A Comparison of Performance and Accuracy of Measurement Algorithms in Software

Omid Alipourfard
Yale University
omid.alipourfard@yale.edu

Masoud Moshref
Barefoot Networks
mmoshref@barefootnetworks.com

Yang Zhou
Peking University
zhou.yang@pku.edu.cn

Tong Yang
Peking University
yang.tong@pku.edu.cn

Minlan Yu
Yale University
minlan.yu@yale.edu

ABSTRACT

Many network functions are moving from hardware to software to get better programmability and lower cost. Measurement is critical to most network functions because getting detailed information about traffic is often the first step to make control decisions and diagnose problems. The key challenge for measurement is how to keep a large number of counters while processing packets at line rate. Previous work on measurement algorithms mostly focuses on reducing memory usage while achieving high accuracy. However, software servers have plenty of memory but incur new challenges of achieving both high performance and high accuracy. In this paper, we revisit the measurement algorithms and data structures under the new metrics of performance and accuracy. We show that saving memory through extra computation is not worthwhile. As a result, a linear hash table and count array outperform more complex data structures such as Cuckoo hashing, Count-Min sketches, and heaps in a variety of scenarios.

KEYWORDS

Network Measurement; Software Switches; Performance Tuning

ACM Reference Format:
Omid Alipourfard, Masoud Moshref, Yang Zhou, Tong Yang, and Minlan Yu. 2018. A Comparison of Performance and Accuracy of Measurement Algorithms in Software. In *Proceedings of (SOSR '18).* ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3185467.3185475

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
SOSR '18, March 28-29, 2018, Los Angeles, CA, USA
© 2018 Copyright held by the owner/author(s). Publication rights licensed to the Association for Computing Machinery.
ACM ISBN 978-1-4503-5664-0/18/03…$15.00
https://doi.org/10.1145/3185467.3185475

1 INTRODUCTION

To reduce the cost and management complexity of hardware switches and middleboxes, there is a growing need of moving network functions to software. For example, today, data centers often run load balancing and firewalls in software [52, 58], and ISPs have started to deploy virtualized network functions (VNFs) to replace their hardware boxes [4].

Measurement is a key component in many network functions: for detecting anomalies (e.g., heavy hitters, superspreaders), profiling traffic of applications, or inspecting individual packets (DPI). Other network functions such as load balancing and traffic engineering also rely on accurate measurement of traffic statistics [14]. Measurement tasks can run on baremetal, e.g., a software switch [53], or inside containers either standalone or as part of another NFV, e.g., a load balancer container that detects and spreads heavy-hitter flows across all the backend servers [36].

To support measurement functions, we need to keep a large number of counters for individual packets and flows. Therefore, most measurement algorithms focus on how to store many counters with limited memory while retaining measurement accuracy, at the expense of more hash functions (e.g., Cuckoo hashing [51] and Count-Min sketch [22, 47]) or more computations (e.g., heaps). Even some recent proposals focusing on software measurement also target reducing memory usage [23, 31, 35, 48, 55, 59].

However, we argue that, in software, the key metric is not memory usage, but packet processing performance (i.e., throughput and latency). This is because modern servers have plenty of memory, an efficient caching hierarchy, and highly optimized compilers. Instead, the key challenge is to achieve high throughput and low latency. If we spend too many CPU cycles to fetch measurement data into cache and compute the right values and locations for counters, we may delay the packet processing and affect throughput. Note that the tail latency also matters because even if a few packets experience

long delay, the queue size increases which causes packet drops.

In this paper, we re-evaluate measurement algorithms in software with a focus on performance and accuracy metrics. We study three measurement tasks (heavy hitters, superspreaders, and change detection) on a variety of measurement algorithms (including hash tables, sketches, and heaps). Our key observations are:

1. We show that saving memory through extra computation is not worthwhile in achieving high performance and high accuracy for measurement in software. For example, using more hash functions in Cuckoo hashing or a Count-Min sketch provides worse performance than a linear hash table or a count array. Using more computationally intensive data structures (e.g., heaps) also hurts performance. Instead, to improve the accuracy, one can simply allocate larger memory to simple linear hash table or a count array while still achieving better performance than the other data structures with more computation. (Section 3)
2. Our conclusion holds for heavy hitter detection and other measurement tasks with different memory access patterns (superspreader detection) and more complex computation (change detection). It also holds for measurements with different entry sizes, value sizes, and traffic skews. (Section 4)
3. On a multicore setting, it is a bad idea to save memory by sharing resources across cores. Instead, we should maintain separate data structures across cores to avoid synchronization and aggregate the results during the reporting time. (Section 5)

In addition to the above observations, we discuss possible ways to improve measurement algorithms in software in Section 6. We describe related works in Section 7 and conclude the paper in Section 8.

2 BACKGROUND AND MOTIVATION

To support various network functions, we need a variety of measurement tasks such as heavy hitter detection, traffic change detection, and flow size distribution estimation. We observe that most of these tasks are often implemented using three classes of algorithms (Table 1). In this section, we give some backgrounds on these measurement tasks and algorithms and their design principles. We then motivate why it is important to re-evaluate these algorithms in the software context.

2.1 Three classes of measurement algorithms

We consider three classes of measurement algorithms: hash tables, sketches, heap/tree-based solutions. To illustrate their design principles, we take heavy hitter detection as an example. We define a heavy hitter as a source and destination IP address pair that sends traffic volume more than a pre-specified threshold. Heavy hitters are very useful for many management tasks. For example, operators can collocate chatty VMs (source-destination pairs with heavy traffic) in the same server or rack to save network bandwidth in data centers.

Hash tables: Hash tables compute a hash function for each key and use the result to locate a bucket in the array to store the key and its value. To handle hash collisions, many hash table designs such as linear hashing, Cuckoo hashing [51], or hopscotch hashing [33] probe a set of additional buckets to identify an empty bucket to hold the key. When the hash table has a high occupancy rate (load factor), finding an empty bucket takes multiple probing rounds, which leads to high packet processing delay and delay variance. We compare Cuckoo hashing that is commonly used for software switches [7, 64] to the linear hash table.

For heavy hitter detection in the hash table, we use the source and destination IP pair as the key and count the number of packets for each pair. A pair is a heavy hitter if its count is above a certain threshold. The implementation details of the hash table may affect the packet processing performance significantly [9]. To speed up the hash table, we applied several system optimizations such as cache prefetching, cache access alignment, and SIMD instructions to calculate the hash function.

Sketches: Sketches are summaries of streaming data to approximately answer a specific set of queries. For example, Count-Min sketch [22] is commonly used to find heavy hitters [21, 49, 50] [1]. A Count-Min sketch keeps a two-dimensional array of counters with d rows and w columns. It computes d hash functions per packet and updates the corresponding d positions in each row. To find the counter for a given IP pair, the minimum counter in the d locations is returned because it has minimum collisions. If the minimum counter is above the threshold, we add the IP pair to a set. Later at the report time, we report the set of IP pairs as heavy hitters. In contrast, a count array sketch computes one hash function per packet. When there are hash collisions, a count array simply adds up the counters for the collided keys.

Heaps and trees: Heaps reduces the memory usage by only keeping the most important entries for the measurement query (e.g., big flows). For example, the SpaceSaving algorithm [47] finds heavy hitters by tracking the volume of traffic from IP pairs in a small hash table. When the hash table gets full, it finds the entry with the minimum volume, say v_{min}, replaces that with the new IP pair, and adds the packet volume to the original counter (v_{min} plus the size of the new packet).

[1] The conclusions of this paper is easily extensible to other sketches.

