \quad Var(Z_i) = \frac{nk}{m}(1 - \frac{k}{m}). \tag{2.14}$$

In addition, Y_i and Z_i are independent with each other. Hence, $Cov(Y_i, Z_i) = 0$. Hence, we have

$$\begin{aligned} Var(X_i) &= Var(Y_i) + Var(Z_i) + 2Cov(Y_i, Z_i) \\ &= \frac{s}{r}(1 - \frac{1}{r}) + \frac{nk}{m}(1 - \frac{k}{m}). \end{aligned} \tag{2.15}$$

Therefore,

$$E(\hat{s}) = E(\sum_{i=0}^{r-1} X_i) - \frac{ndr}{m} = r(\frac{nkr}{m}) - \frac{nkr}{m} = s, \tag{2.16}$$

which means the estimator $\hat{s}$ is unbiased. In addition, we have

$$\begin{aligned} Var(\hat{s}) &= Var(\sum_{i=0}^{r-1} X_i) = r^2 Var(X_i) \\ &= s(r-1) + \frac{nkr^2}{m}(1 - \frac{k}{m}) \\ &= s(r-1) + \frac{nr^2 b(d^h - 1)}{M}(1 - \frac{b(d^h - 1)}{M}), \end{aligned} \tag{2.17}$$

where we have used $m = \frac{d^{h-1}M}{b(d^h-1)}$ and $k = d^{h-1}$. Hence, the standard error of the ratio $\frac{\hat{s}}{s}$ is

$$StdErr(\frac{\hat{s}}{s}) = \frac{\sqrt{s(r-1) + \frac{nr^2 b(d^h-1)}{M}(1 - \frac{b(d^h-1)}{M})}}{s}. \tag{2.18}$$

When n is sufficiently large, the binomial distribution, $Z_i \sim B(n, \frac{k}{m})$, can be approximated by a normal distribution, $N(\frac{nk}{m}, \frac{nk}{m}(1 - \frac{k}{m}))$. Similarly, $Y_i \overset{\text{approx}}{\sim} N(\frac{s}{r}, \frac{s}{r}(1 - \frac{1}{r}))$. Since the linear combination of independent normal distributions also follows normal distribution, $X_i \overset{\text{approx}}{\sim} N(\mu, \sigma^2)$, where $\mu = \frac{nk}{m} + \frac{s}{r}$, and $\sigma^2 = \frac{nk}{m}(1 - \frac{k}{m}) + \frac{s}{r}(1 - \frac{1}{r})$. According to (2.13), we have

$$\hat{s} \overset{\text{approx}}{\sim} N(s, s(r-1) + \frac{nkr^2}{m}(1 - \frac{k}{m})). \tag{2.19}$$

Therefore, the α confidence interval for s is

$$\hat{s} \pm Z_\alpha \sqrt{s(r-1) + \frac{nkr^2}{m}(1-\frac{k}{m})}, \tag{2.20}$$

where Z_α is the $\frac{1+\alpha}{2}$ percentile for the standard normal distribution. For example, when $\alpha = 95\,\%$, $Z_\alpha = 1.96$.

2.6 Counter Tree-Based Maximum Likelihood Estimation

In this section, we provide and analyze another estimator for flow sizes called Counter Tree-based Maximum likelihood Estimation (CTM).

2.6.1 CTM Method

From (2.11), the probability of $Z_i = z_i$ is

$$Prob\{Z_i = z_i\} = \binom{n}{z_i}(\frac{k}{m})^{z_i}(1-\frac{k}{m})^{n-z_i}.$$

The value of n is known from the Counter Tree. The values of m and k are determined by prescribed system parameters M, b and d, h. Hence, $P\{Z_i = z_i\}$ can be written as a function of z_i, denoted as $p(z_i)$. Therefore, the probability for observing $X_i = x_i$ can be calculated by

$$\begin{aligned} Prob\{X_i = x_i\}) &= Prob\{Y_i + Z_i = x_i\} \\ &= \sum_{z_i=0}^{x_i} p(z_i) Prob\{Y_i = x_i - z_i\}) \\ &= \sum_{z_i=0}^{x_i} p(z_i) \binom{s}{y_i}(\frac{1}{r})^{y_i}(1-\frac{1}{r})^{s-y_i} \\ &= \sum_{z_i=0}^{x_i} p(z_i) q(s, y_i), \end{aligned} \tag{2.21}$$

where $y_i = x_i - z_i$, $q(s, y_i) = \binom{s}{y_i}(\frac{1}{r})^{y_i}(1 - \frac{1}{r})^{s-y_i}$, and we have used $Y_i \sim B(s, \frac{1}{r})$. Hence, the likelihood function for observing $X_0 = x_0, X_1 = x_1, \ldots, X_{r-1} = x_{r-1}$ is

$$L(s; x_0, x_1, \ldots, x_{r-1}) = \prod_{i=0}^{r-1} \sum_{z_i=0}^{x_i} p(z_i)q(s, y_i). \tag{2.22}$$

Taking the logarithm for both sides of the likelihood function, we obtain the log-likelihood as follows:

$$\ln L = \sum_{i=0}^{r-1} \ln(\sum_{z_i=0}^{x_i} p(z_i)q(s, y_i)).$$

Using the logarithmic differentiation,[1] we can calculate

$$\frac{d\binom{s}{y_i}}{ds} = \binom{s}{y_i} \sum_{j=0}^{y_i-1} \frac{1}{s-j}.$$

Hence,

$$\begin{aligned}\frac{dq(s, y_i)}{ds} &= \binom{s}{y_i}(\frac{1}{r})^{y_i}(1 - \frac{1}{r})^{s-y_i}(\sum_{j=0}^{y_i-1} \frac{1}{s-j} + \ln(1 - \frac{1}{r})) \\ &= q(s, y_i)(\sum_{j=0}^{y_i-1} \frac{1}{s-j} + \ln(1 - \frac{1}{r})).\end{aligned}$$

Finally, we obtain

$$\frac{d \ln L}{ds} = \sum_{i=0}^{r-1} \frac{\sum_{z_i=0}^{x_i} p(z_i)q(s, y_i)(\sum_{j=0}^{y_i-1} \frac{1}{s-j} + \ln(1 - \frac{1}{r}))}{\sum_{z_i=0}^{x_i} p(z_i)q(s, y_i)}. \tag{2.23}$$

The value of s that satisfies $\frac{d \ln L}{ds} = 0$ will maximize $\ln L$ and thereby the likelihood function L. In addition, we observe that $\frac{d \ln L}{ds}$ is monotonically decreasing with respect to s. Therefore, the bisection search method can be used to find the s value such that $\frac{d \ln L}{ds} = 0$. As a result, we obtain an estimator for s as follows:

$$\hat{s} = \arg\max_{s}\{\ln L\} = \{s | \frac{d \ln L}{ds} = 0\}. \tag{2.24}$$

[1] Since $[\ln(f)]' = \frac{f'}{f}$, we know $f' = f[\ln(f)]'$.

2.6.2 Analysis of $\hat{s}$

Following the analysis in Sect. 2.5.2, $X_i \overset{\text{approx}}{\sim} N(\mu, \sigma^2)$. Hence, the probability density function of X_i is $f(x_i) = \frac{1}{\sqrt{2\pi}\sigma} e^{-\frac{(x_i-\mu)^2}{2\sigma^2}}$. Therefore, the likelihood function for observing $X_0 = x_0, X_1 = x_1, \ldots, X_{r-1} = x_{r-1}$ can also be written as

$$L(s; x_0, x_1, \ldots, x_{r-1}) = \prod_{i=0}^{r} \frac{1}{\sqrt{2\pi}\sigma} e^{-\frac{(x_i-\mu)^2}{2\sigma^2}}.$$

Taking the logarithm of the likelihood function, we have

$$\begin{aligned}\ln L &= \sum_{i=0}^{r-1} \ln\left(\frac{1}{\sqrt{2\pi}\sigma} e^{-\frac{(x_i-\mu)^2}{2\sigma^2}}\right),\\ &= \sum_{i=0}^{r-1} -\ln\sqrt{2\pi} - \ln\sigma - \frac{(x_i-\mu)^2}{2\sigma^2}.\end{aligned}$$

The first and second order derivatives of $\ln L$ with respect to s are

$$\begin{aligned}\frac{d\ln L}{ds} &= \sum_{i=0}^{r-1} -\frac{r-1}{2r^2\sigma^2} + \frac{x_i-\mu}{r\sigma^2} + \frac{(x_i-\mu)^2(r-1)}{2r^2(\sigma^2)^2},\\ \frac{d^2\ln L}{ds^2} &= \sum_{i=0}^{r-1}\left(\frac{(r-1)^2}{2r^4(\sigma^2)^2} - \frac{1}{r^2\sigma^2} - \frac{2(r-1)(x_i-\mu)}{r^3(\sigma^2)^2}\right.\\ &\qquad \left. - \frac{(r-1)^2(x_i-\mu)^2}{r^4(\sigma^2)^3}\right),\end{aligned}$$

where we have used $\frac{d\mu}{ds} = \frac{1}{r}$ and $\frac{d\sigma^2}{ds} = \frac{r-1}{r^2}$. Hence,

$$\begin{aligned}E\left(\frac{d^2\ln L}{ds^2}\right) &= r\left(\frac{(r-1)^2}{2r^4(\sigma^2)^2} - \frac{1}{r^2\sigma^2} - \frac{(r-1)^2\sigma^2}{r^4(\sigma^2)^3}\right)\\ &= -\frac{2r\sigma^2 + (r-1)^2}{2r^3\sigma^4},\end{aligned}$$

since $E(x_i - \mu) = 0$ and $E(x_i - \mu)^2 = \sigma^2$. Hence, the fisher information [9] is $I(s|x_0, x_2, \ldots, x_{r-1}) = -E(\frac{d^2\ln L}{ds^2}) = \frac{2r\sigma^2+(r-1)^2}{2r^3\sigma^4}$. According to the asymptotic properties of maximum likelihood estimators [9], we have

$$\hat{s} \xrightarrow{d} N\left(s, \frac{1}{I(s|x_0, x_2, \ldots, x_{r-1})}\right) = N\left(s, \frac{2r^3\sigma^4}{2r\sigma^2 + (r-1)^2}\right) \tag{2.25}$$

Therefore, the standard relative error is

$$StdErr(\frac{\hat{s}}{s}) = \frac{\sqrt{\frac{2r^3\sigma^4}{2r\sigma^2+(r-1)^2}}}{s}, \tag{2.26}$$

and the α confidence interval for s is

$$\hat{s} \pm Z_\alpha \sqrt{\frac{2r^3\sigma^4}{2r\sigma^2 + (r-1)^2}}. \tag{2.27}$$

2.7 Enhanced Counter Tree Architecture

2.7.1 Motivation

Recall that the design of vertical counter sharing in the Counter Tree reserves counters at higher layers for large flows. However, the question that which flows have actually used the high-layer counters is left open since each high-layer counter can be shared by multiple virtual counters. If two flows share a high-layer counter and only one of them uses the counter for recording its packets, there is no way to figure out which of the two actually uses that counter from the values in the Counter Tree, which leads to ambiguity. Consider a simple example in which f is a small flow with size 1 and g is large flow with size 30,000. As shown in Fig. 2.4, we assume the packet of f is recorded in $V[1]$ (more specifically $C[1]$ since other counters in $V[1]$ are not used), and 10,000 of g's packets are recorded in $V[3]$. As a result, $C[12]$ is used by g to accommodate such a large number of packets. We know that $V[1]$ and $V[3]$ share the component counter $C[12]$. Although $C[12]$ has never been used by f, it is also included as a component in f's virtual counter $V[1]$. As a result, a great noise is introduced by g when we estimate the size of f since $C[12]$ is included in the estimation of f. To handle this problem, our previous design relies on mapping each flow to many virtual counters (which helps amortizing the noise) and using statistical means to remove the noise. However, we suspect that large noise introduced at higher layers can nevertheless degrade the estimation accuracy. In this section, we introduce additional mechanism to address this issue.

We define the *length* of each virtual counter as the number of component counters it truly uses. In the previous example, the length $V[1]$ is 1 since only $C[1]$ is used, while the length of $V[3]$ is 3 since $C[3]$, $C[9]$, and $C[12]$ have been used. We observe that the aforementioned ambiguity results from the uncertainly of the length of each virtual counter after recording. In the previous design of Counter Tree, we simply assume every virtual counter has the same length, which is equal to the height h of the tree. The experiment results in Sect. 2.8 will demonstrate that Counter Tree works well when the tree height h is set small, e.g., $h = 2$. However, the performance of Counter Tree seriously deteriorates when the tree height grows for the purpose of extending estimation range. Theoretically, this observation is embodied by the fact that the variance of $\hat{s}$ increases exponentially with h according to (2.17). To accommodate large flows, reasonably large h should be used, leading to large estimation error. Therefore, we must figure out how to resolve the ambiguity.

2.7.2 Counters with Status Bits

To address this issue, we design an Enhanced Counter Tree (ECT) architecture which determines the length of each virtual counter and thereby resolves the ambiguity by adding a status bit to each counter. For each b-bit counter, its lower $(b-1)$ bits are used for counting packets, and its most significant bit is used as the *status bit*. See Fig. 2.7 for illustration. Only if the counter overflows, will the status bit of that counter be set. With the status bits, we can calculate the true length for each individual virtual counter, thereby reducing the noise caused by vertical counter sharing among virtual counters. More specifically, to build a virtual counter according to its length, we traverse along the path from its layer-0 counter to the root, and include all counters until the first counter (included) whose status bits have not been set. Following the aforementioned example, Fig. 2.8 shows that the introduction of status bits can alleviate the ambiguity in sharing at higher layers. In the figure, $C[1]$ is a component counter of $V[1]$ that f is mapped to. Since $C[1]$ does not overflow (i.e., its status bit is not set), the length of $V[1]$ must be 1 and therefore

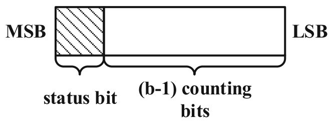


Fig. 2.7 A b-bit counter consists of $(b-1)$ counting bits and a status bit

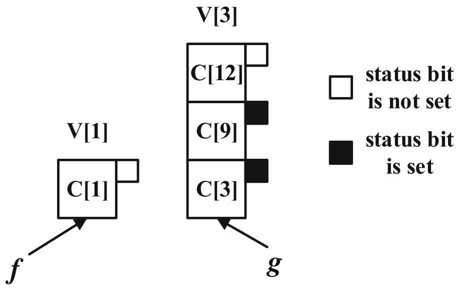


Fig. 2.8 After introducing status bits, we can calculate the lengths of virtual counters based on the status bits in their component counters. The length of $V[1]$ that f is mapped to is 1 since the status bit of $C[1]$ is not set, and the length of $V[3]$ that g is mapped to is 3 since the status bits of $C[3]$ and $C[9]$ are set while the status bit of $C[12]$ is not set

the component counter $C[12]$ should has been included. In contrast, the length of $V[3]$ (which g is mapped to) is 3 since the status bits of both $C[3]$ and $C[9]$ are set while the status bit of $C[12]$ is not. In conclusion, the extra status bits can provide better resolution for virtual counters and help resolving ambiguity about whether a high-layer component counter should be included in a virtual counter or not.

If a counter overflows due to a large flow, the small flows that share the counter will observe a large counter value. However, each flow is assigned to r counters at the bottom layer, where r is large, e.g., in hundreds. The number of large flows is much smaller than the number of small/medium flows in real traffic. For a small flow, if a few of its counters are overflowed due to sharing with large flows, the values of these counters will be large. But as long as most of the small flow's counters have small values, the effect of counter crosstalk will be dampened by the maximum likelihood estimation (2.24), which discounts the outliers (large counter values).

For the first estimator (2.13), when there are many, many flows, even without a large flow, the sheer number of small flows mapped to a counter may cause the counter to overflow. But this noise can be statistically measured and removed when r is sufficiently large. Among the r counters, statistically, some will carry larger-than-average noise and some will carry smaller-than-average noise. When r is large enough, such difference will be evened out due to the law of large numbers. The average noise term is captured by the second term in (2.13).

2.7.3 Recording and Estimation

After employing status bits, the process of online packet recording remains the same except that we need to set its status bit when a counter overflows. For the offline estimation process, we should slightly modify the estimators given in (2.13) and (2.24) to reflect the change. Consider the estimation of flow f. Let l_i be the length of the virtual counter $V_f[i]$ after the measurement period, where $0 \leq i < r$. Instead of using a unified height h for all subtree counters, the height of the subtree counter corresponding to $V_f[i]$ should be l_i, and the value k, meaning the number of leaf nodes in the subtree, is therefore d^{l_i-1}. For example, in Fig. 2.8, the height of the subtree counter corresponding to $V[1]$ is 1 and it contains only one leaf node, while the height of the subtree counter corresponding to $V[3]$ is 3 and it contains 2^{3-1} leaf nodes. Everything else remains the same for the offline estimation. Our experiments in Sect. 2.8 will show that the ECT with status bits remarkably improves the estimation accuracy of CT.

2.8 Experimental Evaluation

2.8.1 Experiment Setup

We have implemented the two estimators based on the Counter Tree, i.e., CTE and CTM, from Sects. 2.5 and 2.6, respectively. CTE and CTM share the same module for online packet recording, which performs the operations described in Sect. 2.4. Hence, when we evaluate the online operations of the Counter Tree, we use CT as the abbreviation. We have also implemented the Enhanced Counter Tree architectures. We compare them with the most related counter architectures: (1) randomized counter sharing (MLM) [10, 11] and (2) Counter Braids (CB) [12, 13]. The bitmap-based MSCBF is less memory efficient than the counter architectures [10]. Hence, we do not include it for comparison. Without losing generality, we use TCP flows for presentation, and we have obtained similar results when carrying out experiments with other types of flows. The network trace we use was captured by Cisco's NetFlow at the main gateway of our university, and we are authorized to store the packets headers in disk for our experiments. The trace contains about 68 million TCP flows and 750 million packets. We implement those counter architectures in software and evaluate their performance by running experiments on the trace. Suppose each measurement period contains 10 million packets. We divide the trace into measurement periods, and perform different estimators in each period. We use all 750 million packets in our experiments, for 75 periods. We randomly pick the results from one period to report. In fact, the results from different periods are quite similar. During the chosen period, there are 1,070,632 TCP flows and 10,053,234 packets, and the minimum, average, and maximum flow size is 1, 9.39, and 10,972 packets, respectively.

We conduct four sets of experiments. The first set is used for comparing the estimation accuracy and range of CT, MLM, and CB. We vary the available memory space M from 0.125, 0.25, 0.5, to 1 MB, which translate to about 1bit/flow, 2bits/flow, 4bits/flow, and 8bits/flow, respectively. For Counter Braids, we use the same settings as [13]: A two-layer CB and three hash functions at both layers. The layer-1 counters are 8 bits deep and the layer-2 counters are 56 bits deep. For MLM, we set the counter length to 6 bits and the size of each storage vector to 100 as in [10]. For CTE and CTM, we implement a 2-layer tree (which is sufficient for our experiments) with $d = 2$ and $b = 4$ by default. For fair comparison with MLM, we set the size r of each virtual counter array to 100. The second set of experiments compare the processing overhead of online packet recording of these counter architectures. In the third set of experiments, we will vary the values of d, b, and r to study their respective impact on performance. In the fourth set of experiments, we compare the performance of Counter Tree and Enhanced Counter Tree under different tree heights.

Table 2.2 Comparison of average processing time for encoding a packet by CB, MLM, and CT

	Number of memory accesses			Number of hash computations		
Memory size (MB)	CB	MLM	CT	CB	MLM	CT
0.25	6.01	2	2.09	3.01	1	1
0.5	6.01	2	2.06	3.00	1	1
1	6.01	2	2.03	3.00	1	1
2	6.01	2	2.02	3.00	1	1

2.8.2 Processing Time for Recording a Packet

The processing time for encoding a packet mainly results from memory accesses to read and write counters and the computations of hash values. A typical implementation of Counter Braids requires 3 hash functions on each layer, mapping each flow to the corresponding counters. To encode a packet, the router needs to read the 3 associated counters on the first layer, increment them by 1, and then write them back to the memory. If any of the 3 counters overflows, the router has to read and write another 3 counters on the second layer, which requires another 3 hash computations. Hence, the lower bounds of the number of memory accesses and the number of hash computations by CB are 6 and 3, respectively. In contrast, MLM aligns all counters on the same layer, and each packet is hashed to only one counter, which requires 2 memory access and 1 hash computation. CT also only requires 1 hash computation to determine the virtual counter for a packet. Recall that (2.9) gives an upper bound of amortized number of memory accesses by CT. When $b = 4$, the amortized number of memory accesses is bounded by $2 + \frac{1}{2^4-1} \approx 2.13$. In the first set of experiments, we record the average number of memory accesses and average number of hash computations for encoding a packet by CB, MLM, and CT. The results are shown in Table 2.2. We can see that CT is almost as efficient as MLM, and they achieve approximately $3\times$ efficiency of CB. Moreover, the average number of memory accesses of CT decreases when more memory (counters) are available since each counter is shared by fewer flows, which reduces the overflows.

2.8.3 Estimation Accuracy and Range

Recall that our main objective is to design a counter architecture that can work in very tight space where existing counter architectures no long work well. So we first compare Counter Tree with CB and MLM in terms of estimation accuracy and range to see how they work under the same available memory. The estimation results of CB are shown in Fig. 2.9 which includes four plots for different values of M. Each point in the plots represents an $(s, \hat{s})$ pair for a particular flow, where the x coordinate is the true flow size s and the y coordinate is the estimated flow size $\hat{s}$. The equality line, $y = x$, is presented for reference: The closer a point is to the

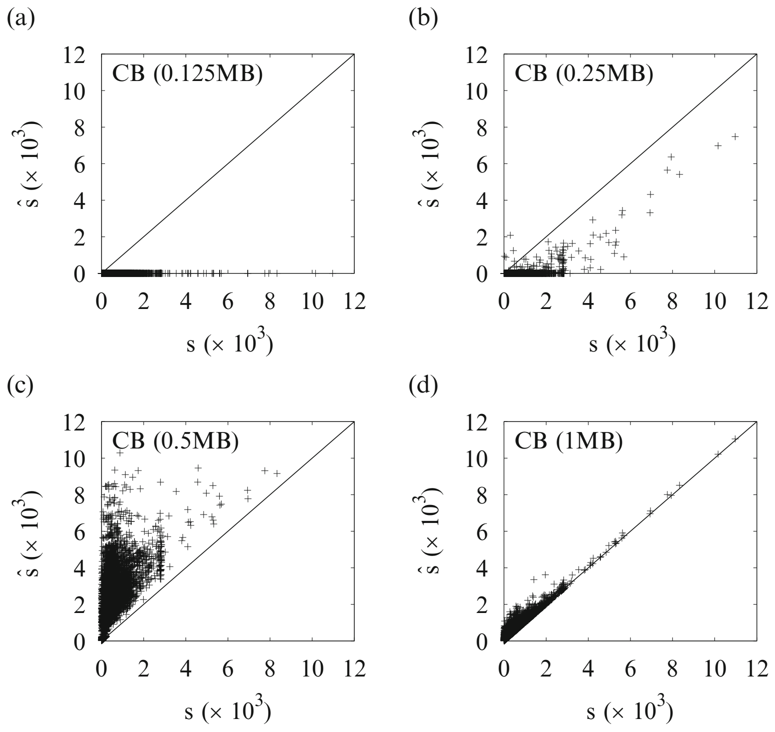


Fig. 2.9 (**a**) Shows estimation results by Counter Braids when $M = 0.125\,\text{MB}$. Each point in the plot represents an $(s, \hat{s})$ pair for a particular flow, where the x coordinate is the true flow size s and the y coordinate is the estimated flow size $\hat{s}$. The equality line, $y = x$, is presented for reference: a point closer to the equality line is more accurate. (**b**) Shows estimation results by Counter Braids when $M = 0.25\,\text{MB}$. (**c**) Shows estimation results by Counter Braids when $M = 0.5\,\text{MB}$. (**d**) Shows estimation results by Counter Braids when $M = 1\,\text{MB}$

equality line, the more accurate the estimate is. We can see that when a very tight memory $M = 0.125\,\text{MB}$ is used, CB cannot produce any meaningful results. When $M = 0.5\,\text{MB}$, CB generates positively biased results that are all above the equality line. When the available memory space increases to 1 MB, the estimation results of CB do not converge. So we terminate the process after 1000 iterations. We find that when $M \geq 2\,\text{MB}$, CB can yield very accurate estimates (which is not shown in the figure). Therefore, CB does not suite for traffic measurement under very tight memory.

