Re-evaluating Measurement Algorithms in Software

Omid Alipourfard, Masoud Moshref, Minlan Yu
University of Southern California
{alipourf, moshrefj, minlanyu}@usc.edu

ABSTRACT

With the advancement of multicore servers, there is a new trend of moving network functions to software servers. Measurement is critical to most network functions as it not only helps the operators understand the network usage and detect anomalies, but also produces feedback to the control loop in management tasks such as load balancing and traffic engineering. Traditional researches on measurement algorithms mainly focus on reducing the memory usage leveraging the fact that measurement can sustain bounded inaccuracy. In this study, we re-evaluate these algorithms on software servers in order to understand their tradeoffs of accuracy and performance. We observe that simple hash tables work better than more advanced measurement algorithms for a variety of measurement scenarios. This is because with better cache design in modern servers and the skewness in the access patterns of measurement tasks, the memory usage of measurement tasks is largely irrelevant to the packet processing performance.

Categories and Subject Descriptors

C.4 [**Computer System Organization**]: Performance of Systems; C.2.3 [**Network Operations**]: Network Monitoring

General Terms

Measurement, Performance

1. INTRODUCTION

With growing concerns of the cost and management difficulty of hardware network devices (switches, middleboxes), there is a new trend of moving some network functions to software. For example, data centers today often run load balancing and firewalls in software [43, 40]; AT&T starts to deploy virtualized network functions (VNFs) to replace their hardware boxes [5].

Measurement is a key component in most network functions. For anomaly detection, accounting, and network profiling, we need to add measurement modules (e.g., heavy hitter and super-spreader detection) in the NFV chain to measure the traffic characteristics of individual applications and tenants. Other network functions (e.g., deep packet inspection) collect different statistics of the traffic by looking into each packet. Even those control-related network functions such as load balancing and traffic engineering also rely on accurate counting of flows before making control decisions. Finally, to understand the performance of individual network functions and identify the bottlenecks in NFV chains, we need to maintain traffic counters for each NFV function.

Recent works on improving the performance of NFVs in software [17, 16] often pick measurement tasks as key examples (e.g., NetFlow, IDS) and show that memory management is critical for packet processing performance and propose new state management designs in NFVs. If we improve the performance of measurement tasks, we can significantly improve most NFV pipelines.

Measurement is often known as a memory intensive task for keeping a large number of counters for many flows. As a result, there have been many algorithms on reducing memory usage of measurement tasks while maintaining the measurement accuracy. For example, to count heavy hitters, there have been sketch-based, heap-based, and sampling-based solutions [36, 12, 20]. While many works compared memory-accuracy tradeoffs across these approaches [38], there are a few works that show the difference of these solutions in performance (i.e., delay and throughput).

In this paper, we re-evaluate the accuracy and performance of these measurement algorithms in software and observe that simple hash tables work better than more advanced solutions. This is because: (a) Modern multicore servers have significantly increased their cache size and cache efficiency; (b) A large variety of measurement tasks do not require a large working set of items in cache especially with data access skew. As a result, the memory usage of measurement tasks is mostly irrelevant to the packet processing performance in software. Trading off memory for CPU and accessing many memory entries to compress the data structure are harmful to packet processing performance. Our preliminary evaluation shows that simple hash tables work better than more advanced measurement algorithms. For example, for heavy hitter detection, the throughput of a simple hash table is 28% and 120% better than sketches and heap based alternatives, respectively while achieving comparable accuracy. To handle those cases where the packet processing pipeline becomes bottleneck even with the hash tables, we propose ways to dynamically tune the pipeline for mea-

HotNets '15 November 16–17 2015, Philadelphia, PA USA
© 2015 ACM 978-1-4503-4047-2/15/11$15.00.
DOI: http://dx.doi.org/10.1145/2834050.2834064

surement functions or redesign measurement solutions to fit better with the software architecture.

2. BACKGROUND

We investigate previous works on a variety of measurement tasks such as heavy hitter detection, detecting traffic changes and computing flow size distribution (Table 1). These measurement tasks are widely used for many networking functions such as traffic engineering, security, and anomaly detection. We find that these tasks are often implemented with three types of algorithms. Here, we describe each type and briefly discuss how they implement the heavy hitter detection task. We define a heavy hitter as a source-destination IP address pair that sends traffic volume more than a pre-specified threshold. As an example, such measurements can be used for collocating chatty VMs in data centers to save network bandwidth.

