Don't Work on Individual Data Plane Algorithms. Put Them Together!

Chen Qian, Shouqian Shi, Xiaofeng Shi, Minmei Wang
University of California, Santa Cruz
Santa Cruz, California, USA
{cqian12, sshi27, xshi24, mwang107}@ucsc.edu

ABSTRACT

Algorithms and data structures for data plane network functions have been extensively studied in the literature. Recently various compact data structures and algorithms have been used in data plane to achieve less memory cost and higher throughput. However, most of these studies only focus on individual network functions, such as packet forwarding information base (FIB), traffic measurement, and load balancing. To our knowledge, no study has been conducted to design compact data structures and algorithms for multiple and co-located network functions. We argue that there is a huge space of optimization if we design algorithms and data structures considering multiple co-located network functions, compared to designing them individually. It is because many of them share similar design goals and building blocks. We use two recently published methods as examples and present a new memory-compact design that serves both FIB and traffic measurement functions by a novel integration of the two methods. The preliminary results show that the new design can achieve almost 2x throughput compared to running them individually while achieving a higher accuracy of measurement using the same memory. In addition, we will discuss potential research directions and challenges.

CCS CONCEPTS

• **Theory of computation → Sketching and sampling**; **Bloom filters and hashing**; • **Networks → Data path algorithms**.

KEYWORDS

Data plane algorithms; Hashing; Sketches

ACM Reference Format:
Chen Qian, Shouqian Shi, Xiaofeng Shi, Minmei Wang. 2020. Don't Work on Individual Data Plane Algorithms. Put Them Together!. In *Proceedings of the 19th ACM Workshop on Hot Topics in Networks (HotNets '20), November 4–6, 2020, Virtual Event, USA.* ACM, New York, NY, USA, 7 pages. https://doi.org/10.1145/3422604.3425932

1 INTRODUCTION

Data plane algorithms and data structures of computer networks have been studied extensively, such as those for packet forwarding, packet measurement and monitoring, load balancing, firewalls, network address translation (NAT), and other network functions. Recent advances of programmable networking [3, 8, 17, 25, 33, 35, 37] use software, running on general-purpose computers or programmable switches, to perform network forwarding and functions, brings tremendous advantages and conveniences for designing and implementing new data plane algorithms and data structures.

We have witnessed a line of extensive studies of using space-compact algorithms and data structures for packet measurement and monitoring [12, 15, 22, 29–31, 43, 45, 50]. These data structures include variants of cardinality-estimation sketches [4, 6, 19] and frequency-estimation sketches [9, 12]. Meanwhile we have also noticed a line of research of applying other space-compact algorithms and data structures for network forwarding information base (FIB) [13, 20, 26, 34, 38, 44, 46, 48, 49] and load balancing [14, 32, 47]. These data structures include variants of Bloom filters [7], cuckoo hashing [36], Bloomier filters [10, 11], and a recent design called Ludo hashing [38]. Both research topics frequently appeared in recent main-stream networking conferences such as SIGCOMM, SIGMETRICS, NSDI, ICNP, INFOCOM, and ANCS. An interesting observation is that, although both packet measurement and forwarding are necessary network functions and many programmable switches/routers that support packet measurement should also support packet forwarding, none of these studies in the first line of research has considered optimizing packet forwarding while designing their methods. Similarly, none of the studies in the second line of research has considered the co-existence of packet measurement. One natural challenge is, since many of these designs set their goal as achieving line rate on hardware or software switches, will the line rate still be preserved if a switch needs to execute both measurement and forwarding – a very common situation in practice? To our knowledge, no study has been conducted to design compact data structures and algorithms for multiple and co-existed network functions from the above ones (e.g., one data structure and algorithm to support both packet measurement and forwarding).

We argue that there is **a huge space of optimization** if we design algorithms and data structures by considering multiple co-located network functions, compared to designing them individually. At the very least, we can get an immediate benefit by putting them together in the following example. Sketches, Bloom filters, and hash tables all require computing multiple universal hash functions. These hash functions must be independent to satisfy certain properties of these algorithms. However, they do not have to be independent across two different functions. One hash value can be used in a measurement sketch and used again in a forwarding table lookup, since they are independent functions by themselves. Hence the number of hash computations can be reduced significantly compared to executing them separately. It has been reported

HotNets '20, November 4–6, 2020, Virtual Event, USA

ACM ISBN 978-1-4503-8145-1/20/11.
https://doi.org/10.1145/3422604.3425932

that the number of per-packet hash computations contributes to the bottleneck overhead that prevents data plane packet processing from achieving the line rate [29]. The simple idea above can definitely help to achieve or at least be closer to the line rate when we need to run both forwarding and measurement functions. The optimization space is more than this single one, as we will explain later.

We expect a great future of optimizing algorithms for co-existed network functions because:

(1) The data plane algorithms and data structures for different network functions usually share similar design goals. First, they require *space efficiency*. It is because they are hosted in special hardware (e.g., SRAM) or high levels of the memory hierarchy (e.g., cache), where the memory is fast, small, expensive, and power-hungry. Second, they need to be fast enough to achieve *line rate*. Third, they should support *dynamic updates* to work in practice.
(2) These algorithms and data structures also share similar computation steps. For example, we explained earlier that they all need to compute multiple independent hash functions.
(3) These algorithms and data structures can be co-located to reduce space cost and/or reduce the number of memory accesses per packet, another main overhead of packet processing [29]. For example, it is possible to store the measurement sketch and the forwarding port of a flow together in a same memory unit to reduce the number of random memory accesses.