Figure 2.10 presents the estimation results of MLM. MLM does not work when $M = 0.125\,\text{MB}$. Although MLM can yield accurate estimates for small or moderate flows when more memory is available, it produces very negatively biased results for large flows. Because large flows may lead to counter overflows, some packets cannot be recorded when the counters are fully used. Although the increase of M can

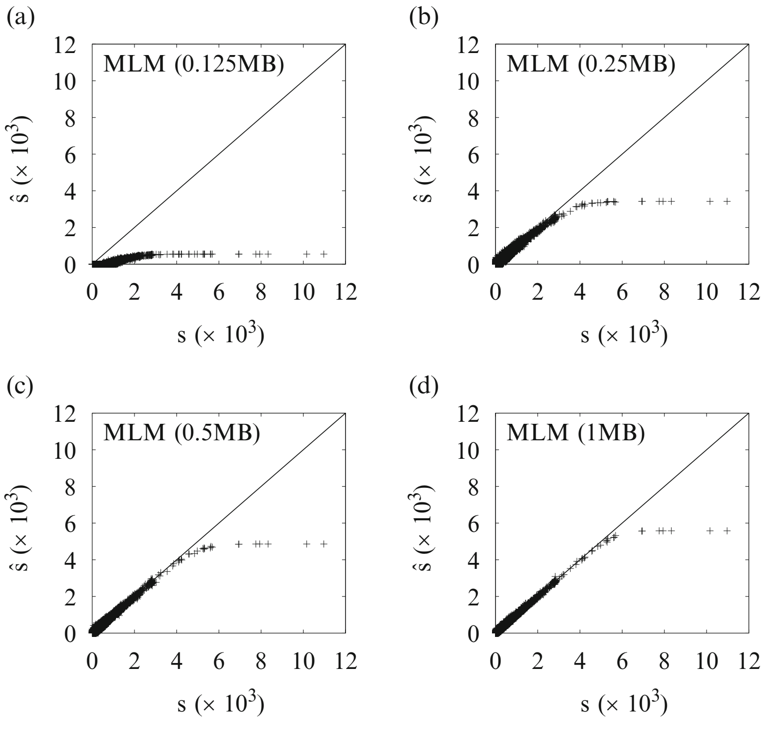


Fig. 2.10 Estimation results by MLM when $M = 0.125$ MB, 0.25 MB, 0.5 MB, and 1 MB (**a**)–(**d**), respectively

enlarge the estimation range of MLM to some extent, it does not address the problem fundamentally. For example, the estimation range is about 6000 when $M = 1$ MB.

The estimation results of CTE and CTM are given in Figs. 2.11 and 2.12, respectively. As expected, the employment of the Counter Tree architecture significantly extends the counting range than MTM. Both CTE and CTM can yield very accurate estimates for all flows, including flows with very large sizes, even under a tight memory. The estimates become more accurate when more memory space is available. The relative estimation bias $Bias(\frac{\hat{s}}{s})$ and the relative standard error $\frac{\sqrt{Var(\hat{s})}}{s}$ of CTE and CTM are presented in Fig. 2.13. We find that CTE and CTM in fact have comparable estimation accuracy. Figure 2.13 shows that the relative errors for large flows are small. Generally, $Bias(\frac{\hat{s}}{s})$ and $\frac{\sqrt{Var(\hat{s})}}{s}$ decrease with the increase of s. Although the relative errors for small flows can be large, the results in Figs. 2.11 and 2.12 demonstrate that no small flows will deviate significantly from the equality line for large absolute errors (which would cause mis-classification). It is expected and true that the relative errors for small flows are large for virtually all estimation methods. We use an extreme example to bring out the idea: There is a difference in interpreting the results for small flows and those for large flows. Consider a small

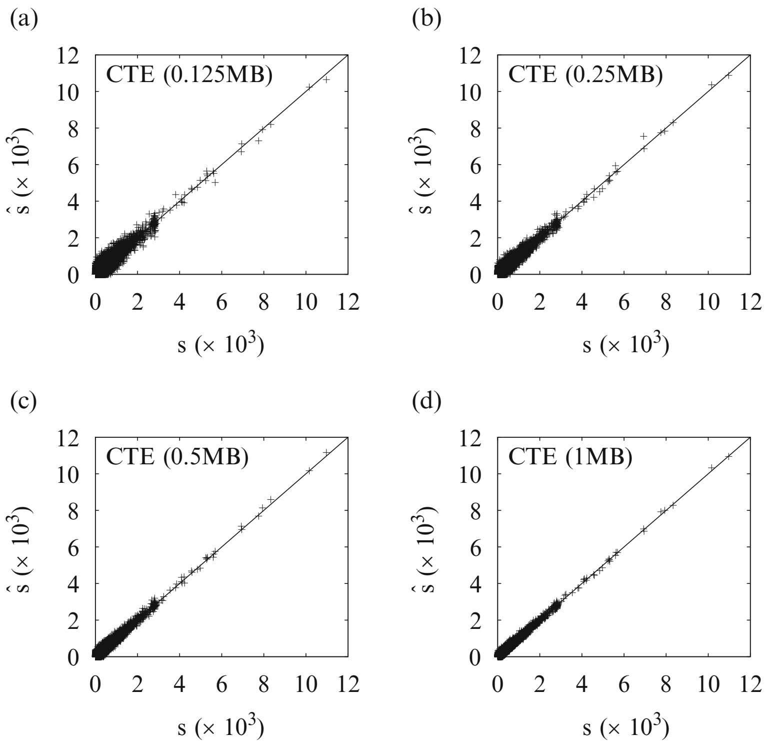


Fig. 2.11 Estimation results by CTE when $M = 0.125$ MB, 0.25 MB, 0.5 MB, and 1 MB (**a**)–(**d**), respectively

flow of size 1. If the estimation is 2 (which is in fact a good result because it is off by just 1), the relative error is 100 %. For a large flow of 10,000, if the relative error is 100 %, it will be 20,000, a truly bad estimation. For the same large flow, if the estimation is off by 10, it is a great estimation because the relative error is just 0.1 %. However, if the previous small flow is off by 10, the relative error is 1000 %—even with such a relative error, we would not necessarily say this is a bad estimation because a flow of 11 packets will not be mis-classified as a large flow, in applications of identifying elephant flows or classifying all flows into a few categories based on their sizes.

To numerically evaluate how large flows affect the estimation results of small flows, we only consider small flows that share counters with large flows, and omit the small flows that do not share any counters with large flows. More specifically, we consider all flows with sizes smaller than 100 as small, while those with sizes larger than 1000 as large. However, we observe that all small flows share multiple counters with large flows. The reason is that each flow uses a large number r of counters, which means a small flow will have a good chance to share at least one

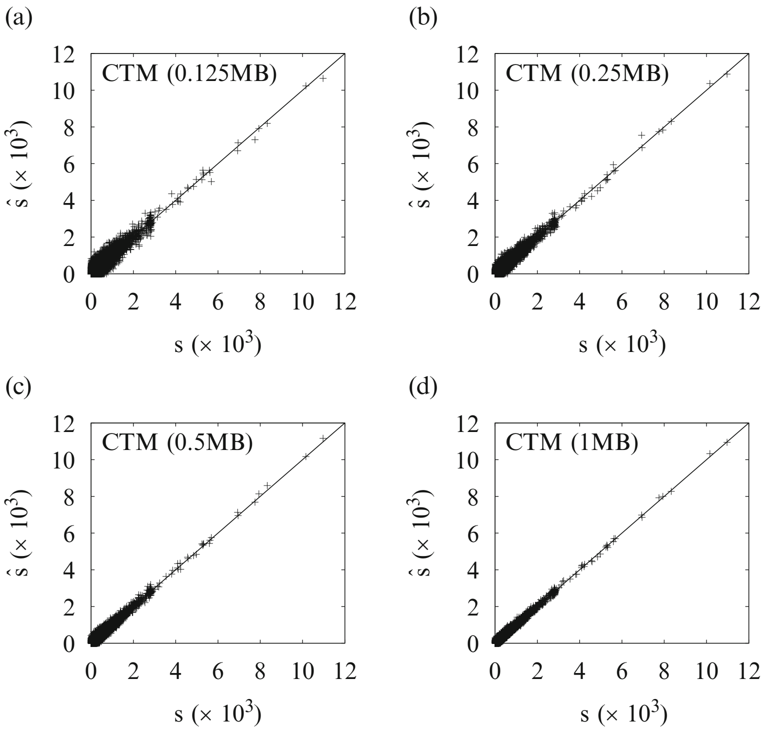


Fig. 2.12 Estimation results by CTM when $M = 0.125$ MB, 0.25 MB, 0.5 MB, and 1 MB (**a**)–(**d**), respectively

counter with one of the large flows. We stress that our method is designed to handle such sharing through noise removal and maximum likelihood estimation.

In conclusion, CT works much better than CB and MLM under a tight memory, and it significantly extends the estimation range when compared with MLM.

2.8.4 Impact of b, r, and d

We now vary the system parameters b, r, and d to study their impacts on the performance of CT, while M is fixed to 0.5 MB. The parameters are set as follows:

1. Impact of b: we fix $d = 2$, $r = 100$, $h = 2$ and vary b from 4, 6, to 8.
2. Impact of r: we fix $b = 4$, $d = 2$, $h = 2$ and vary r from 50, 100, to 200.
3. Impact of d: we fix $b = 4$, $r = 100$, $h = 2$ and vary d from 2, 3, to 4.

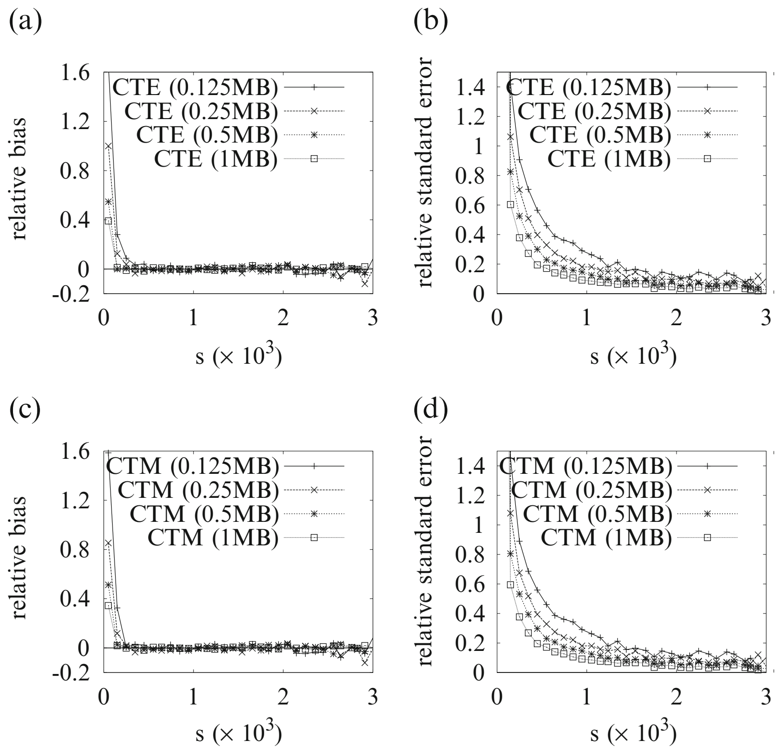


Fig. 2.13 (**a**) Shows the relative estimation bias $Bias(\frac{\hat{s}}{s})$ of CTE. (**b**) Shows the relative standard error $StdErr(\frac{\hat{s}}{s})$ of CTE. (**c**) Shows the relative estimation bias $Bias(\frac{\hat{s}}{s})$ of CTM. (**d**) Shows the relative standard error $StdErr(\frac{\hat{s}}{s})$ of CTM

We find that those parameters affect CTE and CTM in a similar way. Hence, we only present the estimation results of CTE in Figs. 2.14, 2.15, and 2.16. When we increase the b, r, or d, the estimation range will be increased accordingly. But this does not come for free. Since the increase of b, r, or d makes more flows share each counter, it is expected that the estimation accuracy will degrade due to elevated noise. However, the experimental results show that such degradation is small, and the performance of CTE is not very sensitive to the change of b, r, or d (we will show shortly that the change of h can significantly affect the estimation accuracy of CTE).

2.8.5 *Comparison of CTE and E-CTE*

Recall that our analysis in Sect. 2.7 points out that the increase of tree height h can degrade the performance of CTE and CTM. To demonstrate this, we run

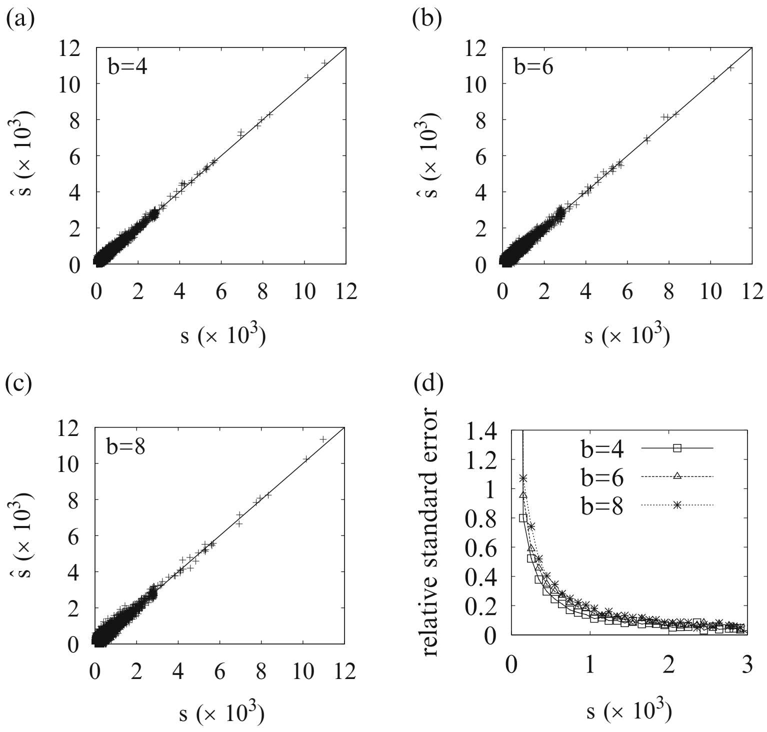


Fig. 2.14 Impact of b on the performance of CTE, where $M = 0.5\,\text{MB}$, $d = 2$, and $r = 100$. (**a**) Shows estimation results of CTE when $b = 4$. (**b**) Shows estimation results of CTE when $b = 6$. (**c**) Shows estimation results of CTE when $b = 8$. (**d**) Shows the comparison of relative standard error when $b = 4, 6, 8$

experiments using the CTE method (similar results can be observed if the CTM method is adopted) with different tree heights. We fix $M = 0.5\,\text{MB}$, $b = 4$, $d = 2$, $r = 100$, and vary h from 2, 4, to 6. The results are presented in Fig. 2.17. As we expect, the estimation accuracy of CTE becomes much worse with the increase of h. The fourth plot in Fig. 2.17 demonstrates that a larger h significantly increases the relative standard error of the estimates. We use E-CTE to stand for the CTE method under the Enhanced Counter Tree architecture which is designed to address this issue. For comparison, we conduct the same experiments on E-CTE, and the results are depicted in Fig. 2.18. Owing to the status bits in E-CTE, the increase of h only slightly degrades the performance of ECT. Therefore, ECT dramatically outperforms CTE when a relatively large h is adopted, e.g., $h = 6$.

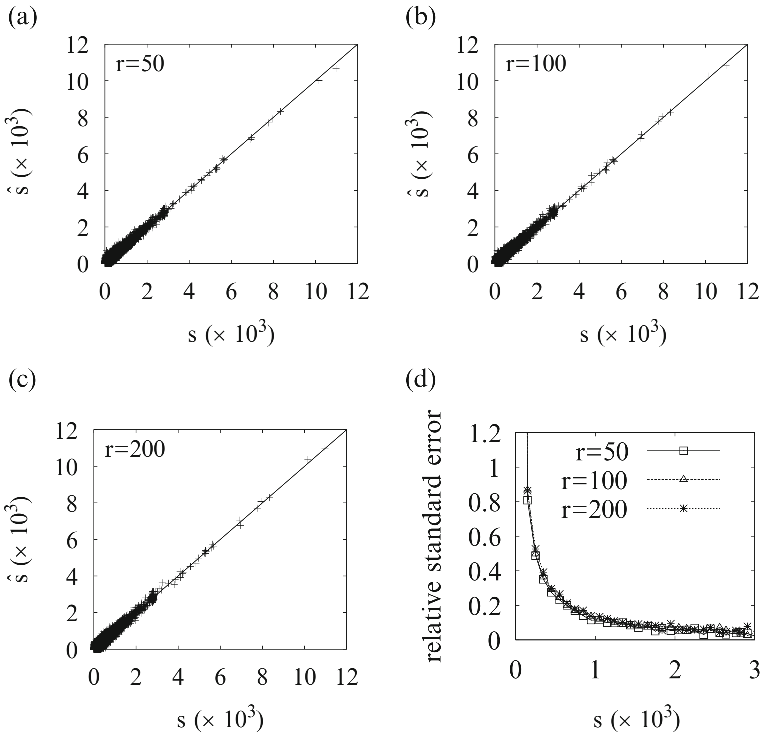


Fig. 2.15 Impact of r on the performance of CTE, where $M = 0.5\,\text{MB}$, $b = 4$, and $d = 2$. (**a**) Shows estimation results of CTE when $r = 50$. (**b**) Shows estimation results of CTE when $r = 100$. (**c**) Shows estimation results of CTE when $r = 200$. (**d**) Shows the comparison of relative standard error when $r = 50,\ 100,\ 200$

2.8.6 Scalability of Counter Tree

To evaluate the scalability of Counter Tree, we apply it to a larger trace, which contains 8,253,368 TCP flows and 100,000,015 packets, and the minimum, average, and maximum flow size is 1, 12.12, and 295,451 packets, respectively. The available memory space M remains to be 0.125, 0.25, 0.5, and 1 MB, which translate to about 0.125bits/flow, 0.25bits/flow, 0.5bits/flow, and 1bits/flow, respectively. We implement Counter Tree with different system parameters. Figure 2.19 shows the estimation results of E-CTE with $b = 4$, $d = 2$, $r = 100$, and $h = 4$. It is clear that Counter Tree can still yield reasonably accurate estimation results under an extremely tight memory space, e.g., 0.125 bits/flow.

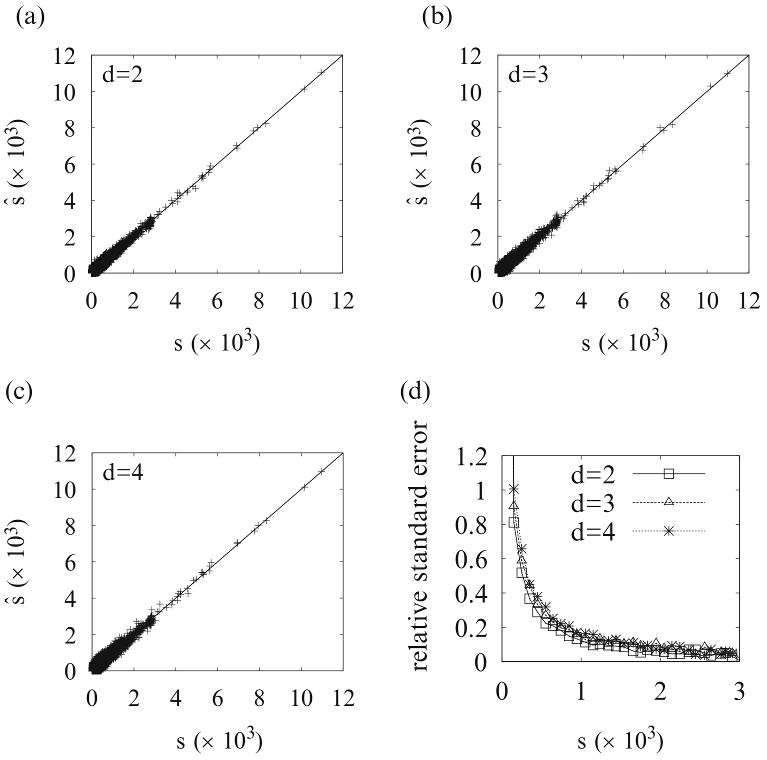


Fig. 2.16 Impact of d on the performance of CTE, where $M = 0.5\,\text{MB}$, $b = 4$, and $r = 100$. (**a**) Shows estimation results of CTE when $d = 2$. (**b**) Shows estimation results of CTE when $d = 3$. (**c**) Shows estimation results of CTE when $d = 4$. (**d**) Shows the comparison of relative standard error when $d = 2, 3, 4$

2.9 Summary

This chapter presents a scalable Counter Tree architecture. We develop a two-dimensional sharing technique, where each counter can be shared by multiple virtual counters and each virtual counter can be shared by multiple flows. As a result, Counter Tree significantly reduces memory requirement and extends estimation range. To encode a packet, Counter Tree only requires a little more than 2 memory accesses, which is asymptotically optimal. We use two offline decoding methods to estimate flow sizes. The extensive experiments with real network trace demonstrate that our methods can yield very accurate estimates even under an extremely tight memory space, e.g., 2 bits per flow.