Sketch: Sketches are compact data structures used in streaming algorithms to store summary information about the state of packets. For example, a Count-Min sketch [12] keeps a two dimensional array of integer 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 address, the minimum counter in the d locations is returned. If the minimum counter is above the threshold, we add the IP address to a set. Later at the report time, we can report the IPs in the set as heavy hitters.

Heap based solutions: Instead of hashing, many measurement solutions leverage data structures like heap to keep a small summary of traffic and reduce the total memory usage. For example, SpaceSaving algorithm [36] finds heavy hitters as follows: It tracks the volume of traffic from IP pairs in a small hash table, but whenever the hash table becomes full, it finds the entry with minimum volume, say v_{min}, and replaces it with the new entry with the sum of the original counter, v_{min}, and the size of new packet. To find the minimum entry, we need to keep a heap data structure [36]. 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.

In addition to heaps, there are algorithms that leverage counters on the IP prefix tree to detect heavy hitters and big changes [27, 48, 38]. These algorithms dynamically zoom-in and zoom-out in the tree based on the monitored traffic counters to reduce the number of monitored prefixes.

Sampling: To reduce the packet processing and memory overhead, we can use a variety of sampling solutions [10, 20, 24]. In the simplest form, we can use packet sampling to find heavy hitters. Then we keep a hash table for IP pairs (similar to flexible NetFlow) and update counters for sampled packets, estimate the actual traffic from the counters, and identify the heavy hitters. As packet sampling may not have good accuracy for some measurement tasks [20, 18, 30, 24], other sampling methods such as flow sampling [24], and sample and hold [20] are introduced in the literature, too.

What is the best measurement algorithm for software? Most previous works on measurement algorithms [20, 42, 47, 27, 31, 18, 48] focus on reducing the memory usage of measurement tasks while bounding measurement errors. This is mainly because many works focus on hardware switches with limited memory. It is unclear whether these works also apply for software with a memory hierarchy. Other works [9, 19, 12, 44] are theoretical with no detailed argument on why the total memory usage is constrained in practice. However, in software, it is not the total memory that matters, but the *working set* (i.e., the amount of data that commonly accessed and stored in cache) that dominates the packet processing delay and throughput, and because of the skew in network traffic [7, 6], the working set is usually much smaller than the total memory usage. Thus, reducing the memory usage by using more CPU and accessing many memory entries (i.e., larger working set) lowers packet processing performance.

Some previous works have evaluated measurement algorithms in software [13, 41, 23, 44, 37]. They mostly claim the delay is good in their own solutions (e.g., sketch [41, 13], heap [23, 37]) without comparing with other solutions that are not limited by memory. The only papers [11, 36] that compare hierarchical Count-Min sketch and heap-based solutions show that heap-based solutions can achieve better performance and accuracy.

Instead, in this paper, we focus on a systematic comparison of both the performance and accuracy of different types of measurement algorithms and conclude that simple hash tables work best for many scenarios in software. The simple hash table maintains per packet or per flow entries where each entry can keep simple traffic counters or more complex traffic information such as packet headers and timestamps. Knowing the exact size of traffic from each flow in the hash table, we can easily implement the tasks in Table 1.

3. RE-EVALUATING MEASUREMENT ALGORITHMS IN SOFTWARE

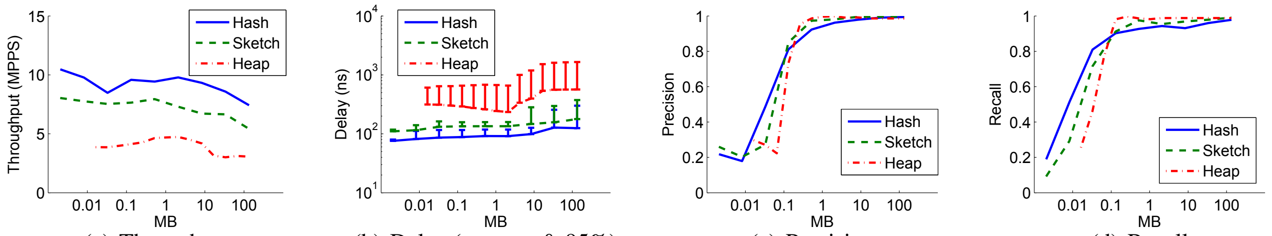
In this section, we re-evaluate the measurement solutions on software servers and identify the right data structures that optimize the performance and accuracy. We first take heavy hitter detection as an example, compare three types of measurement algorithms, and conclude that simple hash tables work best in software. We then build a model to understand the performance of simple hash tables and show that simple hash tables work for a variety of measurement tasks in modern servers. Finally, we discuss potential problems of using simple hash tables and possible solutions.