To demonstrate the power of consolidated design of network functions, we study two very recent papers, one for FIBs and the other for measurement sketches, from the program of *ACM SIGMETRICS* (June 2020) [38, 50] and published by two different groups. We present a new data plane design that serves both memory-compact FIB and traffic measurement functions by a novel integration of the two methods. The preliminary results show that the new design can achieve almost 2x throughput compared to running them individually while achieving higher accuracy of measurement in theory. In addition, we will discuss potential research directions towards finding better algorithm and data structure designs that support multiple network functions. We believe this paper introduces a new research direction that has potential big impacts to networking research.

2 RELATED WORK

Since data plane fast memory is precious resource, extensive studies have been conducted to design memory-compact data plane algorithms and data structures.

Sketching algorithms for traffic measurement. Sketches are important tools for network measurement tasks on programmable software/hardware switches that have limited memory [12, 15, 22, 29–31, 43, 45, 50]. They are able to conduct tasks such as estimating the sizes (number of packets) and cardinalities (number of distinct elements) of network traffic flows and find heavy hitters (elephant flows). Count-Min Sketches [12] are designed to estimate the flow sizes with memory/accuracy trade-offs. OpenSketch [45] is one of the first works that implement sketches on programmable switches for traffic measurement. UnivMon [30] provides a more generic

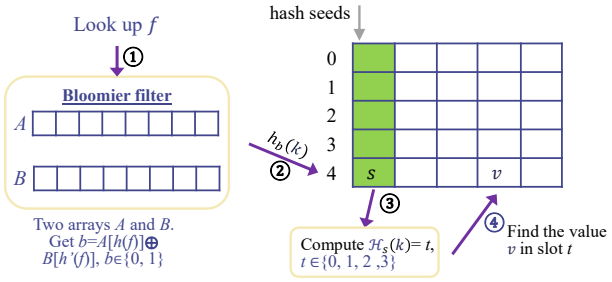

Figure 1: Ludo hashing

algorithm design, in which the control plane can support a wide range of estimation algorithms for different application-level measurement tasks. SketchVisor [22] improves the performance of flow measurement sketches by introducing a fast path (a separate sketch with smaller memory) to process the top-k largest flows. NitroSketch [29] is another design to solve the processing speed issues of sketches using sampling. Recently a generalized sketch family has been proposed to achieve various measurement tasks [50].

Memory-efficient FIBs. A forwarding information base (FIB) on a network router or switch is a fundamental function to process every packet. It tells the forwarding action (which port to send the packet or drop it) based on the lookup result of the packet ID (e.g., flow ID, four tuple, or destination address). Buffalo [44] is an layer-two enterprise network switch design based on Bloom filters. CuckooSwitch [49] is a software switch based on cuckoo hash tables [36]. SetSep [16, 48] is a lookup table that uses brute force to resolve collisions, with applications of LTE packet forwarding [48]. Concise [46] is a memory-efficient FIB design using Bloomier filters [10, 11] and the update method called Othello hashing. Shi *et al.* present the comparison among FIB designs using various compact data structures [39]. Ludo hashing [38] is a recent work that achieves the smallest memory cost of dynamic FIB data structures among existing work.

Other network functions using compact data structures. Cuckoo hash tables and Othello have been used for difference cloud load balancer designs [14, 32, 40]. Bloom filters have been used for cloud firewalls [23].

3 A CONSOLIDATED ALGORITHM DESIGN

We present a new data plane design that serves both memory-compact FIB and traffic measurement functions by a novel integration of the two methods [38, 50] that were recently presented on SIGMETRICS on June 2020.

3.1 Background

We first briefly described the algorithms and data structures used in [38, 50].

Ludo hashing is a key-value lookup engine with very small memory cost [38]. For each key, it can return the corresponding value of the key. Hence it can be used as a FIB on memory-limited devices. For each packet, depending on the network operation requirement, the lookup key can be the destination address for destination-based routing or flow ID (such as the four tuples) for flow-based routing. The returned value is the index for forwarding

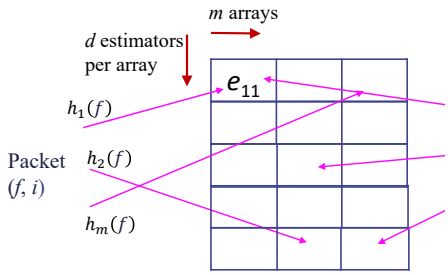

Figure 2: cSketch

actions (such as which port to forward or drop th
use f to denote the key for packet forwarding loc
Fig. 1, the data structure of Ludo hashing is a tup
where T is an array of buckets (on the right side
bucket includes a hash seed s and four slots storing up to 4 v
h_0 and h_1 are two uniform hash functions; O is a Bloomier
[10, 46] that maintains two bitmaps A and B and returns 1-bit
to indicate whether a key f is mapped to bucket $h_0(f)$ or h
and $\mathcal{H}$ is a universal hash function family. The query of a
will output the value v_f. Ludo hashing will query the Blo
filter first to get whether the value is stored in bucket $h_0($
$h_1(f)$ and then determine the slot index t that stores the val
computing $t = \mathcal{H}_s(f)$ where s is the seed stored in this bucket
and $t \in \{0, 1, 2, 3\}$. Finally, the value in slot t of bucket $h_b(f)$ is
returned as v_f. Other designs, such as how to construct and update
the data structure and control plane/data plane communication, are
skipped here.