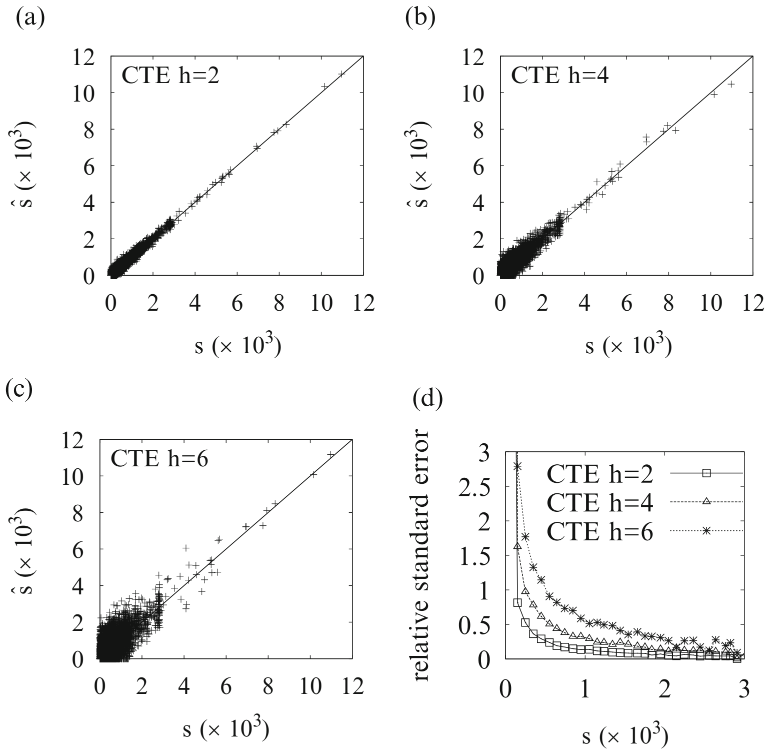


Fig. 2.17 Performance of CTE with different tree height h, where $M = 0.5\,\text{MB}$, $b = 4$, $d = 2$, and $r = 100$. (**a**) Shows estimation results of CTE when $h = 2$. (**b**) Shows estimation results of CTE when $h = 4$. (**c**) Shows estimation results of CTE when $h = 6$. (**d**) Shows the comparison of relative standard error when $h = 2, 4, 6$

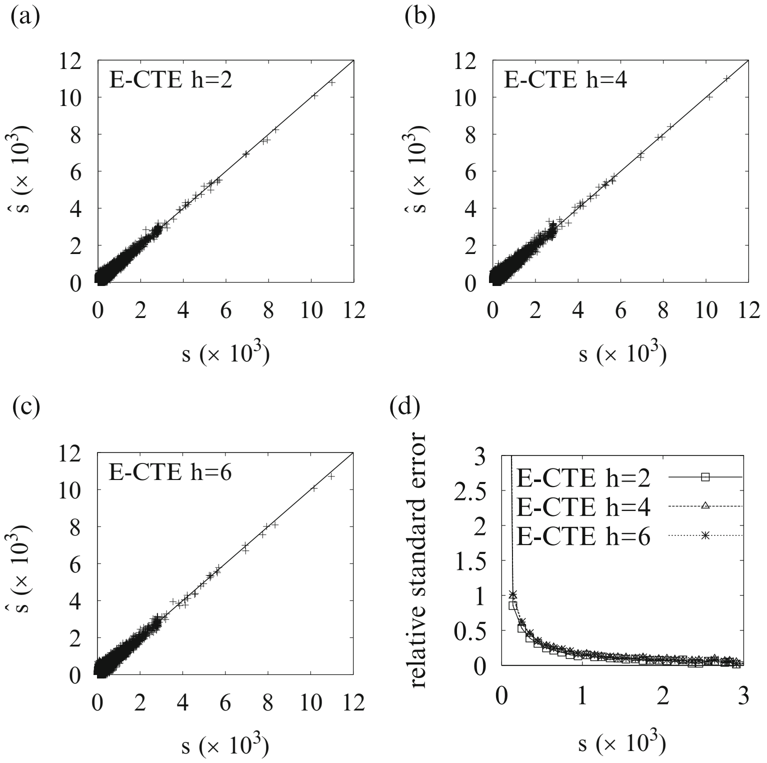


Fig. 2.18 Performance of E-CTE with different tree height h, where $M = 0.5\,\text{MB}$, $b = 4$, $d = 2$, and $r = 100$. (**a**) Shows estimation results of E-CTE when $h = 2$. (**b**) Shows estimation results of E-CTE when $h = 4$. (**c**) Shows estimation results of E-CTE when $h = 6$. (**d**) Shows the comparison of relative standard error when $h = 2, 4, 6$

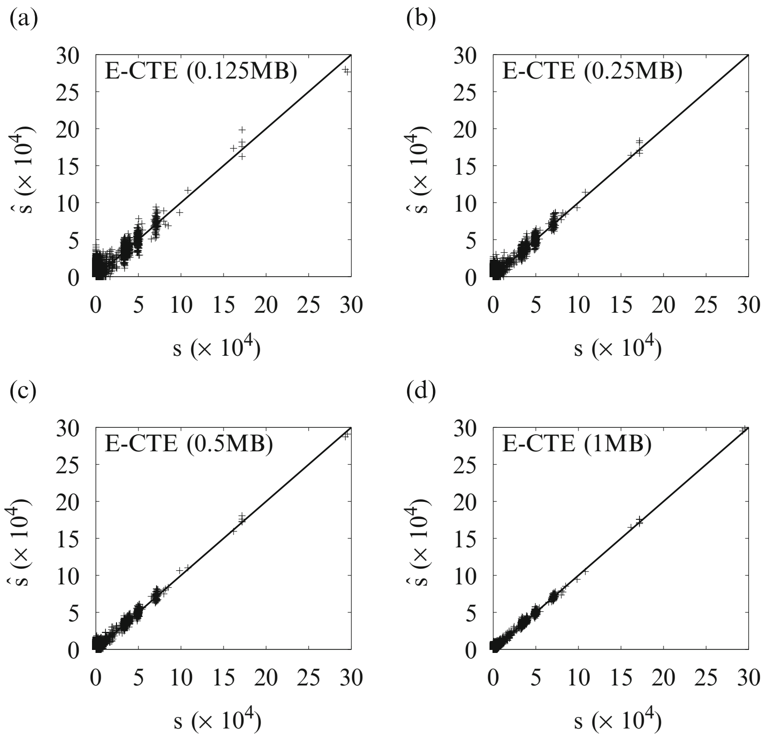


Fig. 2.19 Estimation results of E-CTE for a trace with 100 million packets. We set $b = 4$, $d = 2$, $r = 100$, $h = 4$, and $M = 0.125$ MB, 0.25 MB, 0.5 MB, and 1 MB (**a**)–(**d**), respectively

References

1. Chen, M., Chen, S.: Counter tree: a scalable counter architecture for per-flow traffic measurement. In: Proceedings of IEEE ICNP, pp. 111–122 (2015)
2. Cormode, G., Muthukrishnan, S.: An improved data stream summary: the Count-Min sketch and its applications. In: Proceedings of LATIN (2004)
3. Duffield, N., Lund, C., Thorup, M.: Estimating flow distributions from sampled flow statistics. In: Proceedings of ACM SIGCOMM (2003)
4. Einziger, G., Friedman, R.: Counting with tinytable: every bit counts! In: Proceedings of IEEE INFOCOM Workshops, pp. 77–78 (2015)
5. Estan, C., Varghese, G.: New directions in traffic measurement and accounting. In: Proceedings of ACM SIGCOMM (2002)
6. Kamiyama, N., Mori, T.: Simple and accurate identification of high-rate flows by packet sampling. In: Proceedings of IEEE INFOCOM (2006)
7. Kumar, A., Sung, M., Xu, J., Wang, J.: Data streaming algorithms for efficient and accurate estimation of flow size distribution. In: Proceedings of ACM SIGMETRICS (2004)
8. Kumar, A., Xu, J., Wang, J.: Space-code bloom filter for efficient per-flow traffic measurement. IEEE J. Sel. Areas Commun. **24**(12), 2327–2339 (2006)
9. Lehmann, E., Casella, G.: Theory of Point Estimation. Springer, New York (1998)

10. Li, T., Chen, S., Ling, Y.: Fast and compact per-flow traffic measurement through randomized counter sharing. In: Proceedings of IEEE INFOCOM, pp. 1799–1807 (2011)
11. Li, T., Chen, S., Ling, Y.: Per-flow traffic measurement through randomized counter sharing. IEEE/ACM Trans. Netw. **20**(5), 1622–1634 (2012)
12. Lu, Y., Prabhakar, B.: Robust counting via Counter Braids: an error-resilient network measurement architecture. In: Proceedings of IEEE INFOCOM (2009)
13. Lu, Y., Montanari, A., Prabhakar, B., Dharmapurikar, S., Kabbani, A.: Counter Braids: a novel counter architecture for per-flow measurement. In: Proceedings of ACM SIGMETRICS (2008)
14. Mikians, J., Dhamdhere, A., Dovrolis, C., Barlet-Ros, P., Solé-Pareta, J.: Towards a statistical characterization of the interdomain traffic matrix, pp. In: Networking 2012, pp. 111–123. Springer, Berlin (2012)
15. Ramabhadran, S., Varghese, G.: Efficient implementation of a statistics counter architecture. In: ACM SIGMETRICS Performance Evaluation Review, vol. 31, pp. 261–271 (2003)
16. Shah, D., Iyer, S., Prabhakar, B., McKeown, N.: Analysis of a statistics counter architecture. In: Hot Interconnects 9, 2001, pp. 107–111. IEEE, New York (2001)
17. Zhao, Q., Xu, J., Liu, Z.: Design of a novel statistics counter architecture with optimal space and time efficiency. In: ACM SIGMETRICS Performance Evaluation Review, vol. 34(1), pp. 323–334 (2006)

Chapter 3
Per-Flow Cardinality Measurement

Per-flow cardinality measurement 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 chapter, 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 show that sharing at the register (multi-bit) level is superior than sharing at the bit level. We present 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. Experimental results 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.

3.1 Problem Statement

Per-flow cardinality estimation is one of the fundamental problems in network traffic measurement [6, 7, 10, 12, 20, 21]. 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 any user-defined 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.

We emphasize that per-flow cardinality estimation is different from the problem of per-flow size measurement given in Chap. 2. Consider all packets with a ceratin 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.

3.2 Prior Art

3.2.1 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 [17]: 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 $\hat{n} = -b \ln V$ [17], 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. Figure 3.1 shows the simulation results, where the bitmap size is 1280 bits per flow in Fig. 3.1a, 96 bits per flow in Fig. 3.1b, and 32 bits per flow in Fig. 3.1c, 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. Figure 3.1a clearly shows a limited estimation range. As the bitmap size shrinks, the range shrinks quickly, as shown in Fig. 3.1b–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 [18–21].

3.2.2 MultiResolutionBitmap and PCSA

Sampling is one of the main methods in the literature for dealing with the estimation range problem. MultiResolutionBitmap [7] is essentially the concatenation of multiple bitmaps, each having a different sampling probability. If we let the

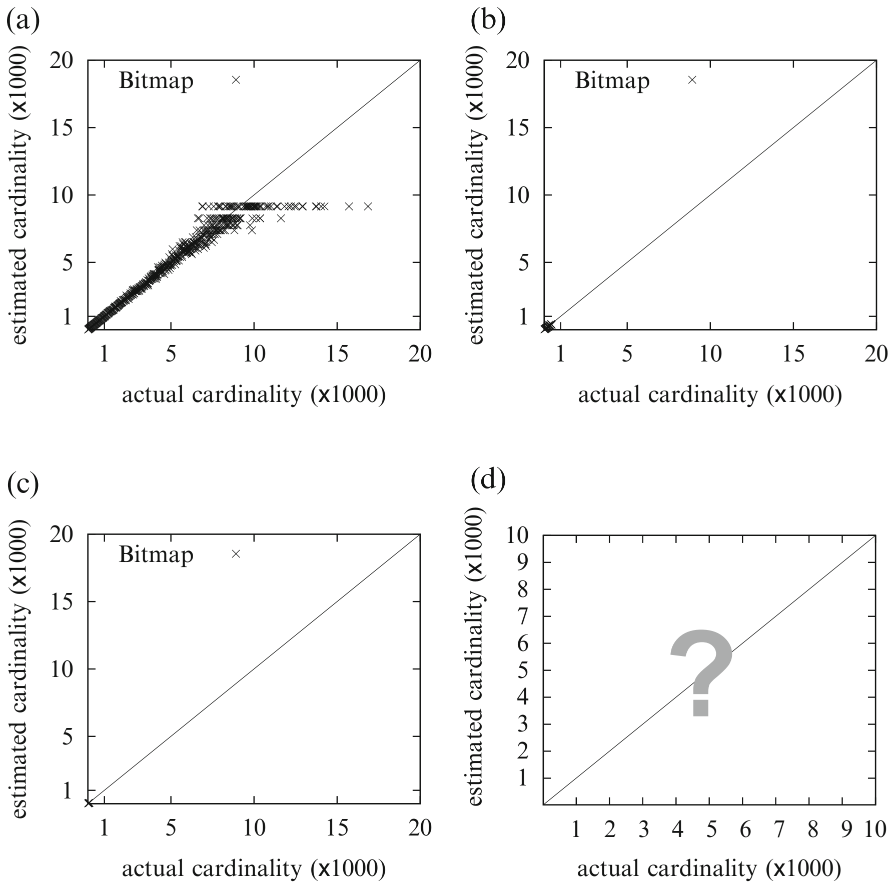


Fig. 3.1 Measurement results of the bitmap approach, whose estimation range is limited. Each flow is represented by one point. The *x*-coordinate is the true cardinality, and the *y*-coordinate is the estimated cardinality. The closer a point is to the equality line, the more accurate the estimation is. (**a**) 1280 bits per flow. (**b**) 96 bits per flow. (**c**) 32 bits per flow. (**d**) less than 1 bit per flow

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 [8]. 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. Figure 3.2 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 Fig. 3.2a. The estimation results are discrete when there are just a few registers in Fig. 3.2b, c.

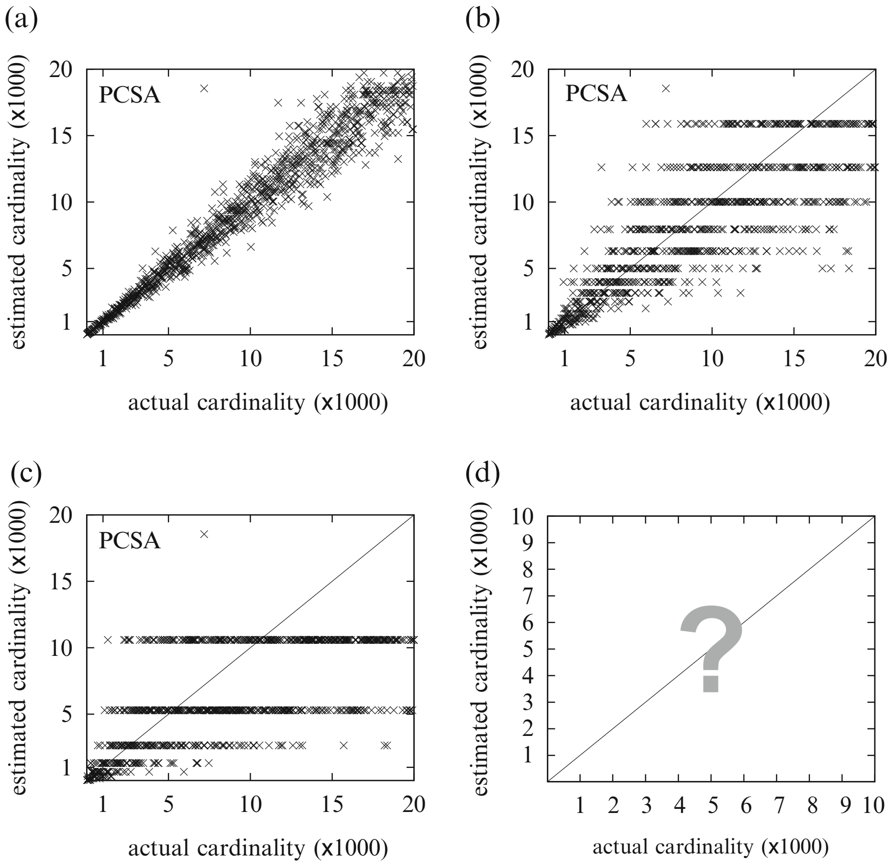


Fig. 3.2 Measurement results of FM or PCSA. (**a**) 1280 bits per flow, 40 registers of 32 bits each, 13 % error. (**b**) 96 bits per flow, 3 registers of 32 bits each. (**c**) 32 bits per flow, 1 register of 32 bits. (**d**) less than 1 bit per flow

3.2.3 LogLog and HyperLogLog

LogLog [5] and HyperLogLog [9] were designed to compress the size of each register from 32 bits to 5 bits for the same estimation range of 2^{32}. Their performance is presented in Figs. 3.3 and 3.4. 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 Figs. 3.3b and 3.4b (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.

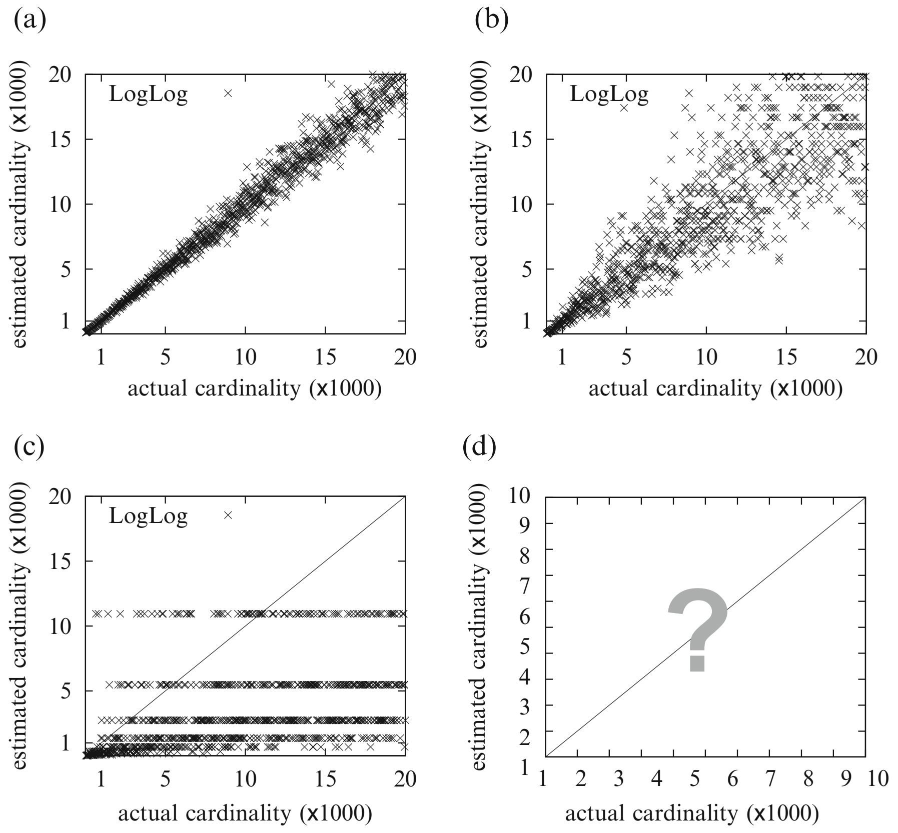


Fig. 3.3 Measurement results of LogLog. (**a**) 1280 bits per flow, 256 registers of 5 bits each, 8.1 % error. (**b**) 80 bits per flow, 16 registers of 5 bits each, 33 % error. (**c**) 5 bits per flow, 1 register of 5 bits. (**d**) less than 1 bit per flow

3.2.4 Performance Summary

The performance of the traditional cardinality estimators is summarized in Table 3.1, where MinCount [1, 2] 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).

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,

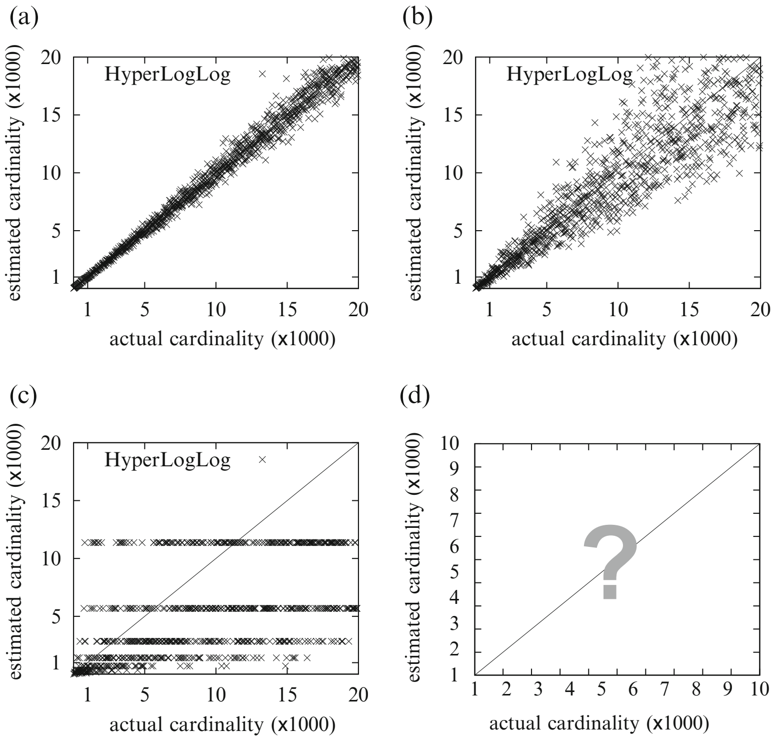


Fig. 3.4 Measurement results of HyperLogLog. (**a**) 1280 bits per flow, 256 registers of 5 bits each, 6.5 % error. (**b**) 80 bits per flow, 16 registers of 5 bits each, 26 % error. (**c**) 5 bits per flow, 1 register of 5 bits. (**d**) less than 1 bit per flow

Table 3.1 Comparison of the prior art

Solution	Std. Err. (σ)	Mem units	Mem (σ=5 %)
MinCount	$1.00/\sqrt{m}$	≤32 bits	1600 bytes [a]
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

[a] 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

the seminal work of LogLog cuts the memory requirement by more than half. The follow-up HyperLogLog cuts the memory further by more than 30 %. HyperLogLog has made great impact on IT industry and was adopted by Google [10], PostgreSQL, file-sharing P2P systems [15], and DDoS attack detection systems [9].

3.3 Register Sharing and Virtual Estimators

3.3.1 Motivation

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) [16]. 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. 3.5, where the measurement period is 10 min 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.

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.

3.3.2 Sharing at Bit Level?

One way to make use of unused bits is to share bits among the estimators. Two solutions were designed for sharing among bitmaps [20] and FM sketches [12]. In the compact spread estimator (CSE) [20], 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.