Function	Meaning	Sketch	Heap/tree-based	Hash table
Heavy hitter	A traffic aggregate identified by a packet header field that exceeds a specified volume	NSDI'13[61] [22]	[47, 48], ANCS'11 [38]	SIGCOMM'02[29]
Super spreader	A source IP that communicates with a more than a threshold number of distinct destination IP/port pairs (Defined for destinations in a similar way.)	NSDI'13[61] [23]		IMC'10 [56], [59]
Flow size distribution	The distribution of sizes of flows distinguished by a set of packet header fields	[42]		IMC'10 [56]
Change detection	A drastic change of volume/# packets from a traffic aggregate compared to a prediction model	IMC'04 [55] [19]	[63]	IMC'10 [56]
Entropy estimation	Entropy (A measure of randomness/diversity) of volume/# packets from different flows	[45]		IMC'10 [56], SIGMETRICS'06 [43]
Quantiles	Dividing an ordered set of flows (e.g., based on source IP) into equal-weight subsets	[60]	SIGMOD'01 [31], SIGMOD'99 [46],[22]	

Table 1: A survey of proposed measurement solutions

To find the minimum entry, we need to keep a heap data structure [47]. Thus for each entry in the hash table, there is a corresponding entry in the heap, and for each packet, the heap must be updated to maintain its property.

Trees are also used to store a hierarchical set of counters [38, 49, 63]. For example, to detect heavy hitters, we can build an IP prefix tree and dynamically zoom in and out the subtrees based on the monitored traffic counters to reduce the number of monitored prefixes.

2.2 Previous works on measurement algorithms

Many previous works on measurement algorithms [19, 22, 27–29, 38, 43, 56, 59, 61, 63] promote the sketch-based solutions which maintain approximate counters with compact memory by leveraging multiple hash functions. This idea fits hardware switches which typically have limited high-speed memory. However, in software with a memory hierarchy, the total memory usage does not matter, but the number of memory accesses at different levels of the cache hierarchy affects the packet processing latency and throughput. As a result, it is not worthwhile to reduce the total memory usage at the expense of more instructions for calculating additional hash functions and more time to access extra entries. In fact, we will show in our evaluation that if we can reduce the number of hash functions and memory accesses, we can still achieve low latency and high throughput with a large total memory.

Unfortunately, even previous measurement works that target software environments [23, 31, 35, 48, 55, 59], only compare the different set of sketch and heap solutions and focus on the comparison of total memory usage. Some papers [21, 47] that compare hierarchical Count-Min sketch and heap-based solutions show that heap-based solutions can achieve better performance and accuracy. Other papers claim to achieve reasonable performance without rigorous testing on modern servers and comparison with single hash-based solutions.

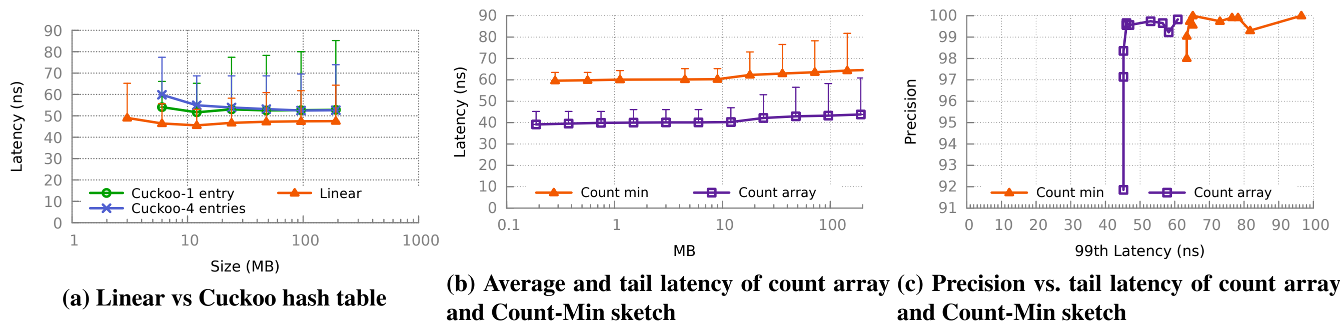
Instead, in this paper, we focus on a systematic comparison of both the performance and accuracy of hash tables, sketches, and heaps through extensive evaluations. We conclude that simple is often the best. For example, the simplest implementations of hash tables and sketches (i.e., the linear hash table and the count array) achieve the best performance and accuracy for heavy hitter detection. We also extend the evaluation to other measurement tasks and over different traffic traces.

3 EVALUATION OF MEASUREMENT ALGORITHMS IN SOFTWARE

Our key observation is that saving memory through extra computation is not worthwhile in achieving high performance and high accuracy for measurement in software. This is because packet batching and memory prefetching can mask the memory access latency. On the other hand, the latency due to extra computation cannot be masked as easily—superscalar processors and compilers already perform efficient interleaving of instructions and utilize the computation resources as much as possible.

We noticed two common approaches that use more computation to save memory: more hashes and complex data structures: (1) Computing multiple hashes to save memory degrades performance. For example, a count-array that uses a single hash function and large memory beats a Count-Min sketch that uses a smaller memory but makes up for accuracy loss by using multiple hashes. Also, the linear hash table has lower average and tail latency than the Cuckoo hash table that saves memory using multiple hashes. (2) It is possible to achieve the accuracy of more computationally intensive data structures by allocating more memory to simpler data structures while achieving better performance: we compare data structures based on sketch, hash table, and heap.

We start by evaluating measurement algorithms for heavy hitter detection in a single-core setting, and then we extend the result to other measurement tasks in Section 4 and multicore settings in Section 5.



(a) Linear vs Cuckoo hash table

(b) Average and tail latency of count array and Count-Min sketch

(c) Precision vs. tail latency of count array and Count-Min sketch

Figure 1: Comparing a single hash function with multiple ones

3.1 Evaluation settings

Testbed: We use a Xeon E5-2650 v3 processor with 10 cores, 256 KB of L2 cache per core, 25 MB of shared L3 cache, and a 10G network interface card. On this processor, the L1 access time is 1.6 ns, L2 access time is 5 ns, L3 is 15 ns, and main memory is 69 ns [5]. Typically, the access time of L1, L2, L3, and main memory follows similar trend across the latest CPU architectures [10].

Traffic traces: We use a one-minute trace from Equinix data center at Chicago from CAIDA [3] with 27 million packets and around 1 million unique flows. The CAIDA trace has a skew of Z=1.1 (which means that the most frequent entry has 10 times more packets than the 8th most frequent one [24, 41]). To generate traffic traces with different skews, we build a pool of source and destination IPs from the base CAIDA trace and sample from this pool using a Zipfian distribution.

In all experiments, we use the smallest TCP packet size, i.e., 64 bytes, to stress-test the measurement tasks under the highest possible per packet rate.

Measurement tasks: We focus on heavy hitter detection. We define heavy hitters as the source and destination IP pairs that have more than 0.1% of the total traffic in an epoch. We report in epochs of 2 million packets, which translates to a 130*ms* time window on a 10Gbps network interface card with 64-byte packets. We evaluate the generality of our observations for tasks that save more information per flow by evaluating heavy hitter detection with a variety of value sizes.

Measurement algorithm implementation: We evaluate three types of measurement algorithms: hash tables including linear hash tables and Cuckoo hash tables, sketches including count arrays and count-min sketches, and heaps (§2.1). By default, we keep the keys as source and destination IP pairs and the values as 12 byte counters. For each algorithm, we do not implement unnecessary features (e.g., for heavy hitter detection, we do not need to perform bookkeeping or have a decrement operator). This decision lets us save as many cycles as possible for each algorithm. We now describe our algorithm implementation in detail:

Hash tables: Our implementation of linear hash table holds one item per bucket and performs linear search on collisions. There are also other collision resolution techniques, e.g., Hopscotch [33] or Robin Hood hashing [18]. We opted not to use them, because as the size of the data structure increases the number of collisions decrease, which hides the impact of collision resolution strategy for packet processing. For Cuckoo hash table, we followed DPDK implementation [6] but removed the bookkeeping (required for deletion) to improve the performance.

Sketches: Count-array implementation is similar to the linear hash table, but instead the collision resolution strategy overwrites previous values. Our Count-Min sketch uses three count-arrays with pairwise independent hash functions.

Heaps and trees: We use a binary min heap as a representative tree like data structure for packet processing that is actively used across many algorithms, e.g., change detection [63], heavy hitter detection [47, 48]. We optimized the implementation by ensuring that we only heapify-down when updating values because the flow metrics, e.g., volume or packet count, can only increase.