Zhou *et al.* present a generalized sketch families for traffic measurement [50]. One proposed sketch is called cSketch, which is an extension of the well-known CountMin Sketch (CM-Sketch) [12]. As shown in Fig. 2, cSketch maintains m arrays, each of which includes d estimators. So there are md estimators from e_{11} to e_{md}. Each estimator provides the approximate count of the flow f that maps to this estimator. An estimator can be implemented by a bitmap [41], FM sketch [19], Hyperloglog sketch [18], or simply an integer counter. Each packet is identified by $< f, i >$. If cSketch wants to count the size of flows to identify elephant flows, f can be set to the four tuples, and i can be the TCP sequence number and other fields that differ among different packets. If cSketch wants to count the 'flow spread', i.e., how many sources have been sent to a particular destination, for applications like DDoS detection, then f can be the destination IP/MAC address and i can be the source IP address. For each packet, for each array j, cSketch computes $h_j(f)$ and update the $h_j(f)$-th estimator. To query the size of a flow, cSketch will return the minimum value of the m estimators of the flow f.

3.2 Integrating Ludo and cSketch

We find that for each packet, Ludo needs to compute four independent hash functions h, h', one of h_0 and h_1, and $\mathcal{H}$. It also needs three random memory accesses to read one bit from A, one bit from B, and a bucket in T. For cSketch, it needs to compute at least m independent hash functions, one for each array. More hashes may

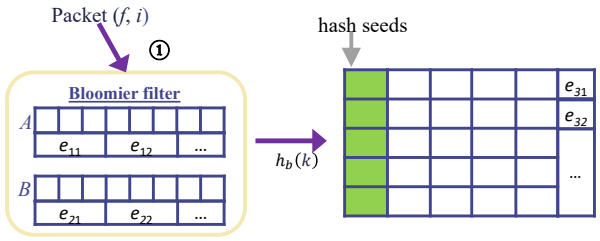

Figure 3: Consolidated Ludo and cSketch (LuCS)

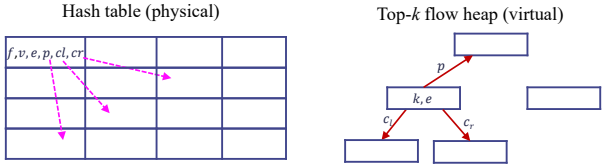

Figure 4: A table for maintaining top-k flows. The left side is the physical organization and the right side show a virtual heap in logic.

be needed to compute complicated estimators such as the Hyperloglog sketch. It also needs m random memory accesses. In practice, m is usually set to 3 to 5, because a large m significantly limits the packet throughput.

Our idea is to reduce the processing time by consolidating Ludo and cSketch is to share the hash results and co-locate the memory spaces of them for a same key f. Both Ludo and cSketch assume a hash value to be 32-bit long to support up to 2^{32} length of arrays/bitmaps/buckets. As shown in Fig. 3, we propose a design to co-locate 3 cSketch arrays with Ludo called LuCS, where each array j contains d estimators $e_{i1}, e_{i2}, ..., e_{id}$. In practice, n, the length of bitmap A, is much larger than d. We align each estimator with $k = n/d$ bits in the bitmap A. The estimator and the k bits are co-located in a same memory unit. In Fig. 3, $k = 3$. Each estimator will be responsible for counting all keys that are hashed to the 3 bits in A. Hence when the $h(f)$ bit in A is accessed, the estimator of f is simultaneously accessed. Similarly bitmap B can be co-located with another estimator array including $e_{21}, e_{22}, ..., e_{2d}$. The third array of cSketch can be put together with the bucket table. For example, in Fig. 3 an estimator is put at the end of each bucket, which counts all packets whose flow ID f is mapped to the bucket. Note the number of estimators d can be dynamically adjusted. Hence it is also possible to put one estimator for two buckets. It is not always possible to put a whole bucket in a 64-bit memory unit but they are in a same cacheline unit. Hence the time of memory access was also significantly reduced. Using this data structure, at least for three estimator arrays, very little extra complexity of hash computation and memory accesses are needed besides the original cost of Ludo. Note Ludo needs a fourth hash function $\mathcal{H}_s(f)$, whose result can be used for the fourth array if the fourth array is needed. A fifth array, if necessary, can be separately maintained since these arrays are mutually independent. The vSketch design proposed in [50] can also be co-located with Ludo similarly.

Consolidated table for top-k flows. Many traffic measurement tools require to track the top-k flows in a heap, in order to notify the control plane any elephant flow, such as UnivMon [30] and NitroSketch [29]. Maintaining the top-k flows is also beneficial for packet forwarding: Since it is well-known that network traffic is likely to follow power-law distribution [21, 27, 42], most packet forwarding actions are also among the top-k flows. Hence if we maintain a hash table that allows one hash computation and one memory access to retrieve the action, the overall lookup performance can be improved. Note the hash value used for the top-k table can be again used for Ludo because they are independent data structures. The bucket overflow of the hash table can be reduced by lowering the load factor. Since k is a small value, lowering the load factor does not significantly increase memory cost.