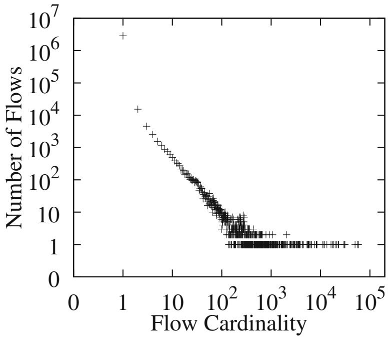


Fig. 3.5 Flow distribution: each point shows the number (y-coordinate) of flows having a certain cardinality (x-coordinate). The average cardinality of all flows is about two

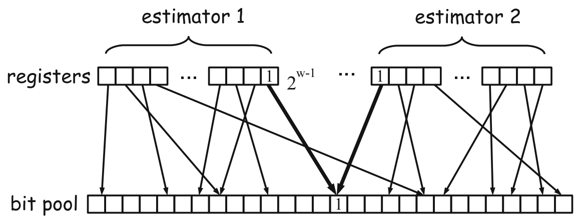


Fig. 3.6 Bit sharing in [12], where the FM sketches (registers) share their individual bits from a common bit pool

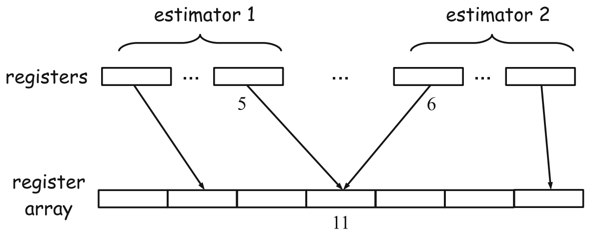


Fig. 3.7 Register sharing, where the estimators share their registers from a common register pool

In the probabilistic multiplicity counting solution (PMC) [12], 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 Fig. 3.6, 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 [12], 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 Fig. 3.6, 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.

3.3.3 Register Sharing and Virtual Estimators

Our idea is to share at the register level, as illustrated in Fig. 3.7. 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 Fig. 3.7 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. 3.5, for example. That means the number of registers that carry large noise accounts 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 [5, 9]). On the contrary, for PMC, all bits that are set to ones in V can cause large noise.

3.3.4 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 [3, 11, 13]. Take CountMin [3] 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.

3.4 A Framework for Virtual-Estimator Solutions

We now present a unified framework for developing virtual-estimator solutions that enable register-level sharing for mainstream sketches such as PCSA [8], LogLog [5], and HyperLogLog [9]. 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 3.2 for quick reference.

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 $M[i]$, $0 \le i < m$. The size of the registers is set based on the type of estimators used [5, 8, 9] 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 $M_f[i]$, $0 \le i < s$, is selected from M as follows:

$$M_f[i] = M[H_i(f)], \tag{3.1}$$

Table 3.2 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

where $H_i(\ldots)$ 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, $0 \le i < s$, can be implemented from a master function $H(\ldots)$ as follows:

$$\begin{aligned} H_i(f) &= H(f \oplus R[i]) \quad \text{or} \\ H_i(f) &= H(f \mid i), \end{aligned} \tag{3.2}$$

where "|" is the concatenation operator, "⊕" 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 [14].

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 [5, 8] or [9], 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 [5, 8] or [9] (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,

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

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 $Bino(n - n_f, \frac{1}{m})$.

$$E(n_s - n_f) = s\,E(Y) = s\,\frac{n - n_f}{m} \tag{3.3}$$

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

$$\begin{aligned} n_s - n_f &\approx \frac{n - n_f}{m} s \\ n_f &\approx \frac{ms}{m - s}\left(\frac{n_s}{s} - \frac{n}{m}\right). \end{aligned} \tag{3.4}$$

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 3.1), 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 (3.4):

$$\hat{n_f} = \frac{ms}{m - s} \cdot \left(\frac{\hat{n_s}}{s} - \frac{\hat{n}}{m}\right) \tag{3.5}$$

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

3.5 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.

3.5.1 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:

$$\begin{aligned} H(e) &= \langle x_1 x_2 \ldots \rangle \\ p &= \langle x_1 x_2 \ldots x_b \rangle \\ q &= \langle x_{b+1} x_{b+2}, \ldots \rangle, \end{aligned} \tag{3.6}$$

where $\langle x_1 x_2 \ldots \rangle$ 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 $q = 001\ldots$, 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,

$$M_f[p] := \max\big(M_f[p],\ \rho(q)\big), \tag{3.7}$$

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 (3.1), and $H_p(f) = H(f \mid p)$ as in (3.2). Combining (3.7), (3.1), and (3.2), we have

$$M[H(f \mid p)] := \max\big(M[H(f \mid p)],\ \rho(q)\big). \tag{3.8}$$

This assignment requires two hash operations: $H(e)$ for p and q in (3.6), and $H(f \mid p)$. It also requires at most two memory accesses, reading $M[H(f \mid p)]$ and writing $M[H(f \mid p)]$ back if its value changes. Note that the writing operation happens rarely since the likelihood for $\rho(q) > M[H(f \mid p)]$ to happen will decrease exponentially as the register's value increases.

Equation (3.8) shows that the operations are actually performed on the physical register array M, and the virtual estimator is logical in the online recording phase.

3.5.2 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 $2^{M_f[i]}$ [9].

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:

$$\hat{n_s} = \alpha_s \cdot s^2 \cdot \Big(\sum_{j=0}^{s-1} 2^{-M_f[j]}\Big)^{-1}, \tag{3.9}$$

where α_s is a bias correction constant that equals

$$\alpha_s = \Big(s \int_0^\infty \Big(\log_2\Big(\frac{2+u}{1+u}\Big)\Big)^m du\Big)^{-1}. \tag{3.10}$$

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 $s \geq 128$.

The estimator in (3.9) is good for large cardinalities, but it is severely biased when dealing with small cardinalities [9]. 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 [17]. This formula is used when the cardinality estimation by (3.9) 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 (3.10) 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 (3.5) to compute the estimated flow cardinality $\hat{n_f}$.

3.6 Estimation Bias and Variance

This section analyzes the bias and standard error of our vHLL estimator. From [9], we have the following theorem:

Theorem 2. *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 (3.9) 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 [9], the functions δ_1 *and* δ_2 *represent oscillating functions of a tiny amplitude, and they can be safely neglected for all practical purposes.*

3.6.1 Estimation Bias

Given an arbitrary flow f, we know from Sect. 3.4 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

$$Prob\{n_s - n_f = i\} = \binom{n - n_f}{i}(\frac{s}{m})^i(1 - \frac{s}{m})^{n-n_f-i}. \tag{3.11}$$

Under the condition of $n_s - n_f = i$, by Theorem 2, we have

$$\begin{aligned} E(\hat{n_s} \mid n_s - n_f = i) &= (n_f + i)\big(1 + \delta_1(n_f + i) + o(1)\big) \\ &\approx n_f + i, \end{aligned} \tag{3.12}$$

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

$$\begin{aligned} E(\hat{n_s}) &= \sum_{i=0}^{n-n_f} E(\hat{n_s} \mid n_s - n_f = i) \times Prob\{n_s - n_f = i\} \\ &\approx \sum_{i=0}^{n-n_f} (n_f + i) \times \binom{n - n_f}{i}(\frac{s}{m})^i(1 - \frac{s}{m})^{n-n_f-i} \\ &= n_f + (n - n_f)\frac{s}{m}. \end{aligned} \tag{3.13}$$

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 2, 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 (3.5), we have

$$
\begin{aligned}
rClE(\hat{n_f}) &= \frac{ms}{m-s}\Big(\frac{E(\hat{n_s})}{s} - \frac{E(\hat{n})}{m}\Big)\\
&\approx \frac{ms}{m-s}\Big(\frac{n_f + (n-n_f)\frac{s}{m}}{s} - \frac{n}{m}\Big) = n_f. \qquad (3.14)
\end{aligned}
$$

Therefore, the vHLL estimator is approximately unbiased.

3.6.2 Estimation Variance

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

$$
\begin{aligned}
Var(\hat{n_f}) &= \big(\frac{ms}{m-s}\big)^2\Big(\frac{Var(\hat{n_s})}{s^2} + \frac{Var(\hat{n})}{m^2}\Big) \qquad (3.15)\\
&= \big(\frac{ms}{m-s}\big)^2\Big(\frac{E(\hat{n_s}^2) - \big(E(\hat{n_s})\big)^2}{s^2} + \frac{Var(\hat{n})}{m^2}\Big)\\
&= \big(\frac{m}{m-s}\big)^2\Big(E(\hat{n_s}^2) - \big(E(\hat{n_s})\big)^2 + (\frac{s}{m})^2 Var(\hat{n})\Big)
\end{aligned}
$$

With $\forall i \in [0, n-n_f)$, under the condition of $n_s - n_f = i$, by Theorem 2, we have

$$
\begin{aligned}
\tfrac{1}{n_f+i}\sqrt{Var(\hat{n_s}\,|\,n_s - n_f = i)} &= \tfrac{\beta_s}{\sqrt{s}} + \delta_2(n_f+i) + o(1)\\
&= \frac{\beta_s}{\sqrt{s}} \approx \frac{1.04}{\sqrt{s}}, \qquad (3.16)
\end{aligned}
$$

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

$$
Var(\hat{n_s}\,|\,n_s - n_f = i) \approx \frac{1.04^2}{s}(n_f+i)^2. \qquad (3.17)
$$

Similarly, we have

$$
Var(\hat{n}) \approx \frac{1.04^2}{m}n^2, \qquad (3.18)
$$

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 (3.12) and (3.17), when s is sufficiently large, we have

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

Combining (3.11) with the above equation, we have

$$
\begin{aligned}
E(\hat{n_s}^2) &= \sum_{i=0}^{n-n_f} E(\hat{n_s}^2 \mid n_s - n_f = i) Prob\{n_s - n_f = i\} \qquad (3.19)\\
&\approx \sum_{i=0}^{n-n_f} (\frac{1.04^2}{s} + 1)(n_f + i)^2 \binom{n-n_f}{i} (\frac{s}{m})^i (1 - \frac{s}{m})^{n-n_f-i}\\
&= (\frac{1.04^2}{s} + 1)\big(n_f^2 + 2n_f E(n_s - n_f) + E((n_s - n_f)^2)\big)\\
&= (\frac{1.04^2}{s} + 1)\Big(n_f^2 + 2n_f(n-n_f)\frac{s}{m}\\
&\qquad + (n-n_f)\frac{s}{m}(1-\frac{s}{m}) + (n-n_f)^2(\frac{s}{m})^2\Big)\\
&= (\frac{1.04^2}{s} + 1)\Big((n_f + (n-n_f)\frac{s}{m})^2 + (n-n_f)\frac{s}{m}(1-\frac{s}{m})\Big).
\end{aligned}
$$

Applying (3.13), (3.18) and (3.19) to (3.15), we have

$$
\begin{aligned}
Var(\hat{n_f}) \approx (\frac{m}{m-s})^2 \Big(&\frac{1.04^2}{s}\big(n_f + (n-n_f)\frac{s}{m}\big)^2\\
&+ (n-n_f)\frac{s}{m}(1-\frac{s}{m}) + (\frac{s}{m})^2 \frac{1.04^2}{m} n^2\Big). \qquad (3.20)
\end{aligned}
$$

Consider the three terms between the parentheses after $\big(\frac{m}{m-s}\big)^2$. 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 (3.3) 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.

3.6.3 Relative Standard Error

We define the relative standard error as

$$
StdErr(\frac{\hat{n_f}}{n_f}) = \frac{\sqrt{Var(\hat{n_f})}}{n_f}. \qquad (3.21)
$$

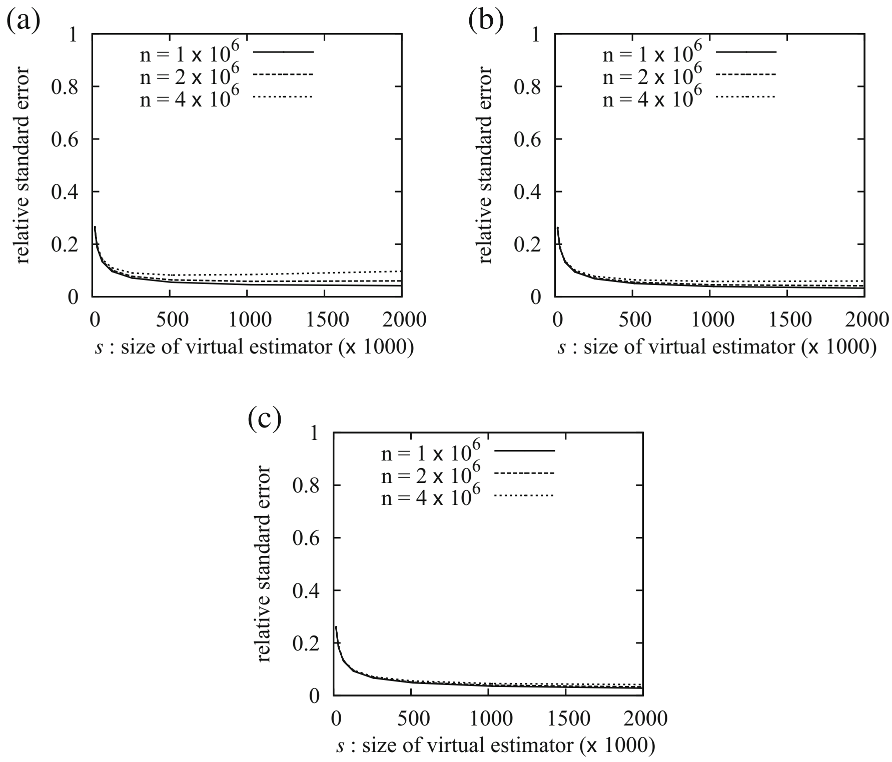


Fig. 3.8 Relative standard error with respect to s, n, and n_f. (**a**) Flow cardinality n_f=1 × 10^4. (**b**) Flow cardinality n_f=2 × 10^4. (**c**) Flow cardinality n_f=4 × 10^4

From (3.20) and (3.21), 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 3.8a shows the numerically computed estimation error by (3.21) 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 2. 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 2 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 Fig. 3.8a–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.

3.7 Experimental Evaluation

We have implemented the vHLL solution and the most related work PMC [12]. 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 [16]. The traces are captured by a high-speed monitor named equinix-sanjose (located in San Jose, CA, USA), which is connected to a 10-Gbit/s Ethernet backbone link. Each trace file captures the packets in 1 min. 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 3.3.

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 Fig. 3.5. 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 [4, 22].

3.7.1 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 vHLL, we configure the

Table 3.3 Trace statistics

Time (min)	Num. of flows	Total cardinality	Mean flow cardinality
1–10	1,473,306	2,675,506	1.8
11–20	1,013,517	1,856,676	1.8
21–30	1,648,779	3,005,649	1.8
31–40	1,562,288	2,881,330	1.8
41–50	1,612,709	3,280,242	2.0
51–60	1,612,605	3,280,138	2.0

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 Figs. 3.9, 3.10, 3.11, and 3.12, 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 Fig. 3.9, 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

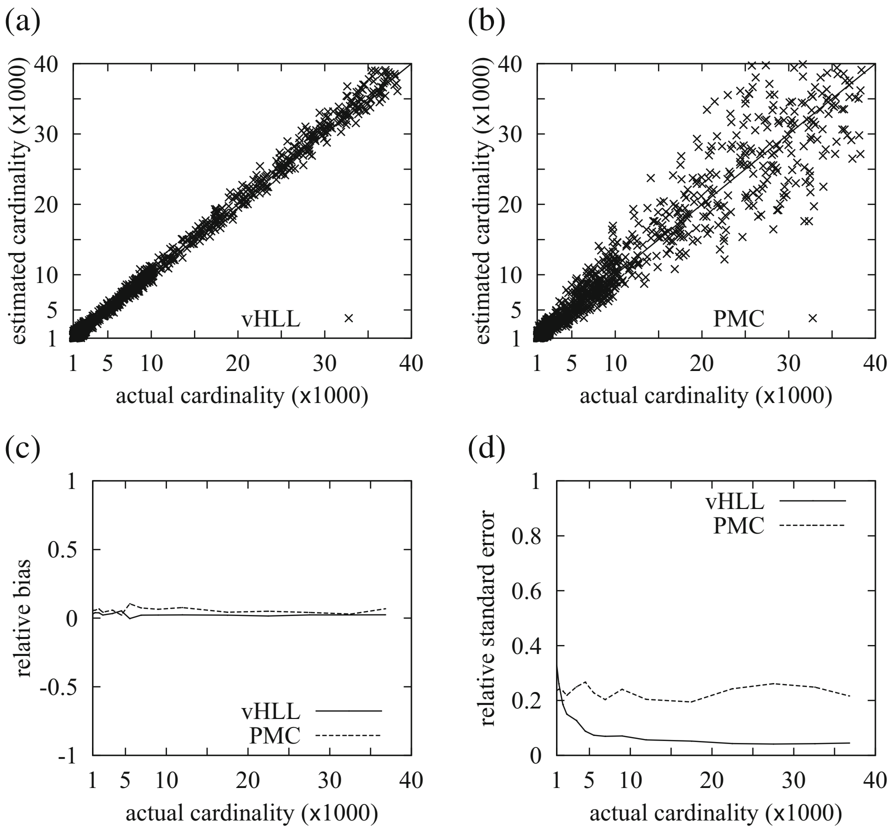


Fig. 3.9 Compare vHLL and PMC with 1 bit per flow. (**a**) vHLL with 1 bit per flow. (**b**) PMC with 1 bit per flow. (**c**) Estimation bias. (**d**) Estimation accuracy

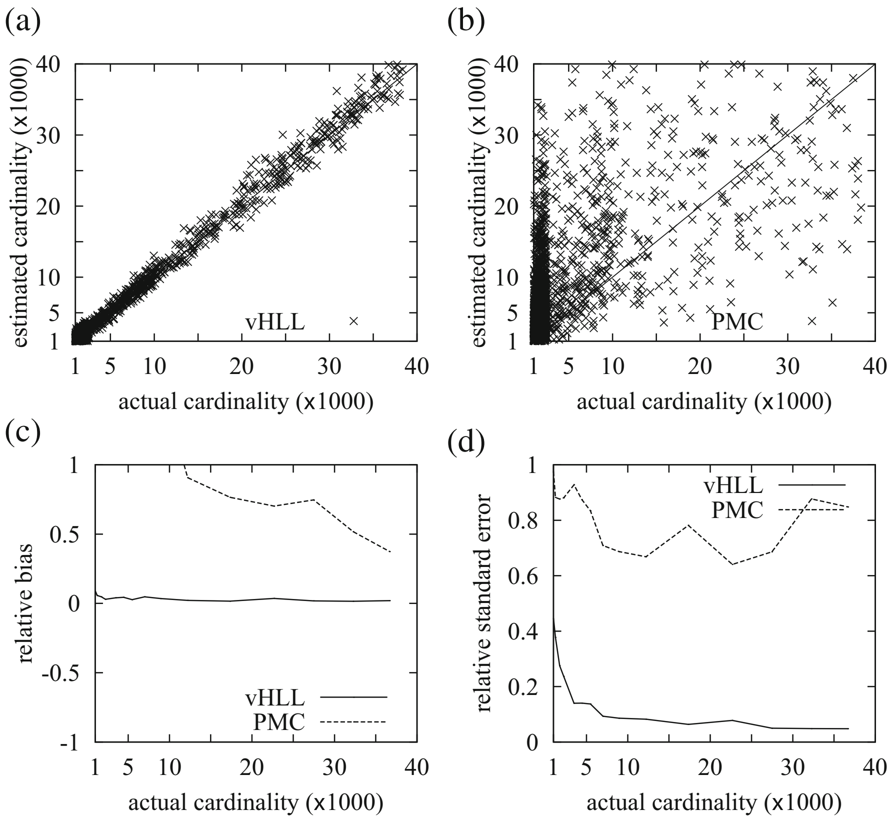


Fig. 3.10 Compare vHLL and PMC with 0.5 bit per flow. (**a**) vHLL with 0.5 bit per flow. (**b**) PMC with 0.5 bit per flow. (**c**) Estimation bias. (**d**) Estimation accuracy

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 [12], 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 10,000, 20,000,

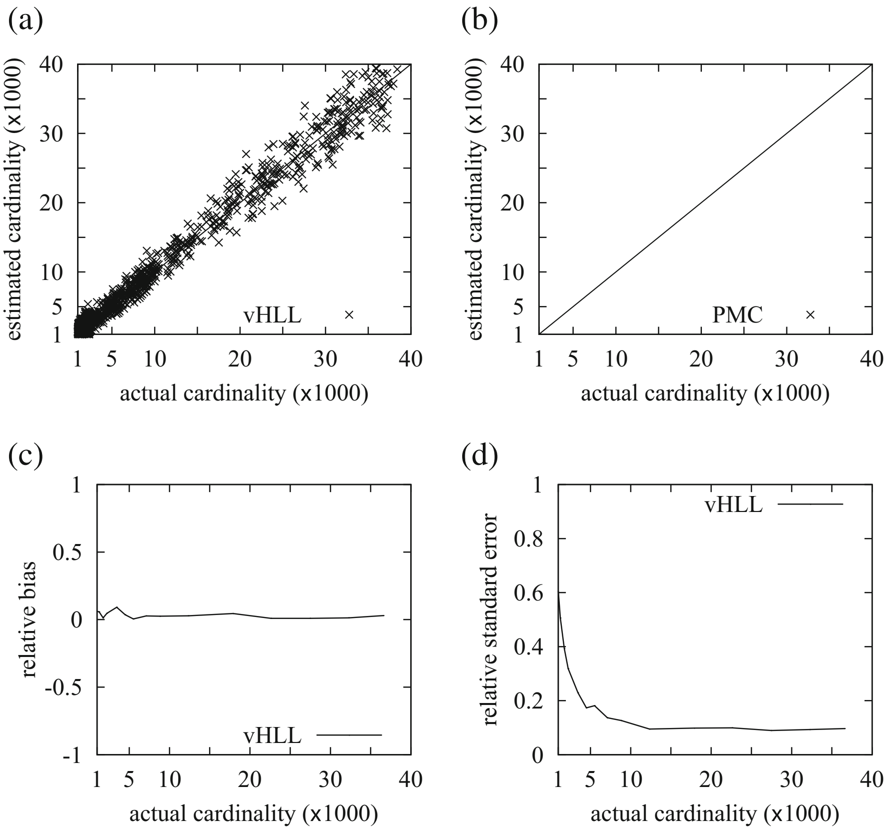


Fig. 3.11 Compare vHLL and PMC with 0.25 bit per flow. (**a**) vHLL with 0.25 bit per flow. (**b**) PMC with 0.25 bit per flow. (**c**) Estimation bias. (**d**) Estimation accuracy

and 30,000, 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 3.10 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 10,000, 20,000, and 30,000, 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, Figs. 3.11 and 3.12 show that vHLL still works with gradually deteriorating accuracy. For 0.25 bit per flow, when the actual cardinality is 10,000, 20,000, and 30,000, 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 10,000, 20,000, and 30,000, 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