3.1 Simple hash tables work better

We evaluated the heavy hitter detection on a Xeon E5-2650 processor with 10 cores, 256 KB L2 cache per core, 25 MB shared L3 cache, and 32 GB memory. We send traffic

Function	Meaning	Sketch	Heap/tree based	Sampling
Heavy hitter	A traffic aggregate identified by a packet header field that exceeds a specified volume	NSDI’13[47] [12]	[36, 37], ANCS’11 [27]	SIGCOMM’02[20]
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[47] [13]		[44]
Flow size distribution	The distribution of sizes of flows distinguished by a set of packet header fields	[30]		SIGCOMM’03 [18]
Change detection	A drastic change of volume/# packets from a traffic aggregate compared to a prediction model	IMC’04 [41] [9]	[48]	IMC’10 [42]
Entropy estimation	Entropy (A measure of randomness/diversity) of volume/# packets from different flows	[33]		SIGMETRICS’06 [31], IMC’10 [42]
Quantiles	Dividing an ordered set of flows (e.g., based on source IP) into equal-weight subsets	[46]	SIGMOD’01 [23], SIGMOD’99 [34],[12]	[19]

Table 1: A survey of proposed measurement solutions



(a) Throughput (b) Delay (average & 95%) (c) Precision (d) Recall

Figure 1: Performance comparison of algorithms with different sizes of data structure (skew parameter $Z = 1$)

with a 10 G network interface card by using a modified version of Click modular router [28] with DPDK 1.8.0 [4]. We used a one-minute trace from CAIDA [3] containing 40 million packets. The CAIDA traffic has a skew $Z = 1$ (which means that most frequent entry has 10 times more packets than the 10^{th} most frequent one [29, 14].) We define heavy hitters as 0.2% of the traffic in a 1M packet epoch [1].

We implemented three algorithms: a simple hash table, a Count-Min sketch [12], and a heap-based solution [36]. The simple hash table maintains one counter at each entry, and increments the counter based on the hash value of each packet. The counters may be inaccurate when multiple flows are hashed in the same bin. The Count-Min sketch is similar to the simple hash table but uses multiple hash functions for each packet. Count-Min sketch and heap-based solution are based on the description in Section 2.

We first compare the throughput and delay of the three algorithms with different sizes of their data structures. Figure 1(a) shows that the throughput for the simple hash function is on average 28% higher than the Count-Min sketch and 120% higher than the heap-based approach. This is consistent to the average latency comparison in Figure 1(b). Sketch has worse performance than Hash because it computes 3 hash functions per packet and accesses 3 random memory locations. Similarly, Heap has the worst performance than the other two because it takes multiple memory accesses to navigate and maintain the heap data structure. Note that our result is different with previous work [11, 36], which shows heap is better than hierarchical sketches. This is because we use a simpler one-layer sketch together with a short list, which requires fewer hash computation and fewer memory accesses than hierarchical sketches, and thus a better choice for software.

With Hash and Sketch, there is a large interval where increasing the size does not affect the average latency (up until 20 MB) (Figure 1(b)). This interval happens because L2 and L3 memory cache the flow counters very effectively and most of the popular counters are served by these two caches. In contrast, Heap has decreasing latency with larger sizes. This is because with larger sizes, the heavy hitters all fall in the leaves in the min-heap. Therefore, we do not need to re-arrange the heap much as packets pass by.

Figure 1(b) also shows the 95 percentile latency for the three approaches. Tail latency is critical for the stability of packet processing performance. This is because with large latency variance, the NIC may maintain longer queues at times leading to higher chances of packet losses. The tail latency in Figure 1(b) has a jump between the 10 MB to 30 MB, which overlaps with the size of L3. Heap has increasing tail latency with larger sizes. In fact, the tail latency grows proportionally with the depth of the heap. This is because at the tail the packets often require reorganizing the heap, which takes longer time with larger heaps.