Assume a faction p of packets are in top-k, the time for computing a hash function is T_h, and the time for one random memory access is T_r. The overall time cost for each lookup with a top-k table is

$$(p + 3 - 3p)T_h + (p + 3 - 3p)T_r = (3 - 2p)T_h + (3 - 2p)T_r$$

If $p = 0.5$, the value is $2T_h + 2T_r$. If $p = 0.8$, the value is $1.4T_h + 1.4T_r$. They are much smaller than the original complexity $4T_h + 3T_r$.

We present a consolidated table for tracking and querying top-k flows, as shown in Fig. 4. The physical organization is a hash table with multiple buckets, each of which has s slots. We do not use Cuckoo hash table [36] because Cuckoo requires 1.5 memory accesses for each lookup among top-k and 2 memory accesses for each outside top-k. Since the hash table may have overflow, certain methods will be used to mitigate this problem, such as lowering the load factor and limited linear probing. Since k is a small number, it is relatively easy to resolve these problems by trading with some extra memory space. Each slot of the table contains the flow ID f, the forwarding action index v, the estimation of the flow size e, the parent node p in the heap, and the left and right children c_l and c_r. p, c_l, and c_r are represented by the indices of these nodes in the table. The virtual heap structure is formed by using the values of p, c_l, and c_r, as shown in the right side of Fig. 4.

Why does this design save memory compared to maintaining a table and a heap separately? f contributes to the majority storage in the table. A four tuple costs 96 bits, e may cost 20 to 30 bits, while each of others only cost no more than 10 bits. Maintaining the table and heap separately needs to store two copies of f hence it increases memory cost.

3.3 Analysis of LuCS

LuCS is a preferred design compared to using Ludo and cSketch individually because 1) it provides higher lookup speed and 2) it provides higher accuracy of flow size estimation at the same memory cost (or potentially smaller memory to reach a target accuracy).

Lookup time analysis. Assume the time for computing a hash function is T_h, the time for one random memory access is T_r, and the time for one memory write is T_w. Note that in a platform with memory hierarchy, T_r may not be fixed, but still the overall lookup time would be less with fewer memory accesses in theory. For every packet, the data plane wants to know the its forwarding action and its estimated size (to report if it is considered an elephant flow), Ludo requires $4T_h + 3T_r$ and cSketch requires $3T_h + 3T_r + T_w$, assuming $m = 3$ and each estimator is simply a counter. Hence they require $7T_h + 6T_r + T_w$ in total. On the other hand, LuCS only requires $4T_h + 3T_r + T_w$. There is another factor that would further reduce LuCS's lookup time. We know that in practice a large proportion of packets belong to the top-k flows. In platform with memory hierarchy such as software switches on commodity servers, it is desired that lookups to the top-k flows should hit the fast memory such as cache. The LuCS design makes the memory locations of top-k flows of both Ludo and cSketch co-located and hence it increases the cache hit rate. Running Ludo and cSketch individually will likely to cause more cache misses.

Accuracy analysis. We show that the estimator arrays co-located with Ludo actually achieves a **better accuracy** than using cSketch separately. It is known that CM-Sketches (and cSketch too) will over-count the size or spread of a flow, due to collisions among different flows. Let X_i denote the excess of an estimator array i of cSketch. Let f be a flow, c_f be the count of f (e.g., the size), and $\tilde{c}_f$ be the reported estimate of f. $Y_{i,j}$ be the indicator of the event $h_i(j) = h_i(f)$ and $a \neq j$ in cSketch. Let X_i denote the excess of an estimator array i of cSketch: $X_i = \tilde{c}_f - c_f$. Hence $X_i = \Sigma_j c_j Y_{i,j}$.

The first and second estimator arrays of LuCS are identical to the first two arrays of cSketch. Let us compare the estimators $e_{31}, e_{32}, ..., e_{3d}$ of LuCS of those in cSketch. Assume the number of all flows is n and the number of buckets in Ludo is $1.05n/4$. Hence $E[Y_{i,j}] = 3.8/n$. Let C be the count of all flows. Thus

$$E[X_i] = \Sigma_j 3.8 f_j/n = 3.8(C - c_f)/n$$

Based on Markov's inequality, we have

$$P\{X_i \geq \epsilon(C - c_f)/n\} \leq 3.8/\epsilon$$

On the other hand, let X' denote the excess of the third estimator array of LuCS and Y'_j be the indicator of the event $h_3(j) = h_3(f)$ and $a \neq j$ in LuCS. Note the number of collisions to f in Ludo is bounded by 3 with average 2.8, i.e., the other three values in a bucket. Hence we have

$$E[X'] = 2.8(C - c_f)$$

$$P\{X' \geq \epsilon(C - c_f)/n\} \leq 2.8/\epsilon$$

Hence using the same memory size and computation time, X' is smaller than X_i by utilizing the feature of Ludo hashing. Since we take the minimum count from the estimators from all arrays, the probability that the reported excess X is bigger than a particular value is the product of that probability from each array.

$$P\{X \geq \epsilon(C - c_f)/n\} = \Pi_i P\{X_i \geq \epsilon(C - c_f)/n\}$$

Hence the overall estimation of LuCS is also more accurate.