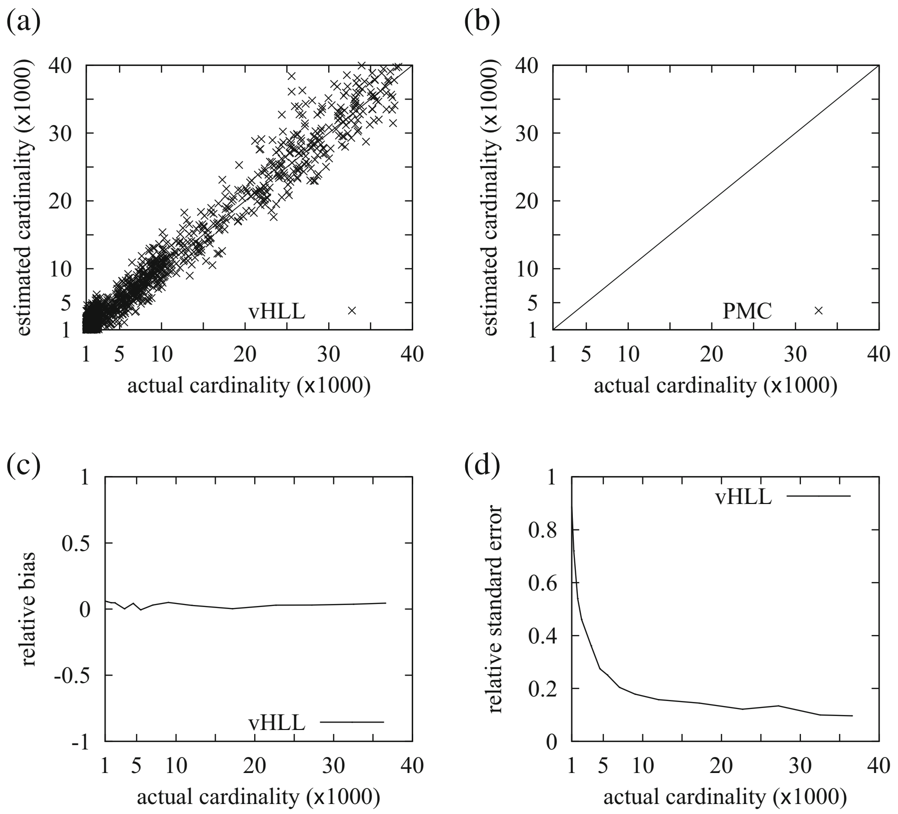
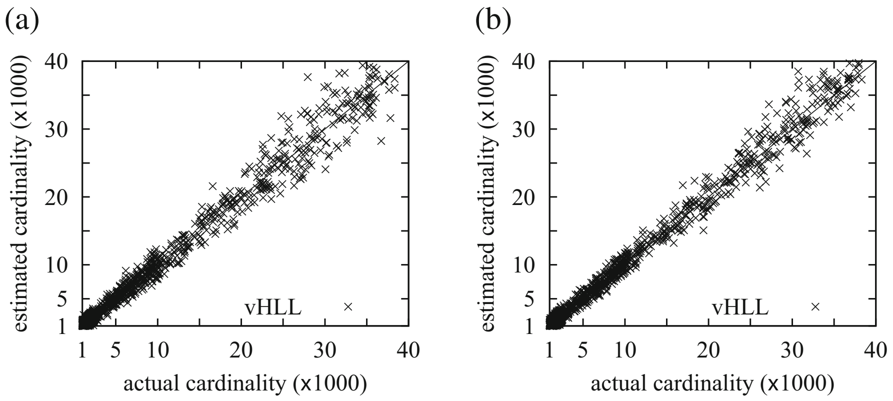


Fig. 3.12 Compare vHLL and PMC with 0.1 bit per flow. (**a**) vHLL with 0.1 bit per flow. (**b**) PMC with 0.1 bit per flow. (**c**) Estimation bias. (**d**) Estimation accuracy

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 mistake a small flow for a large one due to the modest absolute error.

3.7.2 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 Fig. 3.10a 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. 3.13a–c, respectively. Corresponding relative standard errors are shown in Fig. 3.14a–c, respectively.



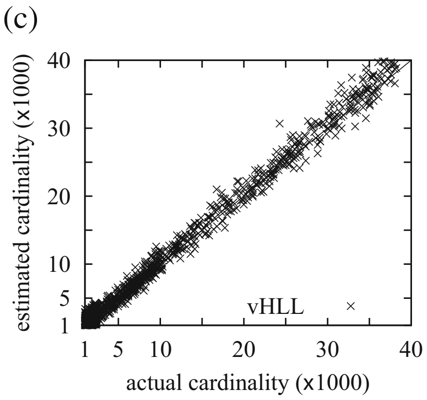


Fig. 3.13 Cardinality estimation with different values of s under average memory of 0.5 bit per flow. (**a**) $s = 128$. (**b**) $s = 256$. (**c**) $s = 1024$

We observe that when s is relatively small at 128, the estimation accuracy in Fig. 3.13a is noticeably worse than that in Fig. 3.10a, which is evident from the fact that the points of the former surround the equality line less tightly. Quantitatively, the errors in Fig. 3.14a with $s = 128$ are larger than those in Fig. 3.14d for vHLL with the default $s = 512$. For example, when the actual cardinality is 20,000, 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 Figs. 3.13b, c and 3.10a, whose s values are 256, 512, and 1024, respectively. For example, when the actual cardinality is 20,000, their errors are 8.1 %, 6.5 %, and 5.2 %, based on from Figs. 3.14b, c, and 3.10d, respectively.

The above observations are consistent with our analysis in Sect. 3.6 and the numerical results in Fig. 3.8 (which has different parameters though). The reasons for these observations have been explained in Sect. 3.6.3 and will not be repeated here.

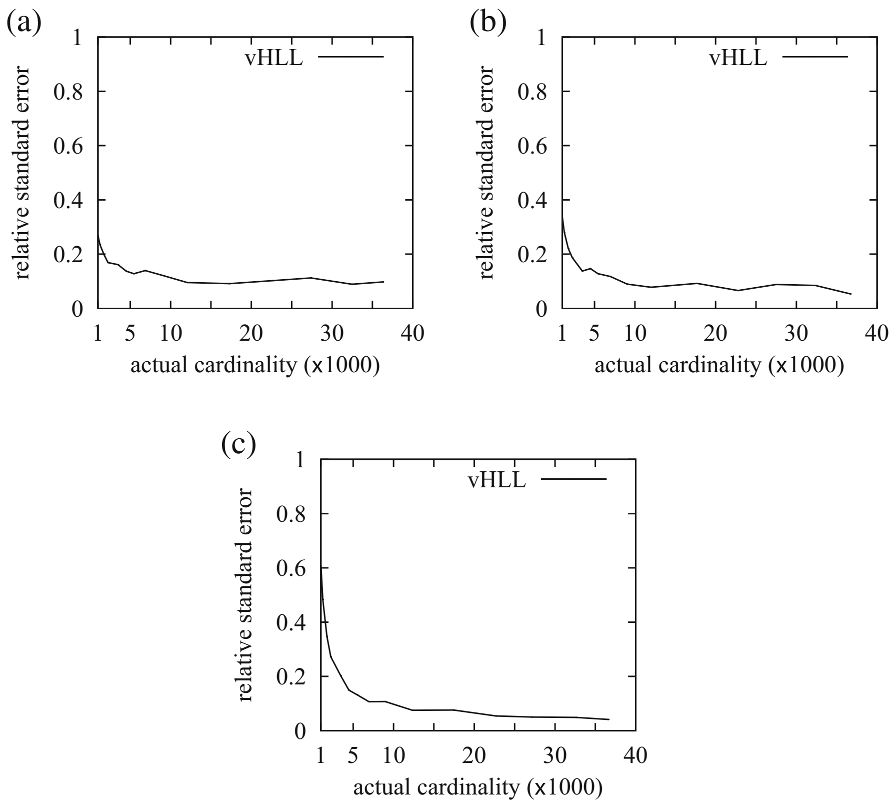


Fig. 3.14 Relative standard errors of cardinality estimation with different values of s under average memory of 0.5 bit per flow. (**a**) $s = 128$. (**b**) $s = 256$. (**c**) $s = 1024$

3.7.3 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 Fig. 3.10a with average memory of 0.5 bit per flow. The results are presented in Fig. 3.15, 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

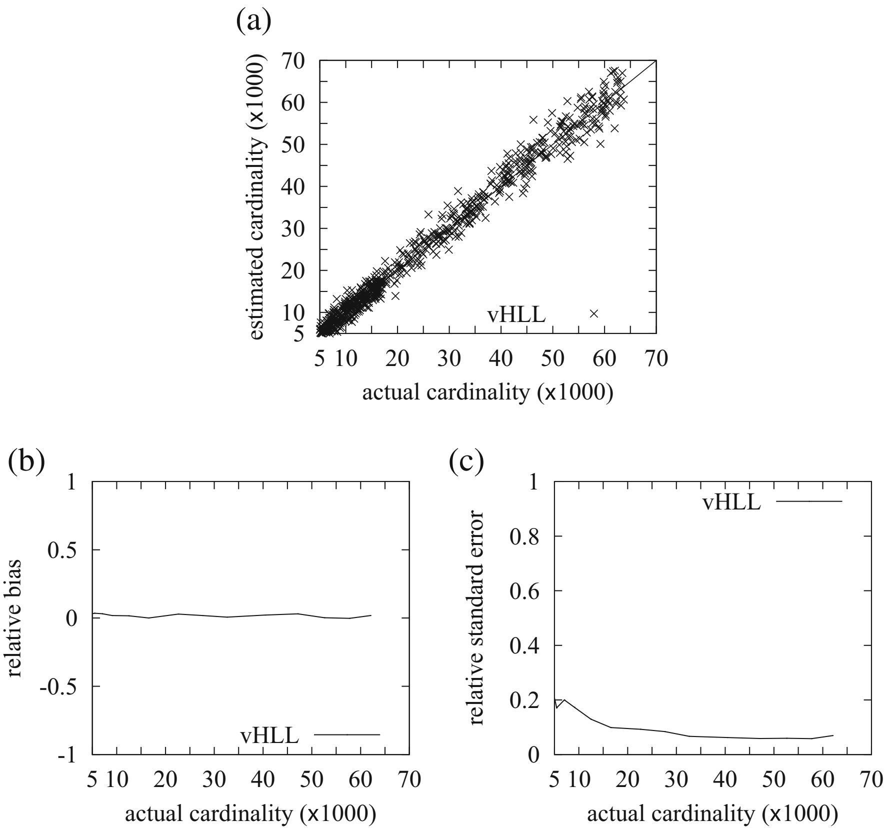


Fig. 3.15 Cardinality estimation with n doubled under average memory of 0.5 bits per flow. (**a**) n is doubled. (**b**) Estimation bias. (**c**) Estimation accuracy

for large flows. The error is modest, but larger than that in Fig. 3.10d 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 Fig. 3.16. Again the bias is close to one, but the error increases.

3.7.4 A Case Study: Detect Super Destinations

Our last set of experiments compare vHLL and PMC based on a hypothetical application for detecting the so-called super destinations. In this case study, we consider per-destination flows and measure the number of distinct sources that

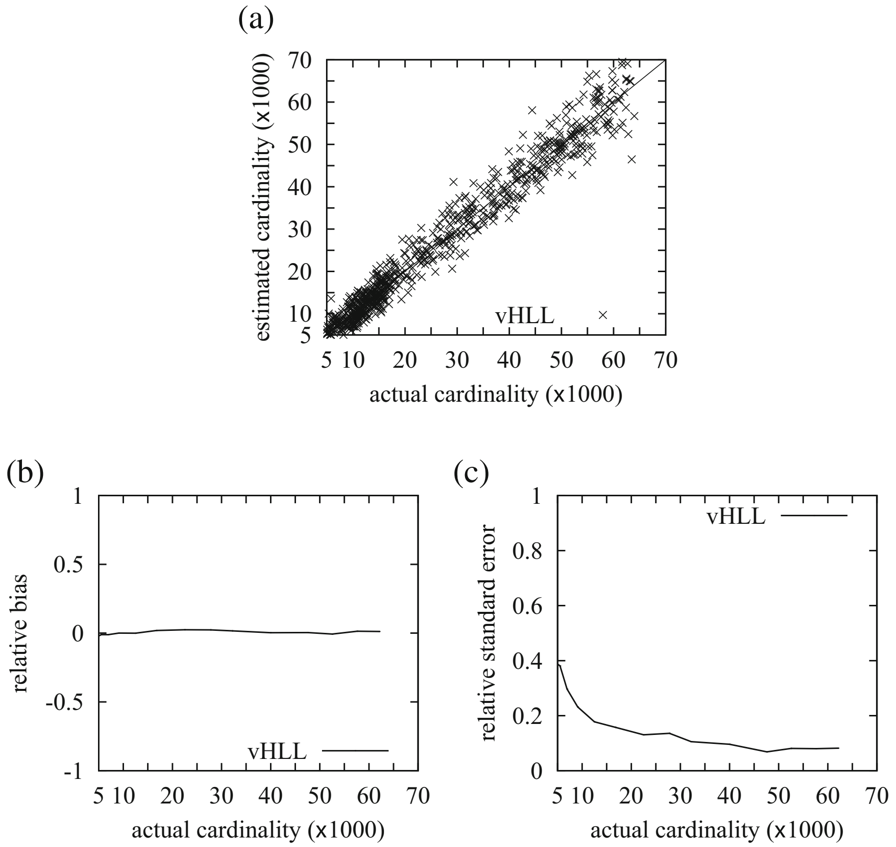


Fig. 3.16 Cardinality estimation with *n* increased four folds under average memory of 0.5 bits per flow. (**a**) *n* is increased by four folds. (**b**) Estimation bias. (**c**) Estimation accuracy

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 5000 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 5000 is reported, it is called a *false positive*. If a destination with a cardinality 5000 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 5000 or more.

Table 3.4 False positive ratio and false negative ratio with respect to memory cost

	PMC		vHLL	
Memory (bit per flow)	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 3.5 $\epsilon = 10\,\%$, false positive ratio and false negative ratio with respect to memory cost

	PMC		vHLL	
Memory (bit per flow)	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 3.6 $\epsilon = 20\,\%$, false positive ratio and false negative ratio with respect to memory cost

	PMC		vHLL	
Memory (bit per flow)	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

The experimental results are shown in Table 3.4. 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. 3.10b. Its FPR is 73.7 % for 0.5 bit per flow and 99.2 % for 0.25 bit per flow.

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 5000 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 3.5, and those for $\epsilon = 20\,\%$ are shown in Table 3.6, 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 3.6, when the memory grows to at least 0.5 bit per flow, FPR and FNR for vHLL become zeros.

3.8 Summary

In this chapter, we present 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.

References

1. Bar-yossef, Z., Jayram, T.S., Kumar, R., Sivakumar, D., Trevisan, L., Luca: Counting distinct elements in a data stream. In: Proceedings of the RANDOM: Workshop on Randomization and Approximation (2002)
2. Beyer, K., Haas, P.J., Reinwald, B., Sismanis, Y., Gemulla, R.: On synopses for distinct-value estimation under multiset operations. In: Proceedings of the ACM SIGMOD (2007)
3. Cormode, G., Muthukrishnan, S.: An improved data stream summary: the Count-Min sketch and its applications. In: Proceedings of the LATIN (2004)
4. Costa, M., Crowcroft, J., Castro, M., Rowstron, A., Zhou, L., Zhang, L., Barham, P.: Vigilante: end-to-end containment of internet worms. SIGOPS Operat. Syst. Rev. **39**(5), 133–147 (2005)
5. Durand, M., Flajolet, P.: Loglog counting of large cardinalities. In: ESA: European Symposia on Algorithms, pp. 605–617 (2003)
6. Estan, C., Varghese, G.: New directions in traffic measurement and accounting. In: Proceedings of the ACM SIGCOMM (2002)
7. Estan, C., Varghese, G., Fish, M.: Bitmap algorithms for counting active flows on high-speed links. IEEE/ACM Trans. Netw. **14**(5), 925–937 (2006)
8. Flajolet, P., Martin, G.N.: Probabilistic counting algorithms for database applications. J. Comput. Syst. Sci. **31**(2), 182–209 (1985)
9. Flajolet, P., Fusy, E., Gandouet, O., Meunier., F.: HyperLogLog: the analysis of a near-optimal cardinality estimation algorithm. In: Proceedings of the AOFA: International Conference on Analysis of Algorithms (2007)
10. Heule, S., Nunkesser, M., Hall, A.: HyperLogLog in practice: algorithmic engineering of a state-of-the-art cardinality estimation algorithm. In: Proceedings of the EDBT (2013)
11. Li, T., Chen, S., Ling, Y.: Fast and compact per-flow traffic measurement through randomized counter sharing. In: Proceedings of the IEEE INFOCOM, pp. 1799–1807 (2011)
12. Lieven, P., Scheuermann, B.: High-speed per-flow traffic measurement with probabilistic multiplicity counting. In: Proceedings of IEEE INFOCOM, pp. 1–9 (2010). doi:10.1109/INFCOM.2010.5461921
13. Lu, Y., Montanari, A., Prabhakar, B., Dharmapurikar, S., Kabbani, A.: Counter braids: a novel counter architecture for per-flow measurement. In: Proceedings of ACM SIGMETRICS (2008)
14. Neustar.biz: How to choose a good hash function: part 3. (2012) http://research.neustar.biz/2012/02/02/choosing-a-good-hash-function-part-3
15. Ntarmos, N., Triantafillou, P., Weikum, G.: Counting at large: efficient cardinality estimation in internet-scale data networks. In: Proceedings of the ICDE, pp. 40–40 (2006). doi:10.1109/ICDE.2006.44
16. The CAIDA UCSD Anonymized 2013 Internet Traces - January 17 (2013). http://www.caida.org/data/passive/passive_2013_dataset.xml
17. Whang, K.Y., Vander-Zanden, B.T., Taylor, H.M.: A linear-time probabilistic counting algorithm for database applications. ACM Trans. Database Syst. **15**(2), 208–229 (1990)

18. Xiao, Q., Xiao, B., Chen, S.: Differential estimation in dynamic RFID systems. In: Proceedings of the INFOCOM (Mini-Conference), pp. 295–299 (2013)
19. Xiao, Q., Qiao, Y., Zhen, M., Chen, S.: Estimating the persistent spreads in high-speed networks. In: Proceedings of the IEEE ICNP, pp. 131–142 (2014)
20. Yoon, M., Li, T., Chen, S., Peir, J.K.: Fit a spread estimator in small memory. In: Proceedings of the IEEE INFOCOM (2009)
21. Zhao, Q., Xu, J., Kumar, A.: Detection of super sources and destinations in high-speed networks: algorithms, analysis and evaluation. IEEE JASC **24**(10), 1840–1852 (2006)
22. Zou, C.C., Gao, L., Gong, W., Towsley, D.: Monitoring and early warning for internet worms. In: Proceedings of the 10th ACM Conference on Computer and Communications Security (2003)

Chapter 4
Persistent Spread Measurement

The persistent spread of a destination host is the number of distinct sources that have contacted it persistently in predefined t measurement periods. A persistent spread estimator is a software/hardware component in a router that inspects the arrival packets and estimates the persistent spread of each destination. This is a new primitive for network measurement that can be used to detect long-term stealthy malicious activities, which cannot be recognized by the traditional superspreader detectors that are designed only for “elephant" activities. This chapter presents an estimator that can use very tight memory space to deliver high estimation accuracy: Its memory expense is less than one bit per-flow element in each time period; Its estimation accuracy is over 90 % better than a continuous variant of Flajolet–Martin sketches; Its operating range to produce effective measurements is hundreds of times broader than the traditional bitmap. These advantages originate from a new data structure called multi-virtual bitmap, which is designed to estimate the cardinality of the intersection of an arbitrary number of sets. The effectiveness of the new estimator is verified using the real network traffic traces from CAIDA.

4.1 Problem Statement

A *persistent spread estimator* is a software/hardware module on a gateway router (or a core router) to monitor the traffic flows passing through the router. Here, a flow can be either a per-destination flow or a per-source flow. An example of a per-destination flow is illustrated in Fig. 4.1, where a server inside an intranet is contacted by a set of external hosts. All the packets sent from the external hosts to the server constitute a per-destination flow, which is inspected by the gateway.

A per-destination (source) flow is all the packets towards a common destination (source) address, and the flow elements are the source (destination) addresses in the packet stream. For a flow of interest, let S_i be the set of flow elements observed

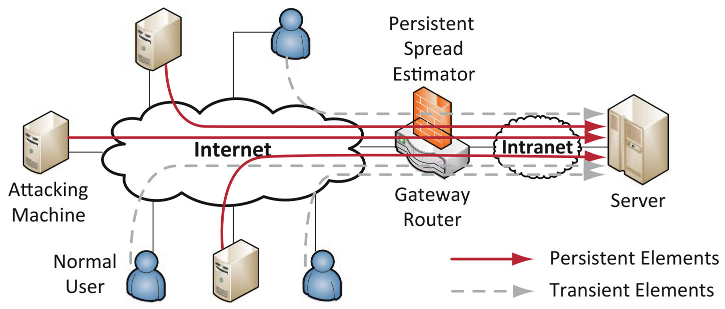


Fig. 4.1 Persistent spreads can help detect stealthy DDoS attacks

by the router in the ith measurement period. These elements can be divided into two subsets. (1) The elements in the set $S^* = S_1 \cap S_2 \ldots \cap S_t$ are called the *persistent elements*, which stay in the flow for t consecutive periods, and t is a system parameter that is configurable by network administrators. (2) The elements in the set $S_i - S^*$ are called the *transient elements* in the ith period. Typically, for a transient element, its packets can be observed by the gateway router only in a few periods, which is similar to a normal user finishing his/her online transaction within one or two periods. We do not deny the possibility that a small proportion of users are heavy users that occupy more than two periods. We only need their online time to be smaller than the number of periods t, which makes them differ from the persistent elements.

The persistent spread measurement is to design an algorithm that can efficiently estimate the cardinality of persistent elements $|S^*|$ for any flow of interest, or called its persistent spread. This problem has many important applications, and we list just a few. (1) For per-destination flows, their persistent spreads can help to expose the stealthy DDoS attacks or the forging of server popularity. An example of stealthy DDoS attacks is shown in Fig. 4.1, where three compromised machines are sending requests repeatedly to the server to downgrade its performance. We want to detect the existence of these persistent attacking hosts from their number. (2) For per-source flows, their persistent spreads can help detect network scanning. Since a network scanner avoids the redundant probing of the same network segment, its persistent spread $|S^*|$ is ultra low, while the number of destination addresses $|S_i|$ it contacted in each period is considerable.

For simplicity, we have assumed t consecutive time periods and treat their intersection $S^* = S_1 \cap S_2 \ldots \cap S_t$ as persistent elements. Note that a router can also record the traffic in non-consecutive periods and define their intersection (e.g., $S_1 \cap S_3 \ldots \cap S_{2t-1}$) as persistent elements, such that the attackers cannot predict our pattern for detecting malicious activities.