Finally, we compare the accuracy across these measurement algorithms. We consider two metrics: precision (the fraction of detected true heavy hitters over all detected ones) and recall (the fraction of true heavy hitters that are retrieved). When the size is above 200 KB, Heap achieves higher accuracy than the simple hash table but only by 4%. Most applications are willing to sacrifice the accuracy for better performance. Only for those applications (e.g., security related) that need very high accuracy, they may use larger hash table. For example, in our experiments, we can use hash tables as large as 5MB without affecting through-

[1]Our evaluation with different threshold numbers resulted in the same conclusion.

put, as visible in Figure 1(a).

3.2 Generality to more measurement tasks

To understand the performance of the simple hash table, we provide a simple model for the relation between the hash table size and the delay/throughput performance for a variety of measurement tasks. We consider a simple hash table with X entries and Y bytes per entry.

When a packet enters a software server, it first is buffered in one of the queues of the NIC. Many software-based packet-processing systems [35, 25] use libraries such as DPDK [4] to poll the packets from the NIC, bypass the kernel, and process the packets in the user space in order to reduce the packet processing latency. Thus, the throughput of software servers depends on how fast we can processes the packets in the user space.

Therefore, to ensure sustainable throughput, the average packet processing time should be less than the packet inter-arrival time. For example, for a 10 Gbps NIC that can receive packets at 14.8 Mpps (with 64 Byte packets), the average packet processing time, T_{packet}, should be less than $1/14.8M = 67ns$.

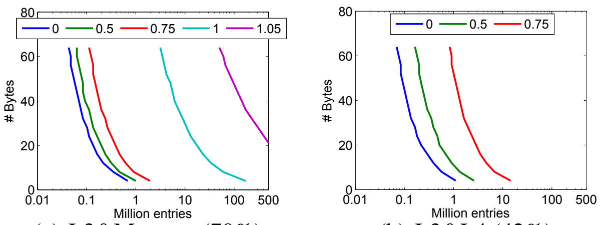
We model the packet processing time by considering the CPU processing time and the data access time separately. The CPU processing time, $T_{process}$, is the time spent on accessing packet header information, accessing the data in L2 cache, and hashing (e.g., for sketches) or other algorithmic calculations (e.g., navigating the heaps or prefix tree). (We consider L1 and L2 cache accesses inside $T_{process}$ because they have smaller impact on the latency than L3 and memory). For simple data-structures like count-array and Count-Min sketch, $T_{process}$ is proportional to the number of hash functions that are calculated over a single packet. The measurement data can be in either L3 cache or memory. To calculate L3 access time, T_{L3}, we assume the measurement data is accessed by one core and postpone the multicore discussion to Section 3.4. If the measurement data is not in cache, it needs to be fetched from memory with delay T_{mem}. We define p as the hit probability of L3 cache, and model the per-packet delay T_{packet} as follows:

$$T_{packet} = T_{process} + p \times T_{L3} + (1-p) \times T_{mem} \qquad (1)$$

In order to keep the average packet delay, T_{packet}, below a bound (and thus to ensure line speed throughput), we should ensure the portion of L3 hits, p, to be above a threshold. For example, if $B = 67ns$ and $T_{process} = 25ns$[2], $T_{L3} = 16ns$, $T_{mem} = 100ns$ [1], p should be above 70%.

The hit probability of L3 cache for each packet, p, depends on the size of data (the number of entries in the hash table (X) and the number of bytes per entry (Y)) versus the L3 cache size. More importantly, p also depends on the traffic skew (Z) because processors often keep popular entries in cache. If we sort entries in the descending order based on

[2]We get $T_{process}$ from running a hash-based heavy hitter detection in Xeon processor



(a) L3&Memory (70%) (b) L3&L4 (42%)

Figure 2: #bytes and entries that can be handled if different percentiles of accesses are handled by 2MB L3 Cache (Legend numbers represent skew parameter, 0 is uniform).

Example	# bytes
Simple hash table	4
5 tuple + 1 counter	17
Simple hash table + distinct counter	20
Source IP + list of destination IPs (4 in avg)	20
NetFlow v5	44

Table 2: Example number of bytes

Example	10Gbps	40Gbps	100Gbps
Packets	14.8M	60M	148M
5-tuple flows	1.25M	5M	125M

Table 3: Example number of entries in 1 second (avg flow size 1kB)

the number of packets hitting them, to ensure p is above a threshold, we need up to p^{th} percentile of entries to fit in the L3 cache [3].