Compared to OpenFlow-style tables. An intuitive approach for co-located network functions is to apply OpenFlow-style tables [2] where each entry of the tables represents one flow or one type of flows. Following the matching fields of each entry, each "action" field can express a network function. The major weakness of this design is that its memory cost is very high (>356 bit per flow), compared to many compact data structures that costs 10 to 20 bits per flow [38, 46]. In addition, OpenFlow tables cannot be integrated with counting sketches because flows cannot shared counters.

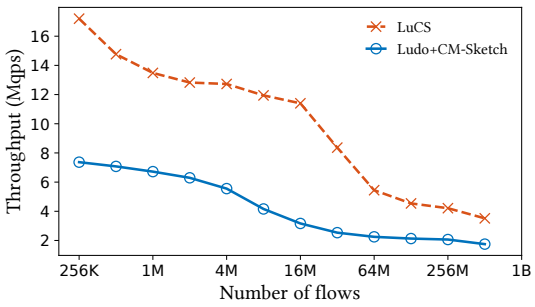

Figure 5: Throughput comparison with uniform traffic

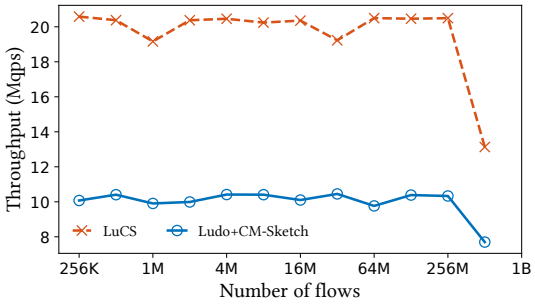

Figure 6: Throughput comparison with Zipfian traffic

3.4 Preliminary results

We implement LuCS and compare it with using both Ludo and CM-Sketch [12] (CM-Sketch can be considered a special case of cSketch). They were run in single thread on a commodity workstation with two Intel E5-2660 v3 10-core CPUs at 2.60GHz. The traffic workloads are in two types: in the uniform distribution and Zipfian distribution. For the uniform distribution, all flows are requested with an equal probability. For the Zipfian distribution, items are requested with biased probabilities, which better simulates the workload in most practical systems.

Figs. 5 and 6 show the throughput results, in millions of queries per second (Mqps), for uniform and Zipfian traffic respectively. The overall memory costs of LuCS and Ludo+CM-Sketch are the same. We find that LuCS achieves almost twice of the throughput of Ludo+CM-Sketch. The throughput results from both figures experience drops when the number of flows exceeds certain values, which indicate the memory cost exceeds the cache size. Compared Fig. 5 with Fig. 6, the throughput with Zipfian traffic is also higher than those of uniform traffic, because Zipfian causes fewer cache misses. The results are close to those from recent work that aims to achieve fast packet processing [29, 50]. The source code of LuCS is available for reproduction [1].

4 POSSIBLE RESEARCH DIRECTIONS

The research of algorithm designs for consolidated network functions is not limited to the above example. We identify possible research topics towards this field in this section.

4.1 Systematical design methodologies

As we stated, numerous algorithmic designs of network functions have been proposed in the past. Can we identify possible approaches to find those algorithms that can be consolidated. We propose some possible methodologies.

1. *Do these network functions apply to the same packet header fields.* Depending on the network operation requirements, different network functions may operate on different packet header fields. For example, in a network with Ethernet semantics, packet forwarding is based on the MAC address [24]. In a flow-based network, packet forwarding may be based on the four-tuple. To count the flow size, the hash key could be the four tuples. To count the flow spread, the hash key should be the destination address (IP or MAC). The first step of designing algorithms for consolidated network functions is to identify their operation targets and find those that share the same operation fields. In addition, some network functions operate on the content of a packet, such as content-based routing and distributed caching [28]. They can also be designed together to optimize network performance.

2. *Do these network functions share similar data structures.* Each compact network algorithm uses one or more data structures, most of which are hashing based. If two network functions obviously rely on similar data structures, it is a natural decision for a consolidated design. For example, BUFFALO [44] and CAESAR [34] use Bloom filters for packet forwarding and Bloom filters are also key building blocks of the bSketch [50]. A co-located data structure based on Bloom filters can be designed similar to LuCS presented in Section 3. Furthermore, even if some network functions do not use exactly the same data structures, like Lodu and cSketch, they can still be optimized by consolidation. The main reason is that many data structures require to map the keys uniformly to a bitmap, array, or table.

3. *Can one feature of a data structure be used by others.* Some data structures achieve certain features by paying some memory or computation cost. For example, Cuckoo hashing [36] uses two hashes and more space (load factor lower than 100%) to resolve collisions. Ludo hashing [38] uses 3.76 bits per key to resolve the collisions. These collision resolution results can also benefit sketches because the estimation errors are caused by collisions. As we show in Section 3 that putting the estimators with Ludo will actually decrease the error. It is interesting to explore other features that can be utilized by multiple data structures.