A precondition for persistent spreads to be useful is that they are small in normal traffics, so that an abnormally large measurement becomes an effective indication of stealthy attacks. We verify this assumption by analyzing the real network traffic traces downloaded from CAIDA [11]. These traces are collected on January 17th,

Table 4.1 Rapid decrease of persistent spread as the growth of number of periods

HTTP Server 224.243.38.27/80						
Number of periods	**1**	**2**	**3**	**4**	**5**	**6**
Persistent spread	31255	3902	223	66	30	14
HTTP Server 50.13.250.2/80						
Number of periods	**1**	**2**	**3**	**4**	**5**	**6**
Persistent spread	50003	578	100	37	16	7
HTTPS Server 224.243.38.27/443						
Number of periods	**1**	**2**	**3**	**4**	**5**	**6**
Persistent spread	58478	6379	780	378	227	133
HTTPS Server 224.243.38.7/443						
Number of periods	**1**	**2**	**3**	**4**	**5**	**6**
Persistent spread	55355	8616	1661	685	301	142

Relation between the persistent spread and the number of periods, where each period lasts 10 min

2013 from a high-speed monitor named equinix-sanjose, connected to a 10G Ethernet backbone link. In these traces, we locate tens of server machines with large spreads in single time period. In Table 4.1, we list the traffic pattern of four such servers which are not under attacks. In the table, the length of one measurement period is configured to 10 min, and the number of periods t varies from one to six. We investigate the impact of t on the persistent spread $|S^*| = |S_1 \cap S_2 \ldots \cap S_t|$, when there are no persistent attacks.

This table shows that, in normal traffic with no attacks, the persistent spread reduces rapidly as t increases, and it becomes negligibly small when t is at least three. Take the HTTP server 224.243.38.27/80 as an example. When the number of periods grows to six (about an hour), the persistent spread decreases to 14, which can be neglected if compared with the one-period spread 31255. Such a phenomenon is not difficult to understand, since very few persons would keep browsing the same website for an hour without taking a break or switching to another website. For HTTPS server 224.243.38.7/443, a similar phenomenon can be found. When the number of periods is six, the persistent spread is 142, a larger number than HTTP servers. It shows that users are prone to stay online for longer time if using HTTPS as the communication protocol. But 142 is still a negligibly small amount if compared with 55355—the spread in one period.

We have tested other types of servers that use unfamiliar ports. We find that when contacting chat or video servers, users will stay much longer than on web servers. But the continuous online time of legitimate users is still limited. Hence, when the length of a measurement period is long enough, it can contain a typical user's continuous behavior within this period. In this case, persistent spreads will become negligibly small when the number of periods t is large enough, since the probability for a legitimate user's online time to be significantly longer than the rest is negligible. Therefore, an abnormally large persistent spread is likely to be a good indicator for stealthy attacks.

Without loss of generality, we consider a per-destination flow corresponding to the server *dst*, and we want to estimate its persistent spread $n^* = |S^*|$. We can alternate the role of source and destination, and use the same estimator to measure the persistent spread of a given source.

4.2 Preliminary Solutions

This section presents two straightforward solutions, to motivate our design based on bitwise AND of multiple bitmaps.

4.2.1 Hash Table Solutions

A naive solution is that a router records the set of source addresses S_i, $1 \leq i \leq t$, that have contacted the server *dst* in each time period. The set S_i in the *i*th period can be stored in the on-chip SRAM as a hash table. When the period ends, S_i can be downloaded to permanent storage for post-processing. With a series of such sets S_1, $S_2, \ldots, S_t$, we can calculate their intersection, which contains the persistent elements that last for t periods. The advantage of this solution is the precise calculation of persistent spread $|S_1 \cap S_2 \ldots \cap S_t|$.

However, the solution has the shortcoming of high memory cost. In the *i*th period, hash set S_i is kept in on-chip SRAM; when the period ends, S_i is offloaded to permanent storage. We only consider the cost of precious on-chip SRAM. Therefore, the memory cost for this algorithm is $O\big((32+32)\cdot\max(|S_i|)\big)$ bits, where the first 32 means the length of an IPv4 address for one flow element, the second 32 means the 32-bit pointer needed by the chained hash table for each element, and $\max(|S_i|)$ is the largest spread in each time period. Therefore, the memory cost is 64 bits per-flow element, which is quite expensive.

In most cases, the exact values of persistent spreads are not necessary, and their approximated values with bounded estimation errors can suffice the requirement of traffic measurement. To approximate a persistent spread, a method is to store the short signatures of IP addresses into a hash table. For each IP address x, its signature is a hash value $H(x)$ that is just k bits long ($k < 32$). This reduces memory cost by multiple folds as compared with the 32-bits IPv4 or 128-bits IPv6 addresses.

However, the enhancement by partial signatures owns two inadequacies. Firstly, it is prone to underestimate the persistent spreads: When two persistent elements are mapped to the same hash bucket and are encoded by the same signature, they will counted as one element. Secondly, its memory cost is still $O\big((k + 32) \cdot \max(|S_i|)\big)$, where k is the length of a partial signature which can be 4 or 8 bits, and 32 is the length of a pointer needed by chained hash table. Hence, the memory cost is still $k + 32$ bits per-flow element in one period. Our vision is to reduce memory cost to less than one bit per element, and with such limited space, still render satisfactory accuracy.

4.2.2 Bitmap-Based Method

Another solution is to adopt *bitmap* algorithm [12] for persistent spread estimation. Let B be a bit array allocated for the flow *dst*, called a bitmap. Its ith bit is denoted by $B[i]$, $0 \leq i < m$. Its number of bits m is configured on the scale of $\max(|S_i|)$.

At the beginning of the ith period, all the bits of array B are initialized to zero. When the router receives a packet $\langle src, dst \rangle$ that is destined to the server *dst*, it categorizes the packet to the flow *dst*, and maps the source address *src* to the flow's bitmap B to record the flow element. The hash function $H(src)$ decides which bit will be set in B.

$$B[H(src) \bmod m] := 1 \tag{4.1}$$

Here, := is the assignment operator, and the hash function H is implemented by MurMur3 hashing. Note that this bitmap structure is "duplicate-insensitive," i.e., duplicated addresses will set the same bit and be filtered.

At the end of the ith period, the router has recorded in the bitmap B all the source addresses that have contacted the destination server *dst* within this interval. We denote the bitmap of the ith period by B_i. The router will download B_i from on-chip SRAM to DRAM for post-processing.

Given a sequence of bitmaps $B_1, B_2, \ldots, B_t$ in main memory that have recorded the flow *dst*'s traffic for t consecutive periods, our problem is to design an algorithm that can use these bitmaps to estimate $n^* = |S_1 \cap S_2 \ldots \cap S_t|$. Here, the persistent spread n^* is the number of distinct elements that persist through the t time periods and appear in all the bitmaps.

Bitwise OR For this problem, a possible solution is to calculate the union bitmap $B_1 \vee B_2 \ldots \vee B_t$ by bitwise OR, and extract information from it about $|S_1 \cup S_2 \ldots \cup S_t|$ to assist the estimation of persistent spread (please search for inclusion–exclusion principle). However, this solution has poor accuracy when t is large. This is because the estimation accuracy of a bitmap algorithm depends on the fill rate—the proportion of bits in a bitmap that are set to one: The higher the fill rate, the worse the estimation accuracy [12]. Since the fill rate of union bitmap $B_1 \vee B_2 \ldots \vee B_t$ increases as t grows, any algorithm based on the union will experience the accuracy degradation. The accuracy loss as t value grows will prohibit network operators to configure an arbitrarily large t, which is critical for differentiating persistent elements from transient elements.

Bitwise AND Instead of the union bitmap, an alternative solution is to calculate the intersection bitmap $B^* = B_1 \wedge B_2 \ldots \wedge B_t$ by bitwise AND. As stated in (4.1), each flow element picks a bit in B_i pseudo-randomly by hash function H. Hence, a persistent element always sets the same bit in bitmap B_i, irrelevant of the index i of a time period. If a persistent element sets the jth bit in B_1 to one, then in subsequent bitmaps, the jth bit will be set to one. Hence, we probably can estimate the number of persistent elements, by counting the bits that are "1" in all the bitmaps $B_1, B_2, \ldots, B_t$, or equivalently, the number of "1" bits in the intersection bitmap $B^* = B_1 \wedge B_2 \ldots \wedge B_t$.

While using the intersection bitmap B^*, the main difficulty to achieve satisfactory estimation accuracy is the *false positive probability*, which is the probability for a bit to be assigned to "1" in each time period by different transient elements, making the bit look as if it were set by a persistent element. This phenomenon occurs mostly frequently when the bitmaps $B_1, B_2, \ldots, B_t$ are overly dense with just a small proportion of zero bits (especially when the number of periods t is small). We will address this false positive issue shortly.

4.3 Estimator Based on Intersection Bitmap

Based on the intersection bitmap B^*, in this section, we present an algorithm to estimate the cardinality of persistent elements in $S^* = S_1 \cap S_2 \ldots \cap S_t$, for an arbitrary number of time periods t. In a single period, putting the persistent elements aside, other elements $S_i - S^*$ are called *transient elements*, which are generated by the comes and goes of normal users. We will filter the short-term network traffic, and estimate the number of persistent elements $n^* = |S^*|$.

4.3.1 Analysis of Real Network Traces

Before proceeding to detailed analysis, we firstly verify our assumption about rough independence of transient elements in different measurement periods. The verification uses the traces of real network traffic from CAIDA [11]. In the trace files, we have identified tens of servers with low persistent spreads and free from malicious attacks. Hence, for such servers, almost all of their traffics can be regarded as transient elements due to the intersection between normal users and servers. In Table 4.2, we have tested the inter-dependency of transient traffics, in two arbitrary measurement periods. The subtable (a) is about a HTTP server 224.243.38.27/80, and the subtable (b) is for a HTTPS server 224.243.38.27/443. In both of them, the length of a measurement period is configured to 7 min, and there is a spacing that lasts for 3 min between any two adjacent periods, in order to reduce the chance of normal users' activities crossing the border of two neighboring periods.

Table 4.2 Weak dependency of normal traffics in different measurement periods, if large persistent flows are absent

Two-period intersection:	1	2	3	4	5	6
(a) HTTP Server 224.243.38.27/80						
1	22604					
2	5.0 %	24598				
3	0.8 %	4.2 %	27561			
4	0.4 %	0.9 %	4.2 %	29426		
5	0.5 %	0.5 %	0.9 %	4.5 %	29456	
6	0.3 %	0.5 %	0.4 %	0.8 %	4.2 %	30489
(b) HTTPS Server 224.243.38.27/443						
1	40205					
2	3.2 %	47119				
3	2.0 %	3.2 %	48433			
4	1.8 %	1.9 %	3.0 %	60050		
5	1.6 %	1.7 %	1.9 %	3.4 %	64332	
6	1.7 %	1.8 %	1.8 %	3.6 %	3.9 %	69356

With six measurement periods, subtable (a) lists the cardinality of the intersection of two element sets S_i and S_j for two arbitrary periods i and j: When $i = j$, we show the spread $|S_j|$ of the jth period; When $i > j$, we are supposed to show the intersected spread $|S_i \cap S_j|$ of the two periods, but we calculate the ratio $\frac{|S_i \cap S_j|}{|S_j|}$ instead, in order to give an impression of how small the dependency between two periods. Subtable (a) shows that, for any pair of non-neighboring periods, their intersected spread is less than 1 %, and hence they can be approximated as independent. Subtable (b) demonstrates a similar phenomenon, where the intersected spreads of two periods are less than 4 %. Hence, when contacting HTTPS servers, although network users are more prone to be heavy users that cross multiple periods, the assumption about rough dependency between different time periods is still valid.

We have also analyzed the traffics of tens of other HTTP and HTTPS servers. The evaluation results are similar: A legitimate user's continuous interaction with a website is pretty short in duration. They only check out necessary information from one website and don't linger for long and jump to another website. For the transient traffic they generated, there exists a rough independence between different periods. There are two key points of establishing such an independence. The first is to configure an appropriate length of measurement periods (e.g., larger than 7 min), in order to let one period contain a normal user's interaction with a website. The second is to add a decent spacing between neighboring periods (e.g., 3 min), to reduce the chance of a user's activity crossing the borders of two periods. What we have accomplished is to confine normal uses' short-term behaviors within one period. We mainly care about the long-term stealthy activities that span multiple periods in order to degrade a website's performance.

4.3.2 Persistent Spread Estimator

In this subsection, we present the formulas of our persistent spread estimator. The inputs are a sequence of bitmaps B_1, B_2, ..., B_t, and their intersection bitmap B^*. There are two cases for a bit in B^* to be set to "1":

1. it contains at least one persistent elements, or
2. it contains none of the persistent elements, but in each time period, it contains at least one transient elements.

The probability of the first case is $1 - P^*$, where P^* is the probability for the bit in B^* to contain no persistent elements.

$$P^* = (1 - \frac{1}{m})^{n^*} \approx e^{-\frac{n^*}{m}} \tag{4.2}$$

Here, we have applied the approximation $(1 - \frac{1}{m})^n \approx e^{-\frac{n}{m}}$ that works for large m value.

Let P_i be the probability for a bit of B_i to contain no transient elements in the ith period. We have

$$P_i = (1 - \frac{1}{m})^{n_i - n^*} \approx e^{-\frac{n_i - n^*}{m}} \quad (1 \le i \le t), \tag{4.3}$$

where $n_i - n^*$ is the number of transient elements in the ith period. Hence, the probability of the second case is $P^* \prod_{1 \le i \le t}(1 - P_i)$, or called the false positive probability. Note that our modeling of the false positive probability assumes the rough independence of transient elements at different periods.

Let X_j^* is the event that the jth bit in B^* is set to "1". The probability of X_j^* is

$$Pr\{X_j^*\} = (1 - P^*) + P^* \prod\nolimits_{1 \le i \le t}(1 - P_i).$$

Let Z^* be the proportion of bits in B^* that remain zeros. We have $1 - Z^*$ equals the arithmetic mean of m random variables:

$$1 - Z^* = \frac{1}{m} \cdot \sum\nolimits_{j=0}^{m-1} 1_{X_j^*}, \tag{4.4}$$

where $1_{X_j^*}$ is the indicator function of X_j^*, which equals one when the event X_j^* happens. Since the bits in B^* are mutually independent, $E(1 - Z^*) = \frac{1}{m} \sum_{j=0}^{m-1} E(1_{X_j^*}) = E(1_{X_j^*})$. This implies that the expected proportion of bits in B^* that are ones $E(1 - Z^*)$ is equal to the probability $Pr\{X_j^*\}$. Hence,

$$E(1 - Z^*) \approx (1 - P^*) + P^* \prod\nolimits_{1 \le i \le t}(1 - P_i). \tag{4.5}$$

By multiplying both sides of (4.5) by $(P^*)^{t-1}$, we have

$$(P^*)^{t-1}E(Z^*) \approx (P^*)^t - \prod\nolimits_{1\le i\le t}(P^* - P^*P_i).$$

Combining (4.2) and (4.3), we have the following approximation:

$$P_i \approx e^{-\frac{n_i-n^*}{m}} = e^{-\frac{n_i}{m}} \,/\, e^{-\frac{n^*}{m}} \approx E(Z_i)\,/\,P^*$$

Applying the approximation, we have

$$(P^*)^{t-1}E(Z^*) \approx (P^*)^t - \prod\nolimits_{1\le i\le t}\big(P^* - E(Z_i)\big). \tag{4.6}$$

Since $1 - Z^*$ is the arithmetic mean of m independent random variables as shown in (4.4), according to the central limit theorem, Z^* approximates a Gaussian distribution. Its variance is inversely proportional to the bitmap size m, which have been proved in Appendix 1. Hence, when m is sufficiently large (e.g., a few thousands), we can substitute the mean value $E(Z^*)$ in (4.6) by an instance value Z^* without producing significant estimation error. By a similar reason, we can replace $E(Z_i)$ by an instance value Z_i. Therefore, we have

$$(\hat{P^*})^{t-1}Z^* \approx (\hat{P^*})^t - \prod\nolimits_{1\le i\le t}(\hat{P^*} - Z_i). \tag{4.7}$$

Here, an estimation of P^* is denoted by $\hat{P^*}$ with an upper hat.

By observing bitmaps B^* and B_i, we can know Z^* and Z_i, respectively. Hence, there is only one unknown variable $\hat{P^*}$ in (4.7). We can solve this equation for $\hat{P^*}$, and then use the relation $P^* \approx e^{-\frac{n^*}{m}}$ in (4.2) to obtain an estimation $\hat{n^*}$. In the following, we present the formula of the estimation $\hat{n^*}$ for different number of periods t.

When $t = 1$, Eq. (4.7) can be simplified as $Z^* = Z_1$. This is natural because we have $B^* = B_1$ when there is a single period. Since $t = 1$, all the flow elements in bitmap B^* are persistent elements. We can estimate their number from the proportion of bits in B^* that are zeros. Hence,

$$\hat{n^*} = -m\ln(Z^*)\ . \tag{4.8}$$

When $t = 2$, Eq. (4.7) becomes

$$\begin{aligned} 0 &\approx (\hat{P^*})^2 - (\hat{P^*} - Z_1)(\hat{P^*} - Z_2) - \hat{P^*}Z^* \\ &\approx (Z_1 + Z_2 - Z^*)\hat{P^*} - Z_1Z_2. \end{aligned}$$

Hence, we have $\hat{P^*} \approx \frac{Z_1Z_2}{Z_1+Z_2-Z^*}$. Combining it with $P^* \approx e^{-\frac{n^*}{m}}$, we can estimate the persistent spread as

$$\begin{aligned}\hat{n^*} &= -m\ln(\hat{P^*}) \approx -m\ln\left(\frac{Z_1 Z_2}{Z_1+Z_2-Z^*}\right)\\ &= m\ln(Z_1+Z_2-Z^*) - m\ln(Z_1) - m\ln(Z_2).\end{aligned} \tag{4.9}$$

When $t = 3$, Eq. (4.7) can be converted to

$$0 \approx \Big(\sum_{1\le i\le 3} Z_i - Z^*\Big)(\hat{P^*})^2 - \Big(\sum_{1\le i<j\le 3} Z_i Z_j\Big)\hat{P^*} + \prod_{1\le i\le 3} Z_i.$$

We firstly solve the above equation for $\hat{P^*}$, and then use the relation $P^* \approx e^{-\frac{n^*}{m}}$ to estimate persistent spread n^* as

$$\hat{n^*} = m\ln\left(\frac{B - \sqrt{B^2 - 4A\Big(\sum\limits_{1\le i\le 3} Z_i - Z^*\Big)}}{2A}\right), \tag{4.10}$$

where

$$A = \prod\nolimits_{1\le i\le 3} Z_i, \qquad B = \sum\nolimits_{1\le i<j\le 3} Z_i Z_j.$$

When $t \ge 4$, because the order of (4.7) about $\hat{P^*}$ grows to at least three, it is complicated to obtain a closed-form estimator. Hence, we solve (4.7) by numerical root-finding algorithms, e.g., Newton–Raphson method.

Firstly, we generate an initial guess of $\hat{P^*}$, using $\hat{P^*} \approx Z^*$. This approximation is obtained by dropping the false positive probability $P^* \prod_{0\le i\le t}(1-P_i)$ in (4.5).

Secondly, we optimize the current value of $\hat{P^*}$, by invoking the following equation iteratively:

$$\hat{P^*} = \hat{P^*} - \frac{z(\hat{P^*})}{z'(\hat{P^*})},$$

where

$$\begin{aligned} z(\hat{P^*}) &= (\hat{P^*})^t - (\hat{P^*})^{t-1} Z^* - \prod\nolimits_{1\le i\le t}(\hat{P^*} - Z_i),\\ z'(\hat{P^*}) &= t\,(\hat{P^*})^{t-1} - (t-1)(\hat{P^*})^{t-2} Z^* - \\ &\qquad \Big(\prod\nolimits_{1\le i\le t}(\hat{P^*} - Z_i)\Big)\Big(\sum\nolimits_{1\le j\le t}\frac{1}{(\hat{P^*} - Z_j)}\Big).\end{aligned}$$

Thirdly, when the optimization process converges, we use the best $\hat{P^*}$ to derive an estimation of the persistent spread.

$$\hat{n^*} = -m\ln\hat{P^*} \tag{4.11}$$

In summary, for an arbitrary t value, we have presented an equation to calculate the estimation of persistent spread $\hat{n^*}$. In order to shield the difference in the estimation equations of n^*, we define a unified function f_t in the following theorem:

Definition 1 (Bitmap-Based Persistent Spread Estimator). Given an arbitrary number of periods t ($t \geq 1$), a unified estimator function to estimate the persistent spread is

$$\hat{n^*} = f_t\big(m, Z^*, \{Z_i\}\big), \tag{4.12}$$

where Z^* is the proportion of bits of the intersection bitmap B^* that are zeros, Z_i is the zero ratio of bitmap B_i in the ith period ($1 \leq i \leq t$), and m is the size of each of the bitmaps. When t is 1, 2, 3 or at least 4, respectively, the function f_t corresponds to Eqs. (4.8), (4.9), (4.10), or (4.11).

4.4 Analysis of Bitmap-Based Estimator

In this section, we analyze the bias and variance of our intersection bitmap-based estimation $\hat{n^*}$ in Definition 1.

Firstly, we prove that the estimation $\hat{n^*}$ is asymptotically unbiased when the bitmap size m is sufficiently large. We know that the zero ratio Z^* of bitmap B^* approximates a Gaussian distribution, because Z^* is the arithmetic mean of a large quantity of independent random variables as in (4.4). For a similar reason, the zero ratio Z_i of bitmap B_i approximates a Gaussian distribution. From (4.7), we know that $\hat{P^*}$ is a polynomial function of Z^* and Z_i, with continuous first partial derivatives. According to the multivariate delta-method [1], when the bitmap size m is enough large, $\hat{P^*}$ approximately follows a Gaussian distribution, and its expected value $E(\hat{P^*})$ satisfies

$$\big(E(\hat{P^*})\big)^{t-1}E(Z^*) \approx \big(E(\hat{P^*})\big)^t - \prod\nolimits_{1\leq i\leq t}\big(E(\hat{P^*}) - E(Z_i)\big),$$

which is obtained by substituting Z^* by $E(Z^*)$, and Z_i by $E(Z_i)$ in (4.7). Combining the above formula with (4.6), we can derive that $E(\hat{P^*}) \approx P^*$. Therefore, $\hat{P^*}$ approximates a Gaussian distribution whose expected value is P^*. Further, we have $\hat{n^*}$ is a function of $\hat{P^*}$ as $\hat{n^*} = -m\ln(\hat{P^*})$. From delta-method [1], $\hat{n^*}$ approximates a Gaussian distribution with

$$E(\hat{n^*}) \approx -m\ln(E(\hat{P^*})) \approx -m\ln(P^*) = -m\ln(e^{-\frac{n^*}{m}}) = n^*.$$

Therefore, the persistent spread estimation $\hat{n^*}$ is asymptotically unbiased, when the bitmap size m is sufficiently large.