We perform extensive system optimizations to make the measurement system as efficient as possible. For example, we use DPDK [6] to read packets from the NIC and send them as a batch to the application. Batching packets has several benefits: (a) it gives the compiler more freedom to optimize the code, e.g., through data-flow analysis [40], (b) it enables the instruction level parallelism across packets in the same batch; and (c) the compiler and the programmer can use prefetching and Single Instruction Multiple Data (SIMD) instructions to hide the latency of memory and CPU operations [32, 54].

Evaluation metrics: We consider two metrics: *(1) Performance:* We measure the average and tail latency (i.e., 99th percentile latency). We measure the latency from fetching packets from the NIC to sending the packets out of the measurement module and maintain the histogram. The average

latency dictates the packet processing throughput. The tail latency indicates the variance of packet processing time. A larger tail latency causes more packet drops because the NIC needs to maintain a longer queue. Note that this can happen even when the average latency per packet is low. *(2) Accuracy:* We measure the precision and recall for each measurement task. For example, to measure the precision of heavy hitter detection, we count the fraction of selected flows that are true heavy hitters; similarly, the recall is the fraction of true heavy hitters that are detected. The recall and precision of other tasks, e.g., superspreader or change detection, follow the same definition.

Evaluation settings: We run a warm up trace right before each experiment to ensure that the software switch code is cached. We perform zero-packet-loss performance benchmark [17]: for each experiment, we replay the trace at the highest throughput where packet loss is zero.

We process packets in batches of 64. To compute the average and tail latency, as it is too expensive to record the delay per packet, we measure the number of cycles to process each batch and add the corresponding per packet cycle into a histogram. The histogram has 2k buckets with each bucket representing 2 cycles.

3.2 A single hash function is better than multiple

We compare data structures with a single hash function to those with multiple hash functions (linear hash table vs. Cuckoo hashing and count array vs. Count-Min sketch). We observe that using a single hash function achieves better performance on average and in tail than using more hash functions without losing accuracy.

The linear hash table has lower average and tail latency than Cuckoo hash table. Figure 1a shows the average and the 99th percentile latency for the linear and Cuckoo hash tables. For the Cuckoo hash table, we first consider an implementation with one entry per bucket. For each hash bucket, we store one key-value pair (i.e., one entry per bucket is labeled as *Cuckoo-1 entry*). The Cuckoo hash table has between 30% (40%) to 10% (13%) higher average (tail) latency than the linear hash table over the whole range. This is because, with lookup misses, Cuckoo hashing always needs two hash functions to verify the miss whereas the linear hash table always requires one. This also means that Cuckoo hashing needs to make two random memory accesses, whereas linear hash table only needs to probe the current entry. The locality and predictability of reference in linear hash table and the size of the cache sizes (64 bytes) further help to ensure the availability of next key in cache. This makes linear hash table have an overall better performance even with larger data structures and when the load factor is low.

To increase the locality of reference, we may reduce the number of memory operations in the Cuckoo hashing by chaining, e.g., saving four entries per bucket (labeled as *Cuckoo-4 entries*)[30]. Thus, when collisions happen, we can save the entry in the same bucket with high probability without computing the second hash. Note that we chose four entries per bucket because the four entries fit in one cache line. Although Cuckoo-4 improves the tail latency of Cuckoo-1, it still has higher latency compared to linear hash table (Figure 1a). This is because with equal sized tables, there are fewer indices available in Cuckoo-4 than linear hash table, and thus, Cuckoo-4 can require multiple comparisons to find the key.

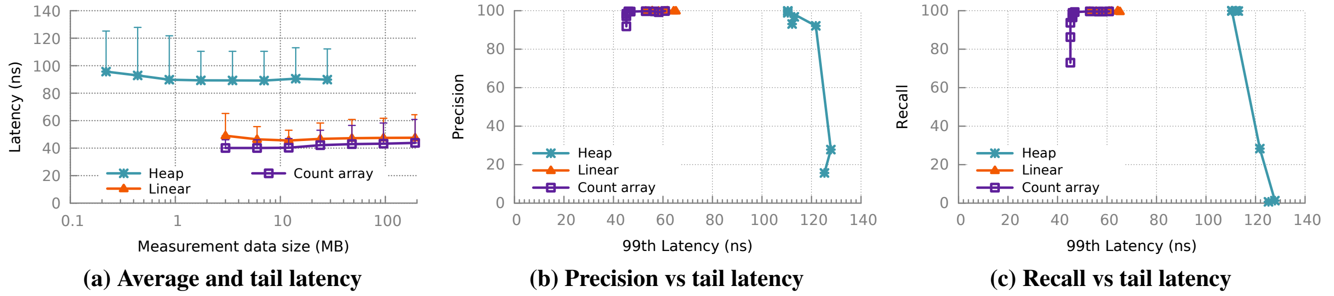
There is a large body of works on using Cuckoo hash tables for applications with high performance such as forwarding tables of switches [65] and for key-values stores [30]. Previous works choose Cuckoo hash tables because they focus on the load factor of the hash table, but in our context, we care less about the load factor since the number of records is much smaller than a table for a key-value store. In other words, Cuckoo hashing is not the fastest in our context because each lookup may require two hash computations and an insertion may require random shuffling of many entries in the hash table. Instead, we can use a large table—because the table size is only a fraction of the total memory size in modern software switches—and avoid computations that allow Cuckoo hash table to achieve a high load-factor.

The count array has lower average and tail latency than the Count-Min sketch. The count array with one hash function has lower average and tail latency with the same accuracy than Count-Min sketch, which uses three hashes, across all data structure sizes (Figure 1b). This is because the Count-Min sketch computes multiple hashes and needs multiple random memory accesses per packet, which defeats the purpose of smaller memory size for packet processing. The tradeoff between the performance (i.e., 99th percentile tail latency) and accuracy (i.e., precision[2]) is shown in Figure 1c. For example, the count array reaches 98% precision with 45 ns tail latency while the Count-Min sketch takes 64 ns for the same precision due to the additional hash function computations. Even when count array memory does not fit in the CPU cache, most of its memory accesses are still served by the cache because of the packet batching, memory prefetching, and traffic skews, which is common in networks [12, 13].

3.3 Use data structures with the simplest computation

We compare three classes of algorithms for heavy hitter detection: count arrays, linear hash tables, and heap-based algorithms. Among the three, count array has the least amount

[2]Recall also has the same trend.



(a) Average and tail latency

(b) Precision vs tail latency

(c) Recall vs tail latency

Figure 2: Performance and accuracy comparison of hashes, sketches, and heaps (traffic skew Z=1.1)

of computations, linear hash table is a bit more complex because of the collision resolution strategy, and the heap-based algorithm is the most computationally demanding but uses smaller memory. We show that using more computation to save memory does not improve the performance.

Count array has the lowest average latency compared to linear hash table and heap. We first compare the average latency of the three algorithms for heavy hitter detection for different sizes of data structures in Figure 2a. The count array has 142% better performance than the heap implementation and 28% better performance than the linear hash table. Figure 2a shows that as the size of the data structure grows, the latency difference between the linear hash table and count array vanishes because collisions rarely happen.

The heap has the worst performance among the three algorithms as it takes multiple memory accesses to navigate and maintain the heap data structure. For example, updating a heap entry with a subtree of height three may require updating all the tree layers.

Note that heap is still faster than more complex algorithms such as *hierarchical* sketches [21, 47]. Hierarchical sketches use multiple sketches to extract the heavy hitter flow information from their counters as counters in sketches do not keep the flow information (e.g., IP). However, updating multiple sketches in software requires many hash computations and memory accesses [3].

With larger data sizes, the tail latency increases significantly for the count array and linear hash table, but decreases for heap. The error bars in Figure 2a show the 99th percentile tail latency. The tail latencies of count array and linear hash table increase significantly when the measurement data size is above the L3 cache of the CPU (25 MB). If the measurement data is larger than the L3 cache, the memory access latency affects the *tail* latency of the packet processing pipeline.

[3]Our approach for using the count array is to simply keep heavy hitters in a set (i.e., add the flow to a set if it updates a counter above the threshold). Thus, we only need one sketch, and count array becomes a better choice than the heap for software.