4. *Do these network functions track similar information.* The consolidated top-k flow table in Section 3 shows that some information can be tracked among multiple network functions and only be maintained in one copy. Similar information could be the popular contents, congested links, and overloaded servers. We will study how maintaining these information helps multiple network functions.

4.2 Update and coordination

Many compact data plane algorithms and data structures require the control plane coordination to maintain consistency and correctness, such as those in [30, 38, 44, 46, 48]. One important challenge is that some updates to these data plane algorithms cannot be immediately determined and reflected in the data plane, because the

	keys	benefit	possible combination
FIB+ flow-size sketch	flow ID	faster processing, higher accuracy	Ludo+cSketch
FIB+ flow-spread sketch	destination	faster processing, higher accuracy	(Ludo + cSketch) or (Buffalo [44] + bSketch [50])
LB+NAT	four-tuple	smaller memory, faster processing	Maglev [14] or Concury [40] + NAT
LB+sketches	four-tuple	faster processing	Maglev [14] or Concury [40] + cSketch
FIB+sketches with samples	flow ID or destination	less memory per node, higher accuracy	Ludo+NitroSketch [29]

Table 1: Examples of possible consolidation of NF algorithms. LB: load balancer; NAT: network address translation;

data plane either does not have enough information or does not have enough computation power. However, both packet measurement and forwarding require instant operations on the packets. One approach that works in practice is the maintain a separate table of the new arrival or changed flows, instead of putting them into the compact data structure. Then periodically the control plane will combine the temporal table into the compact data structure and allow the data plane to update according to the control plane update messages. More approaches should be explored under this topic.

4.3 Network-wide optimization

It is true that every network device needs to have the forwarding functions. However, for other network functions, a network may need only one or a few nodes to provide these functions. In addition, with programmable networking such as SDN, it is possible that multiple network nodes *collaboratively* complete certain network operations. For example, a flow can be monitored at any switch on its forwarding path. The SDN controller can optimize the network operation load distribution to all nodes based on the resource availability and cost on these nodes. We summarize the following challenges of network-wide optimization. **i)** Heterogeneity is ubiquitous in the network, i.e., different nodes and hosts have varied configuration. **ii)** Resource consumptions of different flows are highly coupled together in some cases. For example, in traffic measurement, a bitmap may be used for flow number estimation. If we double the bitmap size, the estimation accuracy does not double. Hence an optimized strategy is to allow different nodes maintain proper sizes of bitmaps, and assign the flow measurement tasks approximately evenly to the nodes. **iii)** Correlation between different tasks further complicates the optimization. First of all, some tasks may composite together to form a more complicated task, such as heavy-hitter detection [5]. Second, some computational resources could be saved if one flow has several tasks conducted at the same node. We notice that most existing designs of compact network functions are based on one single node and they pay little attention to the network-wide optimization. The key motivation of network-wide optimization is that, the increasing use of software switches enables a network function can be executed anywhere in the network. By using the entire network to complete these functions, the overhead on every single node can significantly be reduced. More research is expected to be conducted on this topic.

4.4 Generalizability of NF algorithm co-design

Following the principles discussed above, we suggest a number of possible consolidated data plane algorithms in Table 1. FIB+ flow-size/flow-spread sketches have been discussed in Sec. 3. In addition, a load balancer can be combined with a NAT to achieve both functions while storing the flow IDs in the lookup table only once. Hence this design reduces the memory cost. A load balancer could also be consolidated with sketches to count the flow sizes and the traffic load to different servers. We also propose a co-design of FIB and sampling-based sketches following the idea in Sec. 4.3. Sampling-based sketches such as NitroSketch [29] achieve the line-rate process but significantly increase the memory cost. We propose to embed the sketch function into different FIBs across the network while achieving line-rate on every forwarding node and distribute the memory cost to different nodes.

4.5 Make use of existing design modules

Consolidated data plane algorithms do not simply play the integration versus modularity game. In fact, they can be designed in a way that preserves the current modules of data plane algorithms. Most recently proposed data plane algorithms consist of two main components: 1) the data plane module that processes packets and 2) the control plane module that builds the data plane module and tracks the necessary information such as active flows and destination addresses, such as those in [29, 38, 40]. The data plane and control plane modules communicate via custom-defined control messages. In the proposed design, the data plane modules will be replaced by the consolidated data structures, but the control plane modules of the individual network functions are still preserved. A middle layer will be added in the control plane to use the output of the individual modules as inputs to produce the consolidated data structures.

5 CONCLUSION

In this paper, we introduce the opportunities of designing co-located and consolidated network functions using compact algorithms and data structures, to achieve fast packet processing, smaller memory, and/or higher accuracy of network measurement. We present LuCS that serves both memory-compact FIB and traffic measurement functions of two recently proposed methods Ludo hashing [38] and cSketch [50]. We show LuCS provides 2x throughput in experiments and higher accuracy in analysis, compared to running the two original methods. We also identify the potential research directions and challenges along this research direction. We believe this new field has potential big impacts on networking research.

6 ACKNOWLEDGMENT

The authors were partially supported by NSF Grants 1717948 and 1932447. We thank the anonymous reviewers and our shepherd Ben Leong for their constructive comments.