Secondly, we analyze the variance of estimation $\hat{n^*}$ in the following theorem using the tool of Cramér-Rao bound:

Theorem 3 (Variance of Multi-period Estimators). *For our persistent spread estimator in Definition 1, its variance is*

$$Var(\hat{n^*}) \approx \frac{n^*}{\rho^*} \cdot \frac{1}{(1-\tilde{P})^2\left(\frac{1}{\tilde{P}} + \frac{1}{1-\tilde{P}}\right)}, \tag{4.13}$$

where $\rho^* = \frac{n^*}{m}$ *is the density of persistent elements,* $SNR_i = \frac{n^*}{n_i - n^*}$ *is the signal-to-noise ratio in the ith period, and*

$$\tilde{P} = \left(1 - e^{-\rho^*}\right) + e^{-\rho^*} \prod\nolimits_{1 \le i \le t} \left(1 - e^{-\rho^* \frac{1}{SNR_i}}\right).$$

Proof. Please check Appendix 2 for a proof.

From Theorem 3, we derive the standard estimation error as

$$\frac{\sqrt{Var(\hat{n^*})}}{n^*} = \frac{1}{\sqrt{m}} \Big/ \sqrt{(\rho^*)^2 \cdot (1-\tilde{P})^2\left(\frac{1}{\tilde{P}} + \frac{1}{1-\tilde{P}}\right)}.$$

The relative standard error is affected by four factors: (1) density of persistent elements ρ^*, or call it persistent load factor, (2) the number of bits m, (3) signal-to-noise ratio SNR_i, and (4) the number of time periods t. We analyze their impacts by plotting the relative error against these factors in Fig. 4.2.

Figure 4.2 shows that the estimation accuracy improves as the increase of signal-to-noise level SNR_i, which has been defined as persistent spread n^* divided by the cardinality of transient elements $n_i - n^*$ in the ith period. Subfigure (a) configures the

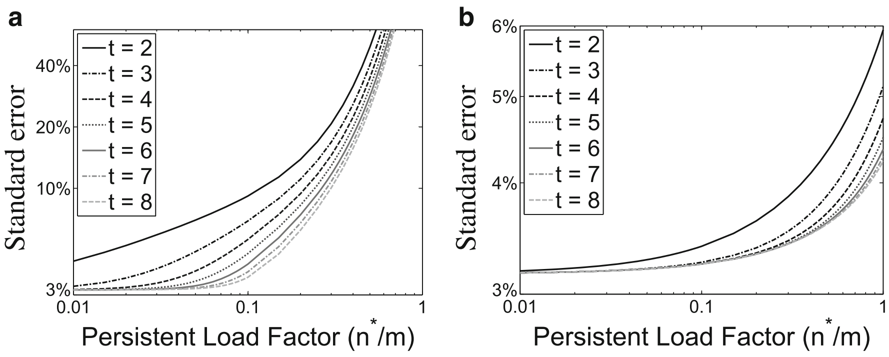


Fig. 4.2 Accuracy of persistent spread estimation with $n^* = 1000$. (**a**) For each ith period, $\text{SNR}_i = 0.1$. (**b**) For each ith period, $\text{SNR}_i = 1$

SNR$_i$ as low as 0.1, and the accuracy ranges between 3 and 40+ % depending on the load factor. In contrast, subfigure (b) increases SNR$_i$ by ten times to 1. Hence, the accuracy improves and fluctuates between 3 and 6 %.

We focus on Fig. 4.2b, and it tells us that the estimation accuracy improves as the number of periods t increases. Given more bitmaps $B_1, B_2, \ldots, B_t$, our persistent spread estimator can to reduce the false positive probability $P^* \prod_{1 \le i \le t}(1 - P_i)$, and better filter the transient contacts. This plot also shows that the estimation accuracy deteriorates as the density of persistent elements ρ^* increases. The explanation is that our estimation relies on the proportion of zero bits in bitmap B^*. If the density of persistent elements ρ^* grows, B^* will become crowded, and when ρ^* exceeds a bound, the proportion of zero bits in B^* approach zero, which cannot be used for accurate estimation.

4.5 Multi-Virtual Bitmap Estimator

In the design of our previous bitmap-based estimator, each flow is allocated with a bitmap to record its elements in a time period, and all the bitmaps are separated and with equal size. Because bitmap algorithm only supports the counting of cardinalities linear to bitmap size [12], this design best fits the case that the flow spreads uniformly distribute. However, the distribution of flow spreads is extremely unbalanced in real networks, especially in core networks. We plot a distribution of flow spreads in Fig. 4.3, which is obtained from real-world traffic traces from CAIDA [11]. In subfigure (a) where the measurement duration is set to 1 min, there are about a million of flows whose spreads are smaller than 100. In contrast, only a few hundred flows have their spreads larger than 1000. Such an unbalanced distribution of flow spreads can also be witnessed in subfigure (b) where the measurement duration extends to 20 min. Throughout the chapter, we use the term

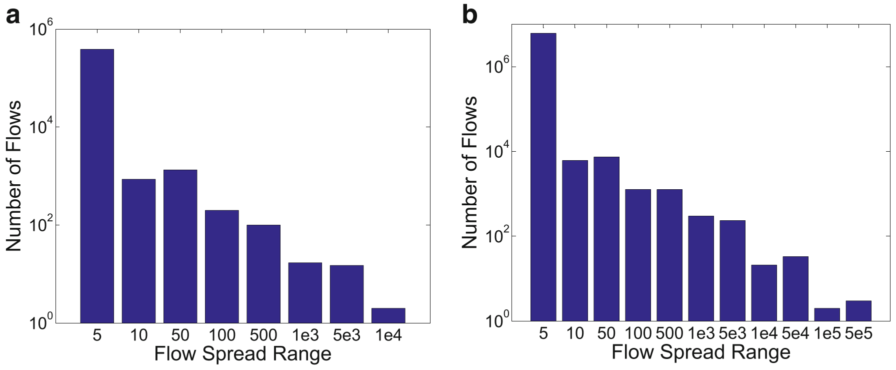


Fig. 4.3 Flow spread distribution by analyzing CAIDA traces. (**a**) Duration = 1 min. (**b**) Duration = 20 min

mouse flows to refer to the flows with spreads less than one hundred, and taking the majority of all flows. The term *elephant flows* is used for the flows with extraordinarily larger spreads than the rest. They typically correspond to the server machines with a large number of concurrent users.

Due to the uneven distribution of flow spreads, if allocating all the flows with separated and equal-sized bitmaps, it incurs a significant waste of precious space of on-chip SRAM, which we explain as follows. Since the bitmap method can only count cardinalities linear to bitmap size [12], we have to configure the bitmap size large enough and proportional to the spreads of elephant flows. Otherwise, these bitmaps, when receiving too many elements from elephant flows, have most their bits to be “1”, which severely degrades the estimation accuracy. However, we cannot predict which flows are elephant flows, and to guarantee the estimation accuracy, we have to allocate all the flows with equal-sized bitmaps that are large enough to accommodate the elephant flows. Therefore, for mouse flows with small spreads, their bitmaps are inevitably sparse with most bits being “0”, which causes a significant waste of the expensive SRAM space, especially considering the fact that the majority of flows witnessed by the router are mouse flows.

To mitigate the memory waste due to uneven distribution of flow spreads, we adopt the idea of *virtualization*: the bitmaps of all the flows are no longer separated but share a common bit pool, which is called the *physical bitmap*. Then, the bitmap of each flow draws its bits pseudo-randomly from the common bit pool, which is called a *virtual bitmap* since it does not physically exist. As illustrated in Fig. 4.4, the bits of a virtual bitmap uniformly distribute in the physical bitmap. For all the virtual bitmaps, we configure a unified size that is large enough to accommodate elephant flows. Through the bit sharing in physical bitmap, the elephant flows can “borrow” bits from the under-used virtual bitmaps of mouse flows. The physical bitmap is denoted by M, which is an array with u bits allocated from on-chip SRAM. We will describe in detail how to utilize this data structure to estimate the persistent spreads simultaneously for multiple flows.

The idea of virtual bitmaps sharing a physical space has been partially discussed in prior literature [13]. However, they concentrate on estimating the cardinality of a single set. In contrast, we estimate the cardinality of intersection of multiple sets, which are collected in different time domains.

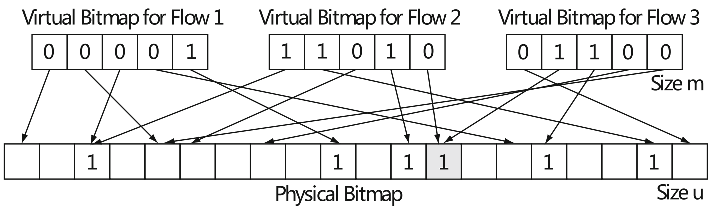

Virtual Bitmap for Flow 1
0 0 0 0 1
Virtual Bitmap for Flow 2
1 1 0 1 0
Virtual Bitmap for Flow 3
0 1 1 0 0
Size m
1 1 1 1 1 1
Physical Bitmap
Size u

Fig. 4.4 Multiple virtual bitmaps share bits in a physical bitmap

4.5.1 Physical Bitmap Encoding

In each time period, the router will observe a large number of traffic flows. For each flow, the router stores its elements to the physical bitmap M, which we explain in details as follows.

Whenever a packet arrives whose header is $\langle src, dst \rangle$, the router uses its destination address to categorize it to the flow dst, and treats its source address src as an element of flow dst, which is mapped to the flow's virtual bitmap. Assume the jth bit in the virtual bitmap has been set to one by the element:

$$j = H(src) \bmod m, \tag{4.14}$$

where H is a hash function and m is the size of virtual bitmaps which is sufficiently large to accommodate an elephant flow.

According to the bit sharing scheme in Fig. 4.4, the jth ($0 \leq j < m$) bit in the virtual bitmap will be drawn from or mapped to the ith ($0 \leq i < u$) bit in the physical bitmap:

$$i = H_{dst}(j) \bmod u,$$

where H_{dst} is a hash function used by the flow dst for bit mapping. It can be implemented from a master hash function H:

$$H_{dst}(j) = H(j \oplus dst), \tag{4.15}$$

where $\oplus$ is bitwise XOR or string concatenation to combine two key values j and dst. Applying Eq. (4.14), we have

$$H(j \oplus dst) = H\big((H(src) \bmod m) \oplus dst\big)$$

In summary, when the packet $\langle src, dst \rangle$ arrives, the following bit in the physical bitmap M will be set to one:

$$M[i] := 1,$$

where $i = H\big((H(src) \bmod m) \oplus dst\big) \bmod u$.

When the time period terminates, the physical bitmap M will be downloaded from on-chip SRAM to main memory. Assume we have t physical bitmaps, denoted by $M_1, M_2, \ldots, M_t$, which correspond to t consecutive time periods.

4.5.2 Persistent Spread Estimation

In this subsection, we describe how to use the sequence of physical bitmaps M_1, $M_2, \ldots, M_t$, to estimate the persistent spread for a particular flow *dst*. An intuitive method is that, from an arbitrary physical bitmap M, we can extract a virtual bitmap B that belongs to the flow *dst*.

$$B = \langle M[H_{dst}(0)],\ M[H_{dst}(1)],\ \ldots,\ M[H_{dst}(m-1)]\rangle$$

Here, we use the relation that the jth $(0 \le j < m)$ bit in virtual bitmap has been mapped to the ith bit in physical bitmap as $i = H_{dst}(j) \bmod u$, and we omit mod u for simplicity. Since we have t physical bitmaps $M_1, M_2, \ldots, M_t$, we can extract t virtual bitmaps, noted as $B_1, B_2, \ldots, B_t$. Then, we can apply our previous algorithm in Sect. 4.3.2, to filter the transient elements and estimate the number of persistent elements hiding in the virtual bitmap of flow *dst*.

However, this method has the problem of overestimating the persistent spread of flow *dst*. In the flow's virtual bitmap, the persistent elements may not belong to flow *dst* alone. Since the virtual bitmap of a flow draws bits from a common bit pool that is shared with other flows, the bits in the virtual bitmap may be assigned by other flows to "1". If some of the "1" bits happen to be set by persistent elements coming from other flows, then these bits will be set to "1" in all the virtual bitmaps B_1, B_2, $\ldots, B_t$ in t time periods, which causes the overestimation of persistent spread of the flow *dst*.

It may appear that transient elements from other flows may also cause overestimation, since they increase the number of elements n_i that are contained in the virtual bitmap B_i and aggravate the false positive probability. However, when we use zero ratio of B_i to estimate the number of elements in the virtual bitmap of ith period, the estimation result already counts the transient elements coming from other flows. Hence, when we estimate the number of persistent elements in virtual estimator of flow *dst*, there won't be any overestimation. The major source of overestimating flow *dst*'s persistent spread is the persistent elements from other flows.

Our solution is to compensate the estimation bias due to persistent elements coming from other flows. Let n^* be the number of persistent elements that belong to flow *dst*, n_m^* be the number of persistent elements in virtual bitmap of flow *dst*, and n_u^* be the number of persistent elements in physical bitmap M. Our basic idea is that the total number of persistent elements from other flows is $n_u^* - n^*$, and they uniformly distribute in the entire physical bitmap, which are *noises*. Let X be the number of noise elements that are mapped to a bit of physical bitmap M. We know that X follows a binomial distribution: $X \sim Binom(n_u^* - n^*, \frac{1}{u})$. The expected number of noises elements mapped to a virtual bitmap equals to $E(mX)$, where m is the number of bits in a virtual bitmap.

$$E(n_m^* - n^*) = E(mX) = mE(X) = \frac{m}{u}(n_u^* - n^*)$$

According to the laws of large numbers in probability theory, when the number of independent variables m is large enough, the variance $Var(\frac{n_m^* - n^*}{E(n_m^* - n^*)})$ approaches to zero. Hence, when the number of trials m is large, the expected value $E(n_m^* - n^*)$ can be approximated by its instance value $n_m^* - n^*$. Then,

$$n_m^* - n^* \approx E(n_m^* - n^*) = \frac{m}{u}(n_u^* - n^*).$$

By conversion, we have an estimator of persistent spread n^*.

$$n^* \approx \frac{um}{u-m}(\frac{n_m^*}{m} - \frac{n_u^*}{u}).$$

In summary, our estimator can be divided into three steps.

First, we estimate n_m^*—the number of persistent elements that are mapped to the virtual bitmap of flow *dst*:

$$\hat{n_m^*} = f_t\big(m, Z_m^*, \{Z_{m,i}\}\big)$$

where f_t is the persistent spread estimator in (4.12), m is the number of bits in virtual bitmaps, $Z_{m,i}$ is the proportion of bits in the ith virtual bitmap B_i that are zeros, and Z_m^* is the ratio of bits in $B^* = B_1 \wedge B_2 \wedge \ldots \wedge B_t$ that are zeros.

Second, we estimate n_u^*—the number of persistent elements in physical bitmap:

$$\hat{n_u^*} = f_t\big(u, Z_u^*, \{Z_{u,i}\}\big),$$

where f_t is the persistent spread estimator in (4.12), u is the number of bits in physical bitmap, $Z_{u,i}$ is the proportion of bits in physical bitmap M_i that are zeros, and Z_i^* is the ratio of bits in $M^* = M_1 \wedge M_2 \wedge \ldots \wedge M_t$ that are zeros.

Third, we compensate the positive bias in $\hat{n_m^*}$ due to noise persistent elements from other flows, and we obtain the unbiased estimation $\hat{n^*}$ below, for flow *dst*'s persistent spread.

$$\hat{n^*} = \frac{um}{u-m}\left(\frac{\hat{n_m^*}}{m} - \frac{\hat{n_u^*}}{u}\right) \tag{4.16}$$

4.6 Simulation Evaluation

In this section, we use simulation to evaluate the estimators for persistent spread measurement: One is based on the intersection of bitmaps, and the other is the multi-virtual bitmaps. Our goal is to design an estimator that is able to use the tight space on on-chip SRAM to deliver high accuracy. Hence, in our experiments, the memory cost, when averaging over all elements appearing in an arrival packet stream, is less than 1 bit per element. The only related work that can work in such tight space is a method based on a continuous variant of Flajolet–Martin sketches, named FMSK for short [2]. We will compare our methods with FMSK in estimation accuracy. We will show the impact of the number of periods t and the signal-to-noise ratio SNR_i on estimation accuracy, which is not quantified by previous works. We will also compare our methods with the aforementioned hash table method storing partial signatures (call it partial hash for short), to show the power of our methods in compressing memory cost.

4.6.1 Experiment Setup

We simulate the real-world network traffic using the following parameters. The number of flows that can be observed by the gateway router is configured to 1024, which simulates a small server farm. For a flow, the average number of elements in the ith ($1 \leq i \leq t$) period is configured to 1200, which simulates multiple users concurrently accessing a single server. Some of the flow elements are persistent elements, which exist throughout the t periods, and the rest are transient elements. In each period, we control the ratio of persistent elements to the transient elements by signal-to-noise ratio SNR_i. For these transient elements, we assume that 90 % of them stay within one period, and the remaining 10 % are heavy users that cross the boundaries between periods.

For fair comparison, we allocate the same size of memory for partial hash, FMSK, and our methods. As listed in Table 4.3, each of the three method is given roughly 1.2M bits SRAM, which means each flow gets 1144 bits on average for its spread estimation. FMSK divides these bits into thirty five float numbers, each of which is 32 bits long and can perform the counting independently. Their stochastic averaging is treated as the final estimation. Our bitmap method uses these bits as a bitmap to record the flow elements (whose expected number is 1200). Our multi-virtual bitmap method does not separate the allocated SRAM space into equal-sized bitmaps. It instead lets the virtual bitmaps to share the space. The length of each

Table 4.3 Settings of algorithm parameters

Algorithm	Memory	Parameters
Partial hash	$\approx$ 9.1 Mbit	Flow signature = 8 bit, buckets per flow = 32, source signature = 4 bit
FMSK	$\approx$ 1.2 Mbit	For a flow, number of buckets = 35. In one bucket, an FMSK = 32 bit
Bitmap	$\approx$ 1.2 Mbit	For one flow, the size of each bitmap = 1144 bit
Multi-virtual bitmap	$\approx$ 1.23 Mbit	virtual bitmap size = 6 Kbit, physical bitmap size = 1.23 Mbit

virtual bitmap is configured as large as 6k bits to accommodate elephant flows. For the most basic method based on partial hashing, we give it 9.1M bits SRAM to show the power of our methods (only having 1.2M bits) in compressing memory cost.

4.6.2 Estimation Accuracy and Operating Range

In this subsection, we compare the four methods (listed in Table 4.3) in estimation accuracy and operating range. The comparison results are presented in Figs. 4.5, 4.6, 4.7, and 4.8.

Figure 4.5 shows that, although the partial hash method is given 9.1M bits memory that is eight times larger than other methods, its estimation is negatively biased. This is because its operating range is merely $2^4 \times 32 = 512$, where 4 is the size of partial signature stored in one bucket and 32 is the number of buckets allocated for one flow. When the persist spread exceeds this range, it is severely underestimated as depicted in Fig. 4.5.

Figure 4.6 states that FMSK can use only 1.2M bits memory to generate unbiased estimations. However, its accuracy is far from satisfactory. This is because FMSK, similar to Flajolet–Martin sketches [4], has the problem of slow start: it has low

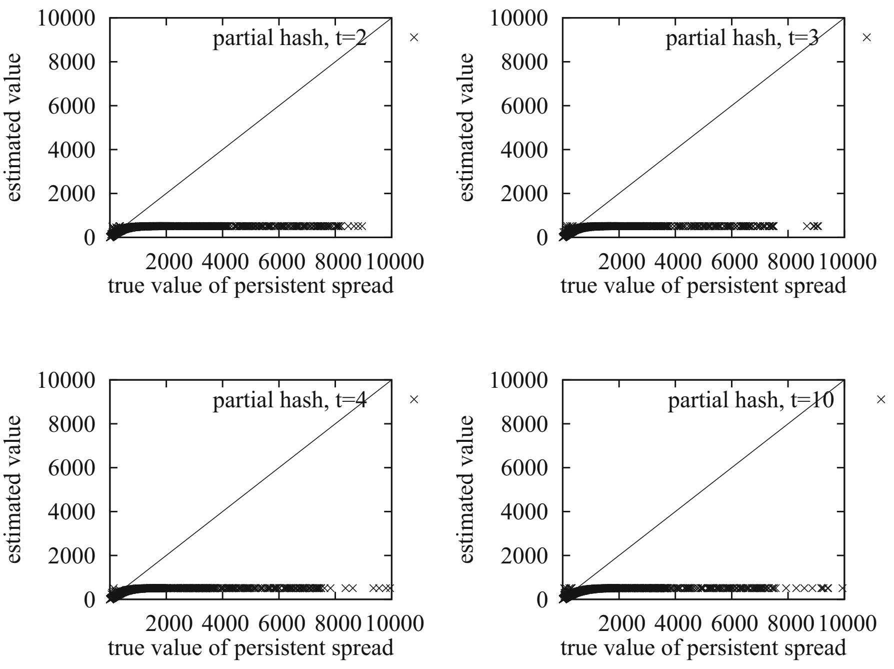


Fig. 4.5 Persistent spread estimation of partial signature, with $\text{SNR}_i = 1$ and number of periods $t = 2, 3, 4, 10$

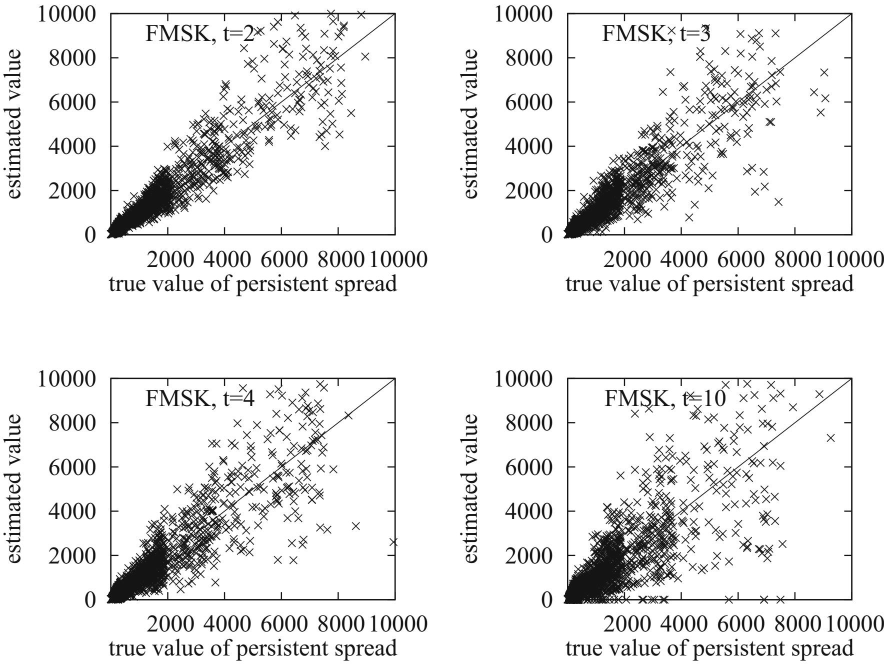


Fig. 4.6 Persistent spread estimation using FMSK algorithm, with $\text{SNR}_i = 1$ and number of periods $t = 2, 3, 4, 10$

inaccuracy when the cardinality to be estimated is on the scale of bits allocated, which is about 1120 bits in the simulation. We will explain later that FMSK has another inadequacy that its accuracy degrades when the number of time periods t grows.