It is worth noting that small linear hash tables have higher latency than the larger ones. This is due to the high load factor of small tables that incurs additional collision resolution cost. For example, in our experiments, a linear hash table with 3 MB performs 22% more memory accesses than a linear hash table with 200 MB.

However, the average and tail latencies of min-heap decrease with more memory. This is because with larger heaps, more heavy hitters end up in the leaves (versus nodes inside the heap), which makes heapify operation cheap because it only touches the leaves.

To achieve 100% accuracy, we should use the linear hash table; if accuracy loss is acceptable, count array has the best performance. Figure 2b compares the tradeoff between accuracy (i.e., precision/recall) and the performance (i.e., latency/tail latency) of different measurement algorithms. Even though heap works well with small memory space, it has the highest latency and the worst accuracy among the three algorithms and is never a good choice for measurement in software.

The linear hash table always achieves 100% precision and recall because it handles collisions. Its average latency is 46*ns* and its tail latency is 53*ns*. However, count array achieves 99.5% precision and 96.54% recall with 40*ns* average latency and 46*ns* tail latency. Saving 6*ns* in average latency improves the throughput by 9% for the smallest packet size where we only have 67ns to process each packet ($\frac{6_{ns}}{67_{ns}}$). The reduction in tail latency also lowers the chance of packet drops in the NIC queue as the maximum queue length drops. Therefore, the count array is the best choice if the consumer of measurement data can tolerate some accuracy loss. For example, for traffic engineering, handling a few small flows as heavy hitters ($<$ 100% precision) or missing a few heavy hitters ($<$ 100% recall) does not have much impact, especially, because the size of false detected heavy hitters and missed heavy hitters is close to the threshold [50].

4 GENERALITY TO DIVERSE MEASUREMENT TASKS

We discussed that for detecting heavy hitters, large and computationally lightweight data structures have better performance and comparable accuracy to small and complex data structures. Here, we generalize the result to a group of measurement tasks that keep per item state and update that state for every incoming packet. All the six measurements in Table 1 follow this model. For example, heavy hitter detection increments the per flow counters, superspreader detection updates a bloom filter per source IP. Such measurement tasks only rely on a data-structure that maps items to their state, i.e., a key-value store. We can implement a key-value store in software using (1) hash tables or (2) tree based algorithms.

Hash tables rely on hash functions and collision resolution strategies to find the location of an item; on the other hand, trees traverse a path from the root node and incur multiple memory accesses to find the location of the item. To compare the solutions, we need to compare the number of cycles used to find the location of a key.

Under no collisions, a hash table requires a single hash function to locate a key-value pair in the table. There are many well designed uniformly random hash function implementations [8], e.g., Metrohash, Cityhash, Murmur3, which typically take between 40~60 cycles for 16 bytes (>5 tuples) of data to execute. In comparison, L2 and L3 accesses take 10 and 40 cycles respectively. Thus, a hash table with no collisions takes between 50~100 cycles to locate the value of a key. On the other hand, a tree based solution requires multiple memory accesses (typically in the $\mathcal{O}(log(n))$ memory accesses and comparisons) to find the location of a key. Assuming the same memory access latency numbers for L2 and L3, a tree that is completely cached in L2 memory can only have between 63~2047 entries—ignoring any computational overhead and branch mispredictions—for a comparable performance to a hash table, which can be much larger. This means that a hash table with no collisions has a much better performance than tree based solutions.

The unique opportunity for network measurement tasks is that they can avoid collisions in hash tables using large tables. This is because the data of measurement algorithms is a fraction of the software memory hierarchy (e.g., 10s of MBs compared to 10s of GBs available on modern software switches). Thus, we can make the hash tables large enough that collisions become rare. Furthermore, we can mask the memory access latency through packet batching and prefetching. In contrast, the database and hardware switch community [30, **?**]], where most streaming algorithms come from, do not have the luxury of serving most queries from cache and thus have to rely on trading off computation and accuracy for memory size.

Finally, different measurement tasks have different strategies for updating the values associated with the keys. For example, when using count array for heavy hitter detection, values that map to the same bucket overwrite each other, whereas a linear hash table would resolve collisions through probing, and a heap would move the items around to preserve the heap property. Later in this Section, we discuss how the general result, use simple but large data structures, also apply to superspreader, which has complex memory access procedure for updates, and change detection, which is computationally complex.

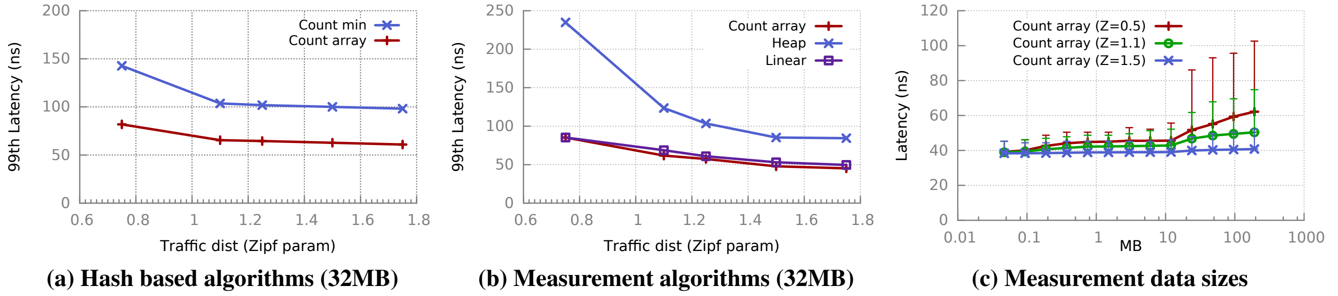
4.1 Impact of traffic skew, data structure size, and value size

The efficiency of memory hierarchy in software switches depends on the location of a state associated with a packet. This is because when the state is in upper layers of the memory hierarchy, the access latency becomes multiplicatively slower. For example, on our test server, the access latency of memory is 4.6 times slower than L3. There are two factors that dictate the location of a packet state in the memory hierarchy: (1) Traffic skew. With a skewed traffic, the packet processing pipeline serves a larger fraction of packets from the cache, leading to overall lower latency per packet. In contrast, a more uniform traffic distributes the state associated with a packet across all the layers, leading to higher latency per packet. (2) The data structure size. Whereas the data of a small data structure may fit in L1-L3 cache, a large data structure might still need to access memory to locate its data, leading to overall higher latency per packet. Here we discuss the impact of entry size, traffic skew, and data-structure size on the performance of packet processing pipelines.

Traffic skew. We study the impact of skew on measurement tasks by fixing the measurement task to heavy hitter detection and the data structure size to 32 MB. This size ensures that the measurement task does not fit in the L3 cache in our test server.

We first compare the impact of the skew across implementations with varying number of hash functions and memory accesses. Typically, as the traffic skew decreases, access patterns distribute more evenly across the memory hierarchy. Thus, measurement tasks with lower number of memory accesses per packet are less affected by the skew. Figure 3a shows the tail latency of count array and Count-Min sketch for heavy hitter detection. Since the Count-Min sketch makes 3 memory access per packet as opposed to only one for count array, the jump from skew 1.1 to 0.75 is larger for the Count-Min than the count array—even though the amount of computation per packet does not change with the skew.

Then, we compare the impact of skew across the heap, count array sketch, and linear hash table. Figure 3b shows



(a) Hash based algorithms (32MB) **(b) Measurement algorithms (32MB)** **(c) Measurement data sizes**

Figure 3: Effect of traffic skews on measurement algorithms and sizes

the 99th percentile per packet processing latency of these implementations across varying skews. Since the data structure is large (32 MB), the collisions are rare, and thus, the performance of linear hash table is only slightly worse than the count array sketch. Thus, under our settings, operators may prefer the linear hash table because it provides a guaranteed 100% accuracy with negligible impact on latency versus the count array.

However, the effect of skew on heap is more prominent: the skew not only affects the number of memory accesses but also the amount of computation that the heap performs. This is because under low skew the heap is more likely to move an item across multiple levels than when the skew is high. Figure 3b shows this where the heap latency grows by 45*ns* from skew 1.5 to 1.1, but by more than 100*ns* when going from skew 1.1 to 0.75.