REFERENCES

[1] Implementation of LuCS in C++. https://github.com/QianLabUCSC/LuCS.

[2] OpenFlow Switch Specification 1.3.4. http://www.opennetworking.org/.

[3] Network Functions Virtualisation: Introductory White Paper. https://portal.etsi.org/nfv/nfv_white_paper.pdf, 2012.

[4] N. Alon, Y. Matias, and M. Szegedy. The space complexity of approximating the frequency moments. *J. Comput. Syst. Sci.*, 1999.

[5] N. Bandi, A. Metwally, D. Agrawal, and A. El Abbadi. Fast data stream algorithms using associative memories. In *Proc. of the ACM SIGMOD*, 2007.

[6] Z. Bar-Yossef, T. S. Jayram, R. K. D. Sivakumar, and L. Trevisan. Counting distinct elements in a data stream. In *Proc. 6th International Workshop on Randomization and Approximation Techniques in Computer Science*, 2002.

[7] B. H. Bloom. Space/time trade-offs in hash coding with allowable errors. *Communications of the ACM*, 13(7):422–426, 1970.

[8] P. Bosshart, D. Daly, G. Gibb, M. Izzard, N. McKeown, J. Rexford, C. Schlesinger, D. Talayco, A. Vahdat, G. Varghese, and D. Walker. P4: programming protocol-independent packet processors. *ACM SIGCOMM Computer Communication Review*, 2014.

[9] M. Charikar, K. Chen, and M. Farach-Colton. Finding frequent items in data streams. *Theor. Comput. Sci.*, 2004.

[10] D. Charles and K. Chellapilla. Bloomier Filters: A Second Look. In *Proc. of ESA*, 2008.

[11] B. Chazelle, J. Kilian, R. Rubinfeld, and A. Tal. The Bloomier Filter: An Efficient Data Structure for Static Support Lookup Tables. In *Proc. of ACM SODA*, pages 30–39, 2004.

[12] G. Cormode and S. Muthukrishnan. An improved data stream summary: the count-min sketch and its applications. *Journal of Algorithms*, 2005.

[13] S. Dharmapurikar, P. Krishnamurthy, and D. E. Taylor. Longest prefix matching using bloom filters. In *Proceedings of ACM SIGCOMM*, 2003.

[14] D. E. Eisenbud, C. Yi, C. Contavalli, C. Smith, R. Kononov, E. Mann-Hielscher, A. Cilingiroglu, B. Cheyney, W. Shang, and J. D. Hosein. Maglev: A Fast and Reliable Software Network Load Balancer. In *Proc. of USENIX NSDI*, 2016.

[15] C. Estan, G. Varghese, and M. Fisk. Bitmap Algorithms for Counting Active Flows on High Speed Links. In *Proceedings of the 3rd ACM SIGCOMM conference on Internet measurement*, pages 153–166, 2003.

[16] B. Fan, D. Zhou, H. Lim, M. Kaminsky, and D. G. Andersen. When cycles are cheap, some tables can be huge. In *Proc. of USENIX HotOS*, 2013.

[17] N. Feamster, J. Rexford, and E. Zegura. The road to SDN: An intellectual history of programmable networks. *ACM Queue*, 2013.

[18] P. Flajolet, É. Fusy, O. Gandouet, and F. Meunier. Hyperloglog: The Analysis of A Near-optimal Cardinality Estimation Algorithm. 2007.

[19] P. Flajolet and G. N. Martin. Probabilistic counting algorithms for data base applications. *J. Comput. Syst. Sci.*, 1985.

[20] A. Goel and P. Gupta. Small subset queries and bloom filters using ternary associative memories, with applications. In *Proc. of ACM SIGMETRICS*, 2010.

[21] L. Guo and I. Matta. The war between mice and elephants. In *Proc. of IEEE ICNP*, 2001.

[22] Q. Huang, X. Jin, P. P. Lee, R. Li, L. Tang, Y.-C. Chen, and G. Zhang. Sketchvisor: Robust network measurement for software packet processing. In *Proceedings of the Conference of the ACM Special Interest Group on Data Communication*, pages 113–126, 2017.

[23] A. R. Khakpour and A. X. Liu. First Step Toward Cloud-Based Firewalling. In *Proceedings of the 31st IEEE International Symposium on Reliable Distributed Systems (SRDS)*, 2012.

[24] C. Kim, M. Caesar, and J. Rexford. Floodless in SEATTLE: A Scalable Ethernet Architecture for Large Enterprises. In *Proc. of Sigcomm*, 2008.

[25] E. Kohler. *The Click Modular Router*. PhD thesis, Massachusetts Institute of Technology, 2000.

[26] D. Li, H. Cui, Y. Hu, Y. Xia, and X. Wang. Scalable data center multicast using multi-class Bloom filter. In *Proc. of IEEE ICNP*, 2011.

[27] X. Li and C. Qian. Low-Complexity Multi-Resource Packet Scheduling for Network Functions Virtualization. In *Proceedings of IEEE INFOCOM*, 2015.

[28] Z. Liu, Z. Bai, Z. Liu, X. Li, C. Kim, V. Braverman, X. Jin, and I. Stoica. DistCache: Provable Load Balancing for Large-Scale Storage Systems with Distributed Caching. In *Proc. of USENIX FAST*, 2019.