Figure 4.7 shows that our bitmap method, when given the same memory of 1.2M bits, can improve estimation accuracy significantly as compared with FMSK. Its shortcoming, however, is its small operating range: When the persistent spread exceeds a point (about 2000 in Fig. 4.7), its estimations are strongly biased. This is because, for elephant flows, their bitmaps will receive too much elements, which set most of their bits to one and cause severe bias. This shortcoming can be overcome by our multi-virtual bitmap method, which permits elephant flows to borrow bits from small flows, to extend their operating range. In Fig. 4.8, this method provides accurate estimations even for persistent spreads as large as 10,000. This is because the size of a virtual bitmap (6k bits) is configured five times larger than the size of a bitmap (1144 bits), as shown in Table 4.3.

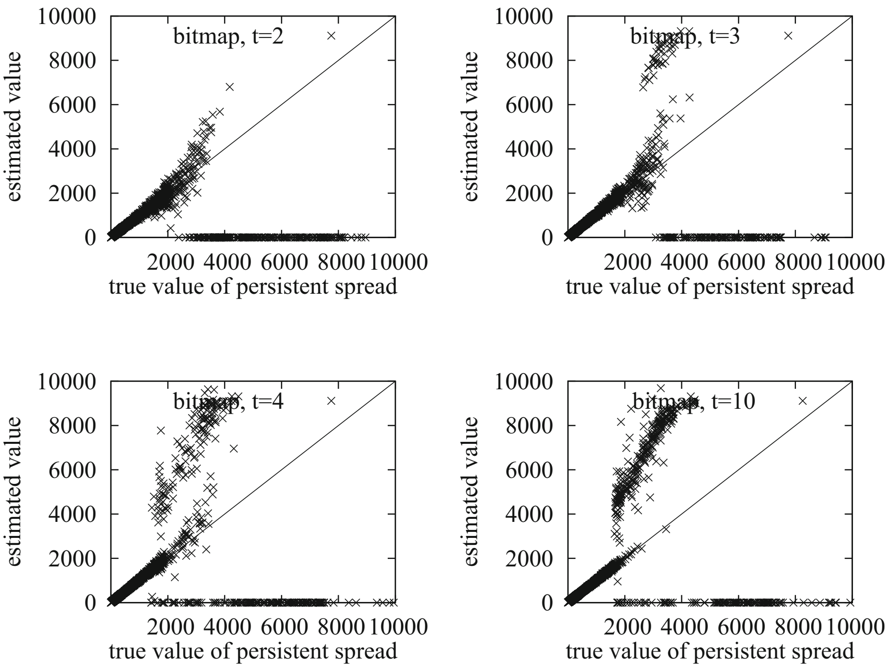


Fig. 4.7 Persistent spread estimation using bitmap algorithm, with $\text{SNR}_i = 1$ and number of periods $t = 2, 3, 4, 10$

4.6.3 *Impact of Time Period t on Accuracy*

An interesting feature of our multi-virtual bitmap method is that its estimation accuracy improves when the number of time periods t increases. Figure 4.8 depicts the case of $t = 2$ in the leftmost subfigure, and illustrates $t = 10$ in the rightmost. It is a useful feature that permits network operators to set arbitrarily large t values to differentiate persistent and transient elements.

In contrast, the accuracy of FMSK declines when t value grows, as illustrated in Fig. 4.6. When t grows to 10, its estimation error becomes even larger than 50 %. This is because the FMSK method estimates the persistent spreads from the fraction of the intersected set to the union set of all t periods: $\frac{|S_1 \cap S_2 \ldots \cap S_t|}{|S_1 \cup S_2 \ldots \cup S_t|}$. As t value grows, the size of union set $|S_1 \cup S_2 \ldots \cup S_t|$ expands, which reduces the fraction of intersected set and degrades the estimation accuracy. Our bitmap method is different. It detects the existence of a persistent element from the phenomenon that a bit is set to "1" in all the bit arrays B_1, B_2, ..., B_t. The false positive probability, which the probability that such a bit is occupied only by transient elements, decreases as t value grows, which is the last term $P^* \prod_{1 \le i \le t}(1 - P_i)$ in (4.5).

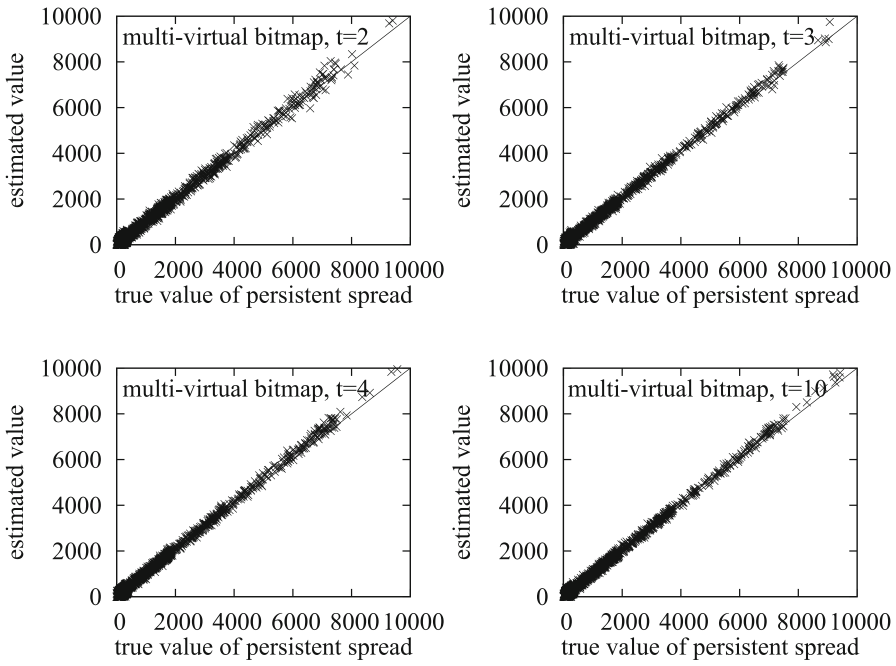


Fig. 4.8 Persistent spread estimation using multi-virtual bitmap algorithm, with $\text{SNR}_i = 1$

4.6.4 Impact of Signal-to-Noise Ratio SNR_i on Accuracy

We present another set of simulation results in Figs. 4.9 and 4.10, to study the impact of signal-to-noise ratio on estimation accuracy. The ability of tolerating heavy noise is important, which makes the designed estimator more flexible to use in practice. First, we evaluate the performance of FMSK, by comparing Figs. 4.6 and 4.9 which configure the signal-to-noise ratio to 1 and 0.4, respectively. They show that the accuracy of FMSK degrades severely as the noise level increases. Its estimation even becomes biased in the last subfigure of Fig. 4.9. Second, we evaluate the noise toleration ability of our multi-virtual bitmap method, by comparing Figs. 4.8 and 4.10 which configure the signal-to-noise ratio to 1 and 0.4, respectively. The two figures show that the accuracy of our method also degrades, which is consistent with the analysis results in Fig. 4.2. However, the degree of degradation is pretty modest, and our method can still render satisfactory estimation accuracy when the signal-to-noise ratio is only 0.4 in Fig. 4.10.

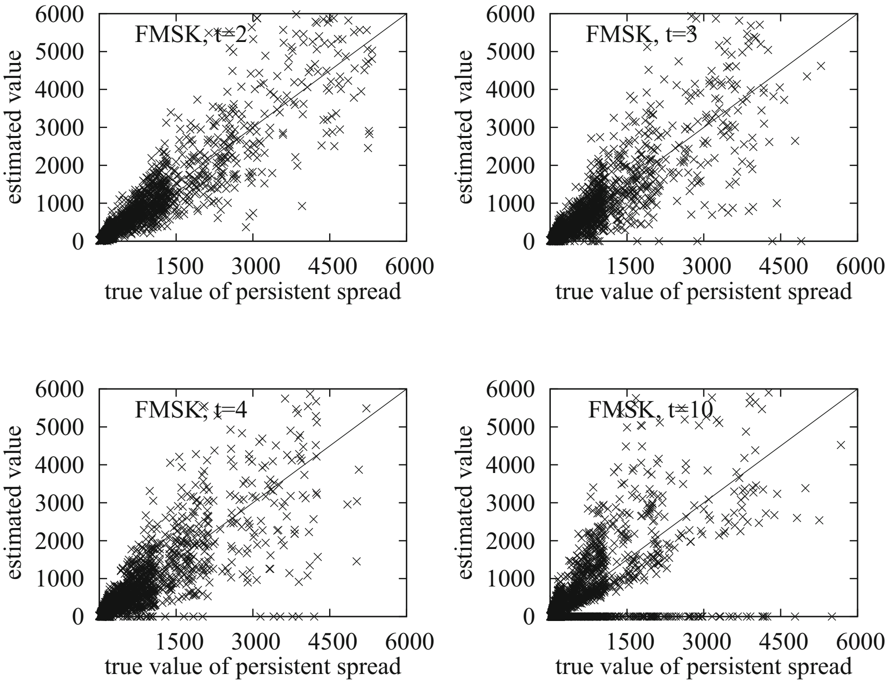


Fig. 4.9 Persistent spread estimation using FMSK method, with $\text{SNR}_i = 0.4$ and number of periods $t = 2, 3, 4, 10$

4.7 Related Work

An important branch of network traffic measurement is the passive measurement techniques, which use built-in components of a router or switcher to silently watch the traffic as it passes by. The traversed packets, according to certain fields in the packet header, are classified into different categories, each of which is called a *flow*. For an individual flow, several kinds of measurements can be taken, including the *flow size* (i.e., the number of packets or bytes or occurrences of certain events in one measurement period) [7], the *flow spread* (i.e., the number of distinct flow elements) [3, 10, 13]. This chapter introduces a new problem of measuring the flow's *persistent spread* (i.e., the number of distinct elements that persist through t time periods), which can be used to detect the long-term stealthy network activities in the background of transient behavior of legitimate users.

Our problem of persistent spread estimation has practical meanings to detect stealthy network activities that last for long periods, including stealthy DDoS attacks, stealthy network scan, and server popularity forging, just to name a few. For this problem, there is a related work which detects the stealthy network scan [6]. It, however, works in spatial domain and detects the presence of a set of hosts that

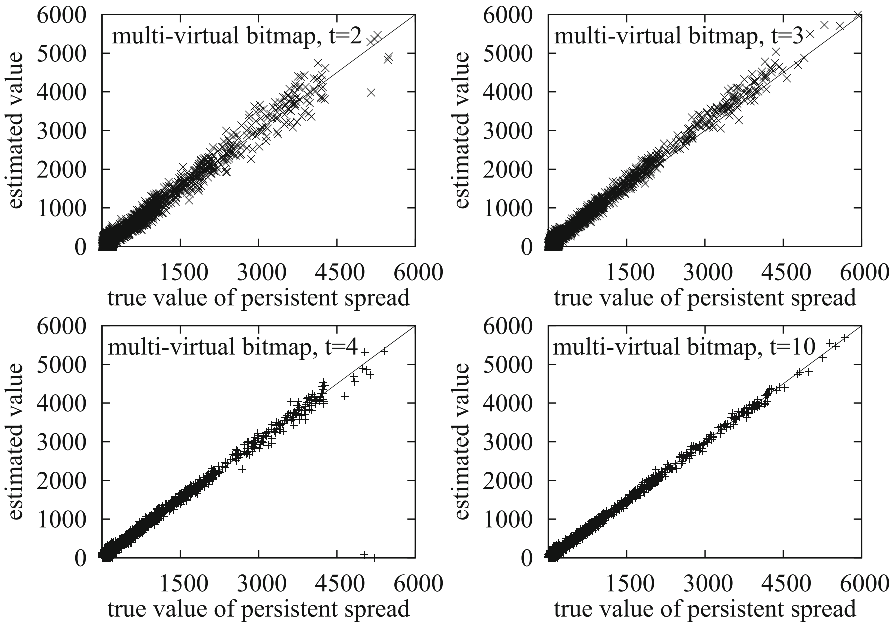


Fig. 4.10 Persistent spread estimation using multi-virtual bitmap algorithm, with $\mathrm{SNR}_i = 0.4$

connect to a sufficiently large number of unique destinations within a given time window. In contrast, we detect the network scan in temporal domain, and check whether a source node probes different network sections at different time windows. Moreover, our work is a generalized primitive that can detect many other kinds of stealthy activities.

To measurement the persistent spread for each flow, the key challenge is the requirement of low memory cost, due to the limited size of on-chip SRAM. The estimators of per-flow measurements [2, 3, 10] allocate each flow a separated equal-sized data structure. These methods ignore the fact that the flow spreads are extremely imbalanced, in real-world network traffic. Some flows are "elephant flows" whose spreads can be thousands of times larger than those of small flows. As a result, their counting data structures, when dealing with elephant flows, become overly dense due to the injection of too many elements. It may appear that, if implementing the counters by Flajolet–Martin sketches [4] or HyperLogLog [5], they can work well with the elephant flows. However, the accuracy of these modern cardinality estimators strongly depends on the space allocated for one estimator. When there are too many flows, the memory available for one estimator is extremely limited.

It is desirable that elephant flows borrow bits from small flows, to improve their estimation accuracy and extend the operating range, To implement the bit lending, a viable solution is to let the bits of virtual bitmaps uniformly distribute in the allocated SRAM space, which have been illustrated in Fig. 4.4. Although this idea of virtual bitmap has been discussed in literature [8, 13], the previous works deal with only one virtual bitmap for each flow, in order to estimate the flow's spread. In contrast, we consider t virtual bitmaps together (collected in t time periods), and estimate their intersection, i.e., the number of elements that persist through the t periods.

For the problem of persistent spread estimation, a critical design choice is what data structure we should adopt to record the information for each flow. The literature in [2] uses a continuous variant of Flajolet–Martin sketches. But we choose the bitmap [12], for the following two reasons. The bitmap structure can achieve higher accuracy than FM sketches and HyperLogLog, if given sufficient memory [9]. This structure, if further enhanced by our multi-virtual bitmaps, can extend the operating range to be sufficiently large for our application.

4.8 Summary

In this chapter, we have presented a new primitive for passive network measurement, called *persistent spread estimation*, which can detect the long-term stealthy network activities in the background of short-term activities of legitimate users. To solve this problem in tight memory space, this chapter has presented a compact data structure called *multi-virtual bitmaps*, which is suitable to deploy in the size-limited on-chip SRAM of high-speed routers. The simulation shows that our estimator can provide satisfactory accuracy and operating range, using small memory of less than 1 bit per-flow element.

Our estimator brings two other advantages as compared with previous work. Its estimation accuracy improves as the number of measurement periods increases, because our method can more effectively filter the short-term behavior of legitimate users. Its operating range of producing effective measurements has been extended if compared with bitmap method. The latter benefit originates from our data structure named multi-virtual bitmaps, which permits elephant flows to "borrow" bits from mouse flows by sharing bits in a common bit pool. These advantages have been verified by both analysis and experimental results.

Appendix 1: Bias Analysis of Bitmap-Based Spread Estimator

We prove that our bitmap-based estimator f_t in Definition 1 is asymptotically unbiased. The estimator f_t is obtained in Sect. 4.3 by solving the following equation set, which has $t + 1$ equations and $t + 1$ unknowns (i.e., P^* and P_i):

$$E(Z_i) = P^* P_i \qquad (1 \le i \le t)$$

$$E(Z^*) = P^* - P^* \prod\nolimits_{1 \le t \le t} (1 - P_i)$$

Getting P^* by solving the above equation set can be regarded a variant of maximum likelihood estimation. In the process, the only operations that will produce estimation bias are the replacement of expectations $E(Z_i)$ and $E(Z^*)$ by the observations Z_i and Z^*. We prove that, when the bitmap size m is large enough, the replacement produces negligibly small error, and the estimator f_t thus is asymptotically unbiased.

In bitmap B_i of the ith period, the number of zero bits mZ_i follows binomial distribution, since different bits are mutually independent roughly. This binomial distribution can be approximated as Gaussian distribution, when the array size m is sufficiently large [12]. For this Gaussian distribution, its mean value $E(mZ_i)$ is $me^{-\frac{n_i}{m}}$, and its variance is

$$Var(mZ_i) = m\,e^{-\frac{n_i}{m}}(1-(1+\tfrac{n_i}{m})\,e^{-\frac{n_i}{m}}).$$

Then, we know that the variance $Var(Z_i)$ approaches zero when m is sufficiently large. For similar reasons that the zero ratio Z^* is some kind of stochastic averaging in bitmap B^*, the variance $Var(Z^*)$ approaches zero asymptotically.

Appendix 2: Variance of Bitmap-Based Spread Estimator

We prove the estimator variance in Theorem 3. The likelihood function of persistent spread n^* using observation Y^* is

$$\mathscr{L}(n^* \mid Y^*) = (1-\tilde{P})^{Y^*} \cdot \tilde{P}^{m-Y^*}, \tag{4.17}$$

where $Y^* = mZ^*$ is the number of zero bits in bitmap B^*, and

$$\begin{aligned}\tilde{P} &= (1-P^*) + P^* \textstyle\prod_{1\le i\le t}(1-P_i)\\ &= \big(1-e^{-\frac{n^*}{m}}\big) + e^{-\frac{n^*}{m}} \textstyle\prod_{1\le i\le t}\big(1-e^{-\frac{n_i-n^*}{m}}\big).\end{aligned} \tag{4.18}$$

The meaning of (4.17) is the probability of observing Y^* zero-state bits and $m-Y^*$ one-state bits in the intersection bitmap B^*, given the facts of persisting spread n^* and the signal-to-noise ratios SNR_i in each period. The symbol $\tilde{P}$ denotes the probability for a bit to be one in bitmap B^*.

For any estimator of n^* based on the observation Y^* and $m-Y^*$, its variance satisfies the Cramér-Rao inequality below:

$$Var(\hat{n^*} \mid Y^*) \ge \tfrac{1}{I(n^*)},$$

where $I(n^*)$ is the Fisher information which can be calculated using the likelihood function $\mathscr{L}$ in (4.17).

$$I(n^*) = -E\Big[\tfrac{\partial^2 \ln \mathscr{L}(n^* \mid Y^*)}{(\partial n^*)^2}\Big]$$

For the log-likelihood function $\ln \mathscr{L}$ in (4.17), its first-order derivative and second-order derivative are as follows:

$$\frac{\partial \ln \mathscr{L}}{\partial n^*} = \frac{1}{\mathscr{L}} \frac{\partial \mathscr{L}}{\partial n^*} = \frac{\partial \tilde{P}}{\partial n^*}\left(\frac{m-Y^*}{\tilde{P}} - \frac{Y^*}{1-\tilde{P}}\right)$$

$$\frac{\partial^2 \ln \mathscr{L}}{(\partial n^*)^2} = \frac{\partial^2 \tilde{P}}{(\partial n^*)^2}\left(\frac{m-Y^*}{\tilde{P}} - \frac{Y^*}{1-\tilde{P}}\right) - \left(\frac{\partial \tilde{P}}{\partial n^*}\right)^2\left(\frac{m-Y^*}{\tilde{P}^2} + \frac{Y^*}{(1-\tilde{P})^2}\right)$$

Because the expected value of Y^* is $E(Y^*) = m(1 - \tilde{P})$, the expected value of $\frac{m-Y^*}{\tilde{P}} - \frac{Y^*}{1-\tilde{P}}$ equals zero. Hence,

$$I(n^*) = -E\left[\frac{\partial^2 \ln \mathscr{L}(n^* \mid Y^*)}{(\partial n^*)^2}\right] = \left(\frac{\partial \tilde{P}}{\partial n^*}\right)^2\left(\frac{m}{\tilde{P}} + \frac{m}{1-\tilde{P}}\right). \tag{4.19}$$

The first-order derivative $\frac{\partial \tilde{P}}{\partial n^*}$ required by the above equation can be derived from (4.18), assuming that the signal n^* is independent with the noise $n_i - n^*$ (i.e., $\frac{\partial(n_i-n^*)}{\partial n^*} = 0$).

$$\frac{\partial \tilde{P}}{\partial n^*} = \frac{1}{m}\left[e^{-\frac{n^*}{m}} - e^{-\frac{n^*}{m}} \prod_{1\le i\le t}\left(1 - e^{-\frac{n^*}{m}\frac{1}{\mathrm{SNR}_i}}\right)\right] = \frac{1}{m}(1 - \tilde{P})$$

Finally, by replacing $\frac{\partial \tilde{P}}{\partial n^*}$ in (4.19) with $\frac{1}{m}(1 - \tilde{P})$ and then using the relation $Var(\hat{n^*} \mid Y^*) \ge \frac{1}{I(n^*)}$, we can obtain the inequality for estimator variance in Theorem 3.

References

1. Casella, G., Berger, R.L.: Statistical Inference, 2nd edn. Duxbury Press, Pacific Grove (2002)
2. Chen, A., Cao, J., Bu, T.: A simple and efficient estimation method for stream expression cardinalities. In: Proceedings of the 33rd International Conference on Very Large Data Bases, VLDB '07, pp. 171–182 (2007)
3. Estan, C., Varghese, G., Fish, M.: Bitmap algorithms for counting active flows on high-speed links. IEEE/ACM Trans. Netw. (ToN) **14**(5), 925–937 (2006)
4. Flajolet, P., Martin, G.N.: Probabilistic counting algorithms for database applications. J. Comput. Syst. Sci. **31**(2), 182–209 (1985)
5. Flajolet, P., Fusy, E., Gandouet, O., Meunier., F.: HyperLogLog: the analysis of a near-optimal cardinality estimation algorithm. In: Proceedings of the AOFA: International Conference on Analysis of Algorithms (2007)
6. Gao, Y., Zhao, Y., Schweller, R., Venkataraman, S., Chen, Y., Song, D., Kao, M.: Detecting stealthy spreaders using online outdegree histograms. In: Proceedings of the IEEE IWQoS, pp. 145–153 (2007)
7. Lu, Y., Montanari, A., Dharmapurikar, S., Kabbani, A., Prabhakar, B.: Counter braids: a novel counter architecture for per-flow measurement. In: Proceedings of the ACM SIGMETRICS (2008)

8. Marold, A., Lieven, P., Scheuermann, B.: Distributed probabilistic network traffic measurements. In: 17th GI/ITG Conference on Communication in Distributed Systems (KiVS), vol. 17, pp. 133–144 (2011)
9. Metwally, A., Agrawal, D., Abbadi, A.E.: Why go logarithmic if we can go linear? Towards effective distinct counting of search traffic. In: Proceedings of the EDBT (2008)
10. Roesch, M.: Snort—lightweight intrusion detection for networks. In: Proceedings of 13th Systems Administration Conference, USENIX (1999)
11. The CAIDA UCSD Anonymized 2013 Internet Traces - January 17 (2013). http://www.caida.org/data/passive/passive_2013_dataset.xml
12. Whang, K.Y., Vander-Zanden, B.T., Taylor, H.M.: A linear-time probabilistic counting algorithm for database applications. ACM Trans. Database Syst. **15**(2), 208–229 (1990)
13. Yoon, M., Li, T., Chen, S., Peir, J.K.: Fit a spread estimator in small memory. In: Proceedings of the IEEE INFOCOM (2009)