Data structure size. To study the effect of skew changes together with the size of data structures, we fix the measurement task algorithm to count array and measure the packet latency when the traffic skew changes. Here, the skew dictates the working set of the data structure in L1, L2, and L3 cache of CPU. Traffic with higher skew is more likely to access the lower layer cache for packet data than the traffic with higher latency. Figure 3c shows the average and tail latency of the count array when we change its size from 48 KB to 200 MB over different traffic skews.

The two jumps at 200kB and 32MB indicate the size of the L2 and L3 cache. When the data structure is small enough to fit in L2 cache, no matter how the access pattern looks like, it will always get served from the L1 and L2 cache. As the data structure size increases, data gets distributed across other layers of the memory hierarchy. With less skewed traffic, we are more likely to access upper layer memories, and thus the latency gets affected more. This is visible in the figure by the separation of latency for different traffic skews once we pass the L3 cache size.

Entry size. A key factor for the difference in performance of measurement algorithms is the size of the stored state. The size of the entry dictates the percentage of the entries that are available in the lower layer of the memory hierarchy. Fewer number of larger entries fit in lower layer cache as opposed to smaller entries.

An entry contains both the key and the value. Typically, the key size depends on the flow granularity, e.g., whether we keep one IP address (4 bytes), source and destination IP addresses (8 bytes), or 5 tuples (13 bytes). For the value field, we keep a 4 byte counter together with the first few bytes of the latest packet to fill out the remaining space for that entry. We fix the key size to avoid incurring additional memory comparisons and hash function computation overhead and only keep the source and destination IPs (8 byte keys).

We implementat heavy hitter detection algorithms as discussed in the previous section. Measurement tasks may keep additional information, e.g., timestamp per flow (with a total value size of 12 bytes) or keep a list of destinations that the flow has contacted (e.g., 20 bytes on average). But that should not affect the generality of the impact of entry size on the performance.

Figure 6 shows that the tail latency of count array, linear hash table, and heap increases as the entry size grows. This is because we are more likely to access the upper layers of the memory hierarchy to locate our data. The jump for the heap here is linear and smaller than the jump shown in Figure 3b because here the number of memory accesses or the amount of computation of the heap does not change with varying entry size—we still use the packet counter to reorder the heap.

4.2 Impact of measurement tasks and storage of key-values

To cover the impact of memory and computational aspect of measurement task, we study two tasks: (1) superspreader detection, which updates a large memory portion per value, and (2) change detection, which is computationally more intensive than heavy hitter detection. We show that our results from the previous section still hold even on the two extremes of memory and computation complexity. Finally, we study the impact of value size on the performance, and suggest a

strategy to decide whether the key and values should be colocated in the hash table or not.

Superspreader detection. Superspreaders are the sources that chat with a large number of distinct destinations. They can identify distributed denial of service attacks (DDoS) or sudden changes in traffic pattern. We implement the superspreader module to report all the source IPs that send traffic to more than 128 different destinations in every epoch (2 mil packets). For every source IP, we keep a distinct Bloom filter counter per entry [15] with three hash functions and 1024 bits of data to identify new destinations. Due to the Bloom filter, superspreader has a more complex *update* procedure than heavy hitter detection.

Figure 4b shows the average and tail latency of different implementations of superspreader detection. Count array still has the lowest latency among the algorithms while reaching 97% precision (Figure 4b). The hash tables all have a precision of 99% and recall of 100% (The accuracy is less than 100% because of the Bloom filter error in distinct counting) with linear hash table being the fastest. The conclusion here follows the result for heavy hitter detection algorithms.

Change detection. Change detection identifies anomalies in packet streams, e.g., when the traffic pattern of a host suddenly changes or when the traffic volume changes too rapidly. Operators can use change detection for detecting compromised hosts, or as a signal to a control framework, e.g., load balancing, when sudden changes happen. For evaluation, we use an EWMA model to predict the traffic of each flow and report the flows that are outside the predicted value. Due to the prediction model, change detection is more computationally intensive than heavy hitter detection in updating per flow state.

Figure 5b shows the average and tail latency for different implementations of change detection. Count array still has the lowest latency among all the algorithms while reaching 99% precision (Figure 5b). The linear hash table has a precision of 100% and recall of 100%. Similarly, heap also has a precision and recall of 100% but with 80-100 ns higher latency.

Direct and indirect key-value storage. For measurement tasks with large values, it is better to store the values separately and only store a pointer in the hash table. We can then keep a contiguous list of keys to increase the locality of memory accesses for lookups when collisions happen. However, when the value size is small, it is more beneficial to keep the key and values together so that they share the cache line. To understand this tradeoffs, we implement two versions of linear hash tables: *Linear* which store keys and values together and *LinearPtr* which stores the keys with a pointer to the values.

Figure 7a shows the tail latency of both solutions with different traffic skews. For the lowest skew (Z = 0.75), the working set does not fit in cache and entries come in and go out of the cache. Each pointer is 8 bytes so keeping values smaller than 8 bytes only incurs additional delay. However as the value becomes larger, using a pointer becomes more beneficial. For example, for value size of 60 bytes, using a value pointer (LinearPtr) decreases the tail latency by 30%. This is because with large values lookups and insertions in a linear hash table are more likely to traverse multiple cache lines. Instead, value pointers promote key locality, which improve insertions and lookups by lowering cache lines that we go through.

For higher skew traffic (Z = 1.75 and Z = 1.25), the working set is small enough to fit in the CPU cache while the additional memory accesses due to the separation of keys and values has negligible overhead (about 5ns).

5 MEASUREMENT ALGORITHMS ON MULTIPLE CORES

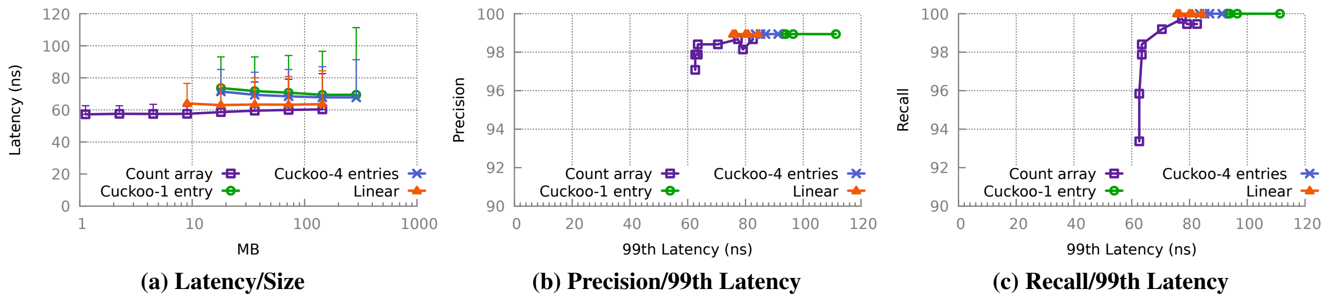
Measurement tasks never run in a standalone fashion. With a pipeline of network functions, it becomes harder for a single core to sustain the line rate packet processing. To get around this, we can load balance the incoming traffic across multiple cores based on a hash of the flows [2]; each core then runs the pipeline for a subset of flows [26]. Although, this leaves us with isolated measurement functions on each core and requires state synchronization across the cores. In this section, we will first investigate how to share the measurement data across cores running only measurement tasks. We will then study the impact of sharing resources with other applications.

5.1 Sharing states across multiple cores

When a measurement function runs over multiple cores, we need to synchronize states across cores. Maintaining locks on the shared state for consistency has a huge overhead, especially when the cache line that holds the lock is passed between the cores [16]. To get around this, we can either use (a) shared lockless data structures or (b) separated data structure for each core.