[29] Z. Liu, R. Ben-Basat, G. Einziger, Y. Kassner, V. Braverman, R. Friedman, and V. Sekar. NitroSketch: Robust and General Sketch-based Monitoring in Software Switches. In *Proc. of ACM SIGCOMM*, 2019.

[30] Z. Liu, A. Manousis, G. Vorsanger, V. Sekar, and V. Braverman. One Sketch to Rule Them All: Rethinking Network Flow Monitoring with UnivMon. In *Proc. of ACM SIGCOMM*, 2016.

[31] Y. Lu, A. Montanari, B. Prabhakar, S. Dharmapurikar, and A. Kabbani. Counter braids: A novel counter architecture for per-flow measurement. *ACM SIGMETRICS Performance Evaluation Review*, 36(1):121–132, 2008.

[32] R. Mao, H. Zeng, C. Kim, J. Lee, and M. Yu. SilkRoad: Making Stateful Layer-4 Load Balancing Fast and Cheap Using Switching ASICs. In *Proc. of ACM SIGCOMM*, 2017.

[33] N. McKeown, T. Anderson, H. Balakrishnan, G. Parulkar, L. Peterson, J. Rexford, S. Shenker, and J. Turner. Openflow: Enabling innovation in campus networks. *SIGCOMM Comput. Commun. Rev.*, 2008.

[34] M. Moradi, F. Qian, Q. Xu, Z. M. Mao, D. Bethea, and M. K. Reiter. Caesar: High-Speed and Memory-Efficient Forwarding Engine for Future Internet Architecture. In *Proceedings of ACM/IEEE ANCS*, 2015.

[35] B. A. A. Nunes, M. Mendonca, X.-N. Nguyen, K. Obraczka, and T. Turletti. A Survey of Software-Defined Networking: Past, Present, and Future of Programmable Networks . *IEEE Communications Surveys and Tutorials*, 2014.

[36] R. Pagh and F. F. Rodler. Cuckoo hashing. *Journal of Algorithms*, 2004.

[37] B. Pfaff, J. Pettit, T. Koponen, E. Jackson, A. Zhou, J. Rajahalme, J. Gross, A. Wang, J. Stringer, P. Shelar, K. Amidon, and M. Casado. The Design and Implementation of Open vSwitch. In *Proc. of USENIX NSDI*, 2015.

[38] S. Shi and C. Qian. Ludo Hashing: Compact, Fast, and Dynamic Key-value Lookups for Practical Network Systems. In *Proceedings of ACM SIGMETRICS*, 2020.

[39] S. Shi, C. Qian, and M. Wang. Re-designing Compact-structure based Forwarding for Programmable Networks. In *Proc. of IEEE ICNP*, 2019.

[40] S. Shi, Y. Yu, M. Xie, X. Li, X. Li, Y. Zhang, and C. Qian. Concury: A Fast and Light-weight Software Cloud Load Balancer. In *Proceedings of ACM SOCC*, 2020.

[41] K. Whang, B. T. Vander-Zandan, and M. H. Taylor. A linear-time probabilistic counting algorithm for database applications. *ACM Transactions on Database Systems*, 1990.

[42] W. Willinger, M. S. Taqqu, R. Sherman, and D. V. Wilson. Self-similarity through High-variability: Statistical Analysis of Ethernet LAN Traffic at the Source Level. *IEEE/ACM Transactions on Networking*, 1997.

[43] T. Yang, J. Jiang, P. Liu, Q. Huang, J. Gong, Y. Zhou, R. Miao, X. Li, and S. Uhlig. Elastic Sketch: Adaptive and Fast Network-wide Measurements. In *Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication*, pages 561–575, 2018.

[44] M. Yu, A. Fabrikant, and J. Rexford. BUFFALO: Bloom filter forwarding architecture for large organizations. In *Proc. of ACM CoNEXT*, 2009.

[45] M. Yu, L. Jose, and R. Miao. Software defined traffic measurement with OpenSketch. In *Proc. of USENIX NSDI*, 2013.

[46] Y. Yu, D. Belazzougui, C. Qian, and Q. Zhang. Memory-efficient and Ultra-fast Network Lookup and Forwarding using Othello Hashing. *IEEE/ACM Transactions on Networking*, 2018.

[47] Y. Yu, X. Li, and C. Qian. SDLB: A Scalable and Dynamic Software Load Balancer for Fog and Mobile Edge Computing. In *Proc. of ACM SIGCOMM Workshop on Mobile Edge Computing (MECCOM)*, 2017.

[48] D. Zhou, B. Fan, H. Lim, D. G. Andersen, M. Kaminsky, M. Mitzenmacher, R. Wang, and A. Singh. Scaling up clustered network appliances with scalebricks. In *SIGCOMM*, 2015.

[49] D. Zhou, B. Fan, H. Lim, M. Kaminsky, and D. G. Andersen. Scalable, High Performance Ethernet Forwarding with CuckooSwitch. In *Proc. of ACM CoNEXT*, 2013.

[50] Y. Zhou, Y. Zhang, C. Ma, S. Chen, and O. Odegbile. Generalized Sketch Families for Network Traffic Measurement. In *Proceedings of ACM SIGMETRICS*, 2020.