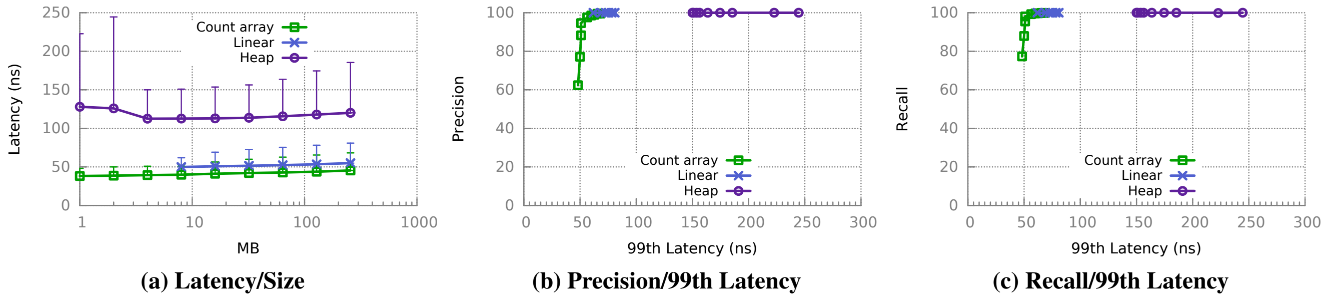
Shared lockless data structures. The linear hash table and the count array are easy to implement in a lockless fashion. For example, we can use compare-and-swap (or similar atomic operations) to update a counter atomically in a multithreaded environment. However, it is harder to implement lockless access for more complex data structures such as a heap.

Separated data structures. Each core maintains its own copy of the data structure. When we need to report the overall measurement results, we can merge the state/results from each data structure accordingly. Typically, merging the measurement results from multiple cores has little overhead if the reporting frequency is a few orders of magnitude greater



(a) Latency/Size **(b) Precision/99th Latency** **(c) Recall/99th Latency**

Figure 4: Performance and accuracy of superspreader detection



(a) Latency/Size **(b) Precision/99th Latency** **(c) Recall/99th Latency**

Figure 5: Performance and accuracy of change detection

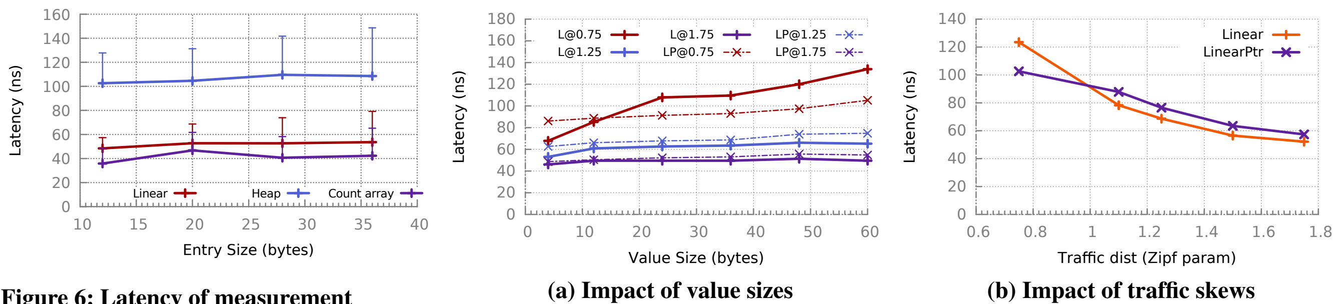


Figure 6: Latency of measurement algorithms for tasks with various entry sizes (traffic skew Z=1.1)

(a) Impact of value sizes **(b) Impact of traffic skews**

Figure 7: Comparing linear hash tables with and without pointers (key size=48 bytes)

than the packet processing time (e.g., >10ms reporting frequency vs. 67ns processing per packet). This is because a separate core can merge the data with low memory bandwidth usage. For example, with a measurement interval of 100ms and a 5MB data structure per core, a reporting core merging measurements of 10 cores only requires 500MB/s memory bandwidth, which is less than 1% of the total available memory bandwidth of our Xeon processor.

Separated option has lower latency than shared option. Figure 8 compares the latency of heavy hitter detection for the two options using a count array of different sizes. The average and tail latency of the separate approach are consistently lower than the shared one (The accuracy is not shown because it is the same). For example, when the size of count array is 32 KB, the tail latency of the shared count array is 12 ns higher than the separated count arrays. This is because of the overhead of running the compare and swap operations for maintaining the consistency of the shared data structure. Moreover, because the L3 cache is shared, the cache-coherency protocol will perform additional operations when a cache line in a core is read by a different core. On modern CPUs, this additional overhead can be as large as 40 cycles [44]. In contrast, the overhead of merging separate data structures is lower because we only need to pay the overhead at the reporting time rather than on a per-packet basis. Thus, saving memory also decreases the performance on multiple cores because it requires some sort of synchronization and wastes CPU cycle.

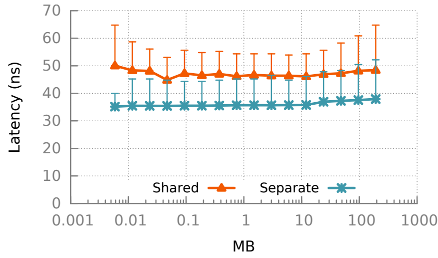
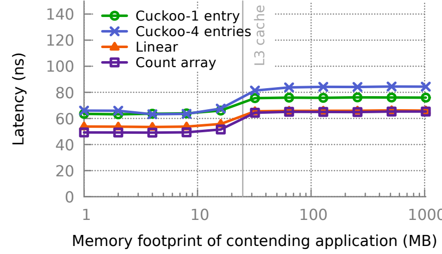
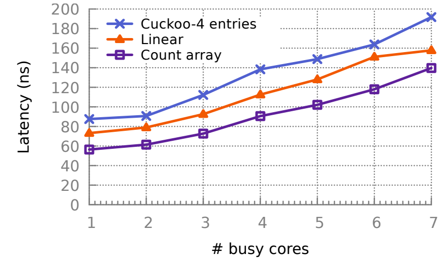


Figure 8: Latency of shared vs. separate count array across two cores



(a) Impact of L3 cache contention



(b) Impact of memory controller contention

Figure 9: Impact of resource sharing across applications

The shared count array latency is initially high and then decreases by 10% and remains almost constant until 20MB. For smaller count-arrays there is a higher chance that the two cores access the same entry (and cache line) in the count array, leading to extra latency due to compare-and-swap and the cache coherency protocol.

5.2 Sharing resources with other applications

The measurement tasks may run in conjunction with other network functions or applications on the same machine. Because all of these applications share resources, e.g., the cache and memory bandwidth, they end up affecting each other. For example, previous works have shown that cache-hungry applications can degrade the performance of other network functions [26]. To understand the impact of sharing resources on measurement algorithms, in addition to running the heavy hitter detection algorithm, we run two types of concurrent applications: (a) We run a single L3 aggressive application on a separate core that accesses random memory locations to show the impact of the contention on the L3 cache; (b) We run multiple of such applications on different cores that aggressively read and write memory to show the impact of the contention at the memory controller.

Impact of the L3 cache contention. We run a memory aggressive application on a core in the same NUMA domain as our measurement task. The application uses a hash function to access a random memory address and increment the value there. To guarantee that this application has higher priority for using the L3 cache than our measurement pipeline, we lowered the traffic rate so that the measurement task accesses the L3 at a much slower pace than the application core. We then measure the latency of the measurement task as the memory footprint of this application increases.

Figure 9a shows that the latency of the measurement task remains almost constant up to the L3 cache size. After, the memory aggressive application starves L3 cache and leaves no room for the measurement task, which cause the measurement task to access the main memory, leading to the sudden jump. Increasing the memory footprint of cache aggressive applications further does not increase the latency. This is because the next bottleneck is the memory bandwidth and our bandwidth usage is less than 10% of the available bandwidth of a NUMA domain (1.1GB/s out of 17GB/s) [1].

Impact of memory controller contention. Today, many big data analytics frameworks rely on the large memory available on modern servers to improve their performance. For example, Spark [62] keeps most of the intermediate data in memory for later usage; Hadoop [57] keeps portions of the files in memory for faster successive accesses. These applications can quickly drain the available memory bandwidth. Previous studies [37] show that Spark on average uses 40% of the memory bandwidth can can burst up to 90%. This high memory bandwidth usage affects the performance of the measurement tasks running on the same server. To study this, we wrote an application that aggressively utilizes the memory bandwidth. A single instance of this application utilizes 12GB/s of the 17GB/s of memory bandwidth[4]. We run many instances of this application to increase the contention of the memory bandwidth.

Figure 9b shows that as the number of applications increases, the latency of the measurement task increases. This is because with more requests to the memory controller, it becomes harder for the measurement task to fetch the packet data from memory, and therefore, with 7 cores the average latency of the measurement task increases by a factor of 2.9 for the count array.

Note that in both cache and memory bandwidth contention scenarios, the differences between the measurement algorithms still hold. The count array always has the lowest latency in all settings.

[4]We found out that even by running multiple instances of this application, we cannot utilize more than 14.5GB/s of the bandwidth, which we attribute to the queuing effect and the CPU parameters.

6 RELATED WORK

In addition to the related works covered in Section 2, our previous workshop paper [11] performed a preliminary evaluation of measurement algorithms. This paper extends the workshop paper in the following aspects: (1) Implementation: Our previous study was on the Click modular router [39], which is limited in throughput as it did not let us use advanced techniques such as batching and packet data prefetching from the cache. In this paper, we run all algorithms directly on DPDK and apply different techniques to reach the maximum packet rate. (2) Algorithms: Our previous study mainly focuses on count array, Count-Min sketch, and heap. In addition, this paper investigates more in hash table implementation. It compares the linear hash table and the Cuckoo hashing and shows that the linear hash table is the fastest choice when we need 100% accuracy. (3) Measurement tasks: In addition to heavy hitter detection in [11] which identifies keys with heavy volume counters, we also tested superspreader detection which counts the number of distinct items and change detection which identifies anomalies in traffic. (4) Settings: We also evaluate these algorithms on a variety of scenarios including multiple cores, different traffic skews, and a variety of entry sizes.

In addition to the three classes of algorithms introduced in Section 2, there are other packet and flow sampling solutions [20, 29, 34, 56]. These solutions are orthogonal to our algorithms and can always be combined to reduce the measurement load.

Recent works on optimizing the performance of network function in software switches [25, 26] mostly focus on better management of the memory usage of different network functions. Our work can help improve the performance of network functions by guiding developers to design and select the best measurement algorithms. Dobrescu et al. [26] associate the degradation of the network functions performance with the number of L3 references that competing applications make. We give insights on how to improve the performance of measurement components in such settings.

7 DISCUSSION

Theoretical model. While a theoretical model for estimating the latency of measurement pipeline helps in making design and optimization decisions, it is a challenging task as the performance of the packet processing pipeline depends on many factors, e.g., implementation of the algorithm (packet batching and/or prefetching, SIMD instructions), other resident applications, CPU properties (pipelining, speculative execution). Our previous work [11] shows a preliminary model for estimating the measurement algorithm latency. We incorporate the above factors into the model in the future.

8 CONCLUSION

With the trend of running network functions in software, keeping states inside these functions, and performing measurement to guide the deployment of these functions, it is important to understand which algorithms and data structures work the best in software. The key metrics in software are performance and accuracy rather than memory and accuracy in hardware. Our experiments and analysis show that simple is often the best. For measurement tasks that do not require perfect accuracy, a count array, which is general enough for a wide range of measurement tasks, has the lowest latency and the highest throughput. For tasks that require 100% accuracy, we recommend a linear hash table. We verified this conclusion for a variety of traffic settings, measurement tasks, and multiple core settings.

9 ACKNOWLEDGMENT

We thank the anonymous reviewers and our shepherd, Theophilus Benson, for their feedback on this paper. Further thanks to Mingyang Zhang, Sivaramakrishnan Satyamangalam Ramanathan, and the SOSR reviewers for their comments on earlier drafts of this paper. This research is partially supported by DHS-80515914, CNS-1701923, CNS-141397, and CNS-1701754.

REFERENCES

[1] Intel Performance Counter Monitor. https://software.intel.com/en-us/articles/intel-performance-counter-monitor
[2] Scaling in the Linux Networking Stack. https://www.kernel.org/doc/Documentation/networking/scaling.txt
[3] 2012. CAIDA Anonymized Internet Traces 2012. (2012). http://www.caida.org/data/passive/passive_2012_dataset.xml
[4] 2013. AT&T Domain 2.0 Vision White Paper. (2013). http://goo.gl/YqSjkA
[5] 2013. Intel Haswell. (2013). http://www.7-cpu.com/cpu/Haswell.html
[6] 2016. DPDK. http://dpdk.org. (2016).
[7] 2017. OVS hash map. (2017). https://github.com/openvswitch/ovs/blob/master/lib/cmap.h
[8] 2017. SMHasher. (2017). https://github.com/rurban/smhasher
[9] 2017. Writing a Damn Fast Hash Table With Tiny Memory Footprints. (2017). http://www.idryman.org/blog/2017/05/03/writing-a-damn-fast-hash-table-with-tiny-memory-footprints/
[10] 2018. 7-Zip LZMA Benchmark. (2018). http://www.7-cpu.com/
[11] Omid Alipourfard, Masoud Moshref, and Minlan Yu. 2015. Reevaluating Measurement Algorithms in Software. In *Proceedings of the 14th ACM Workshop on Hot Topics in Networks*.
[12] Mohammad Alizadeh, Albert Greenberg, David A Maltz, Jitendra Padhye, Parveen Patel, Balaji Prabhakar, Sudipta Sengupta, and Murari Sridharan. 2011. Data Center TCP (DCTCP). *SIGCOMM computer communication review* (2011).
[13] Theophilus Benson, Aditya Akella, and David A. Maltz. 2010. Network Traffic Characteristics of Data Centers in the Wild. In *IMC*.
[14] T. Benson, A. Anand, A. Akella, and M. Zhang. 2011. MicroTE: Fine Grained Traffic Engineering for Data Centers. In *CoNEXT*.
[15] Burton H. Bloom. 1970. Space/Time Trade-offs in Hash Coding with Allowable Errors. *Commun. ACM* (1970).

[16] Silas Boyd-Wickizer, M Frans Kaashoek, Robert Morris, and Nickolai Zeldovich. 2012. Non-scalable Locks Are Dangerous. In *Linux Symposium*.

[17] S. Bradner and J. McQuaid. 1999. Benchmarking Methodology for Network Interconnect Devices. https://tools.ietf.org/html/rfc2544.html, (1999).

[18] Pedro Celis, Per-Ake Larson, and J Ian Munro. 1985. Robin hood hashing. In *Foundations of Computer Science, 1985., 26th Annual Symposium on*. IEEE, 281–288.

[19] Moses Charikar, Kevin Chen, and Martin Farach-Colton. 2002. Finding Frequent Items in Data Streams. In *Automata, Languages and Programming*.

[20] Benoit Claise. 2004. Cisco systems NetFlow services export version 9. http://tools.ietf.org/html/rfc3954.html, (2004).

[21] Graham Cormode and Marios Hadjieleftheriou. 2008. Finding Frequent Items in Data Streams. *VLDB Endowment* (2008).

[22] Graham Cormode and S. Muthukrishnan. 2005. An Improved Data Stream Summary: The Count-Min Sketch and its Applications. *Journal of Algorithms* (2005).

[23] G Cormode and S Muthukrishnan. 2005. Space Efficient Mining of Multigraph Streams. In *PODS*.

[24] Graham Cormode and S Muthukrishnan. 2005. Summarizing and Mining Skewed Data Streams. In *SIAM International Conference on Data Mining*.

[25] Mihai Dobrescu, Katerina Argyraki, Gianluca Iannaccone, Maziar Manesh, and Sylvia Ratnasamy. 2010. Controlling Parallelism in a Multicore Software Router. In *PRESTO*.

[26] Mihai Dobrescu, Katerina Argyraki, and Sylvia Ratnasamy. 2012. Toward Predictable Performance in Software Packet-Processing Platforms. In *NSDI*.

[27] Nick Duffield, Carsten Lund, and Mikkel Thorup. 2003. Estimating Flow Distributions from Sampled Flow Statistics. In *SIGCOMM*.

[28] Daniel Egloff and Markus Leippold. 2010. Quantile Estimation with Adaptive Imprtance Sampling. *The Annals of Statistics* (2010).

[29] Cristian Estan and George Varghese. 2002. New Directions in Traffic Measurement and Accounting. In *SIGCOMM*.

[30] Bin Fan, David G. Andersen, and Michael Kaminsky. 2013. MemC3: Compact and Concurrent MemCache with Dumber Caching and Smarter Hashing. In *Proceedings of the 10th USENIX Conference on Networked Systems Design and Implementation*.

[31] Michael Greenwald and Sanjeev Khanna. 2001. Space-efficient Online Computation of Quantile Summaries. In *SIGMOD*.

[32] Sangjin Han, Keon Jang, KyoungSoo Park, and Sue Moon. 2010. PacketShader: A GPU-accelerated Software Router. In *Proceedings of the ACM SIGCOMM 2010 Conference*.

[33] Maurice Herlihy, Nir Shavit, and Moran Tzafrir. 2008. Hopscotch Hashing. In *Proceedings of the 22Nd International Symposium on Distributed Computing*.

[34] Nicolas Hohn and Darryl Veitch. 2003. Inverting Sampled Traffic. In *IMC*.

[35] Qun Huang, Xin Jin, Patrick P. C. Lee, Runhui Li, Lu Tang, Yi-Chao Chen, and Gong Zhang. 2017. SketchVisor: Robust Network Measurement for Software Packet Processing. In *SIGCOMM*.

[36] Jinho Hwang, K. K. Ramakrishnan, and Timothy Wood. 2014. NetVM: High Performance and Flexible Networking Using Virtualization on Commodity Platforms. In *NSDI*.

[37] Tao Jiang, Qianlong Zhang, Rui Hou, Lin Chai, Sally A Mckee, Zhen Jia, and Ninghui Sun. 2014. Understanding the behavior of in-memory computing workloads. In *Workload Characterization (IISWC), 2014 IEEE International Symposium on*.

[38] Faisal Khan, Nicholas Hosein, Chen-Nee Chuah, and Soheil Ghiasi. 2011. Streaming Solutions for Fine-Grained Network Traffic Measurements and Analysis. In *ANCS*.

[39] Eddie Kohler, Robert Morris, Benjie Chen, John Jannotti, and M. Frans Kaashoek. 2000. The Click Modular Router. *Transaction on Computer Systems* (2000).

[40] Martin Kong, Richard Veras, Kevin Stock, Franz Franchetti, Louis-Noël Pouchet, and P. Sadayappan. 2013. When Polyhedral Transformations Meet SIMD Code Generation *(PLDI)*.

[41] Flip Korn, S Muthukrishnan, and Yihua Wu. 2006. Modeling Skew in Data Streams. In *SIGMOD*.

[42] Abhishek Kumar, Minho Sung, Jun (Jim) Xu, and Jia Wang. 2004. Data Streaming Algorithms for Efficient and Accurate Estimation of Flow Size Distribution. In *SIGMETRICS*.

[43] Ashwin Lall, Vyas Sekar, Mitsunori Ogihara, Jun Xu, and Hui Zhang. 2006. Data Streaming Algorithms for Estimating Entropy of Network Traffic. In *SIGMETRICS/Performance*.

[44] David Levinthal. 2009. Performance Analysis Guide for Intel Core i7 Processor and Intel Xeon 5500 processors. (2009). https://software.intel.com/sites/products/collateral/hpc/vtune/performance_analysis_guide.pdf

[45] Ping Li and Cun-Hui Zhang. 2011. A New Algorithm for Compressed Counting with Applications in Shannon Entropy Estimation in Dynamic Data. In *COLT*.

[46] Gurmeet Singh Manku, Sridhar Rajagopalan, and Bruce G. Lindsay. 1999. Random Sampling Techniques for Space Efficient Online Computation of Order Statistics of Large Datasets. In *SIGMOD*.

[47] Ahmed Metwally, Divyakant Agrawal, and Amr El Abbadi. 2005. Efficient Computation of Frequent and Top-k Elements in Data Streams. In *ICDT*.

[48] Michael Mitzenmacher, Thomas Steinke, and Justin Thaler. 2011. Hierarchical heavy hitters with the space saving algorithm. *arXiv preprint arXiv:1102.5540* (2011).

[49] Masoud Moshref, Minlan Yu, and Ramesh Govindan. 2013. Resource/Accuracy Tradeoffs in Software-Defined Measurement. In *HotSDN*.

[50] Masoud Moshref, Minlan Yu, Ramesh Govindan, and Amin Vahdat. 2015. SCREAM: Sketch Resource Allocation for Software-defined Measurement. In *CoNEXT*.

[51] Rasmus Pagh and Flemming Friche Rodler. 2004. Cuckoo Hashing. *J. Algorithms* (2004).

[52] Parveen Patel, Deepak Bansal, Lihua Yuan, Ashwin Murthy, Albert Greenberg, David A. Maltz, Randy Kern, Hemant Kumar, Marios Zikos, Hongyu Wu, Changhoon Kim, and Naveen Karri. 2013. Ananta: Cloud Scale Load Balancing. In *SIGCOMM*.

[53] Ben Pfaff, Justin Pettit, Teemu Koponen, Ethan J Jackson, Andy Zhou, Jarno Rajahalme, Jesse Gross, Alex Wang, Joe Stringer, Pravin Shelar, and others. 2015. The Design and Implementation of Open vSwitch. In *NSDI*.

[54] Luigi Rizzo. 2012. netmap: A Novel Framework for Fast Packet I/O. In *Presented as part of the 2012 USENIX Annual Technical Conference (USENIX ATC 12)*.

[55] Robert Schweller, Zhichun Li, Yan Chen, Yan Gao, Ashish Gupta, Yin Zhang, Peter A. Dinda, Ming-Yang Kao, and Gokhan Memik. 2007. Reversible Sketches: Enabling Monitoring and Analysis over High-speed Data Streams. *Transaction on Networking* (2007).

[56] Vyas Sekar, Michael K. Reiter, and Hui Zhang. 2010. Revisiting the Case for a Minimalist Approach for Network Flow Monitoring. In *IMC*.

[57] Konstantin Shvachko, Hairong Kuang, Sanjay Radia, and Robert Chansler. 2010. The Hadoop Distributed File System. In *Proceedings of the 2010 IEEE 26th Symposium on Mass Storage Systems and*

Technologies (MSST).

[58] Amin Vahdat. 2014. Enter the Andromeda zone - Google Cloud Platform's Latest Networking Stack. (2014). http://goo.gl/smN6W0

[59] Shobha Venkataraman, Dawn Song, Phillip B. Gibbons, and Avrim Blum. 2005. New Streaming Algorithms for Fast Detection of Superspreaders. In *NDSS*.

[60] Richard Wang, Dana Butnariu, and Jennifer Rexford. 2011. OpenFlow-based Server Load Balancing Gone Wild. In *Hot-ICE*.

[61] Minlan Yu, Lavanya Jose, and Rui Miao. 2013. Software Defined Traffic Measurement with OpenSketch. In *NSDI*.

[62] Matei Zaharia, Mosharaf Chowdhury, Michael J. Franklin, Scott Shenker, and Ion Stoica. 2010. Spark: Cluster Computing with Working Sets. In *Proceedings of the 2Nd USENIX Conference on Hot Topics in Cloud Computing*.

[63] Ying Zhang. 2013. An Adaptive Flow Counting Method for Anomaly Detection in SDN. In *CoNEXT*.

[64] Dong Zhou, Bin Fan, Hyeontaek Lim, David G. Andersen, and Michael Kaminsky. 2013. Scalable, High Performance Ethernet Forwarding with CuckooSwitch. In *CoNEXT*.

[65] Dong Zhou, Bin Fan, Hyeontaek Lim, Michael Kaminsky, and David G. Andersen. 2013. Scalable, High Performance Ethernet Forwarding with CuckooSwitch. In *Proceedings of the Ninth ACM Conference on Emerging Networking Experiments and Technologies*.