How effective is a simple hash table? Given the packet processing model, we show how a simple hash table works for most measurement tasks, and we do not need more complex measurement algorithms discussed in Section 2.

Figure 2(a) shows the number of entries and the number of bytes per entry to support line rate packet processing of a 10 Gbps NIC on our Xeon processor. This means that we need 70^{th} percentile of entries to fit into a 2 MB cache (the average L3 cache size a core can use) with different skew parameters. With a skewed traffic with a conservative parameter ($Z = 1$), we can easily fit 5.5 million entries with 44 bytes (NetFlow) or 20 million entries with 17 bytes (5 tuples + one counter).

Table 2 and Table 3 show the number of entries and bytes per entry for typical measurement tasks. A measurement task keeps either per flow information or per packet information. Table 3 shows the number of packets and 5-tuple flows we need to keep for 10 Gbps and 40 Gbps links if the measurement interval is 1 second. Counting the number of requests for a key-value store is an example of per packet measurement task, and finding heavy hitter flows for traffic engineering is an example of flow-based tasks. Note that there can be different variations of such measurement tasks but we expect them to have comparable number of entries. For example, the median cluster size in Google is 10k servers with 9 tasks (similar to VM) or less for 50% of them [45]. Even if we assume, tasks in a rack contact every

[3]Note that this is a simplified model where entries do not contend for cache lines. This is a reasonable assumption because the entries are randomly accessed based on the hash function.

other task, the total number of entries for counting traffic for source-destination IP addresses pairs for a rack of size 48 will be $48 * 9 * 10000 * 9 = 39M$.

We may need different number of bytes per entry for different measurement tasks (Table 2). Simply keeping a counter (4 bytes) is already useful as a sketch for finding heavy hitters similar to Count-Min sketch and estimating flow size distribution [30] and entropy. To find quantiles, we can keep tuples information and a counter for volume. In order to find super-spreaders, we can keep a distinct counter [21] in the simple hash table or explicitly keep the list of destination IPs for each source IP [44]. Finally, in the extreme case, NetFlow v5 keeps many counters per flow including volume, # packets, TCP flags, and the arrival time of the first and last packets of a flow.

Cache growth in the future: Recently, L4 with low latency and large capacity (128MB) is introduced for Intel Haswell and IBM Power7. This can be a trend in the future as it uses eDRAM technology that avoids power and area usage limitations of SRAM in L3 [15]. Since L4 cache is large enough to store most measurement data structures, we replace the L_{mem} with L_{L4} in Equation 1, where $L_{L4} = 60ns$ [2]. Therefore, p in Equation 1 reduces to 42%. This allows more than 500M entries even with skew parameter $Z = 1$ and 64 bytes per entry. Comparing Figure 2(a) and Figure 2(b) shows that for skew $Z = 0.75$, the number of entries for 44 bytes per entry increases from 0.16 million to 1.2 million. Also note that the graphs in Figure 2 are for 2 MB cache (the average L3 cache size per core), but L3 cache is usually larger and shared among cores (e.g., 20 MB shared among 10 cores). Measurement has one of the largest memory/L3 references among NFV components [17, 22] and thus may get more than 2 MB of L3 cache, which means that simple hash tables can handle more entries than shown in Figure 2.

3.3 Impact of skew

Traffic with higher skews can serve a higher fraction of packets from the cache, which leads to higher throughput and lower latency. To generate traces with different skews, we select flow sizes such that they follow a Zipf distribution and then keep the original source and destination IPs from the CAIDA trace.

We start by comparing different algorithms for different skews for data-structure of size 32 MB; this size ensures that the data cannot fit in the L3 cache. The simple hash table still works the best compared to the sketch and heap. The throughput of all algorithms increases as the skew increases from 0.5 to 2 (Figure 3(a)). For Hash and Count-Min sketch, higher skews lead to more packets being handled by the entries in the cache. The Heap throughput increases more than the simple hash table and Count-Min sketch with higher skews. This is because in the min-heap, heavy-hitter entries are often in the leaves of the Heap. With higher skews, there are more counter updates at the leaves of the Heap and fewer heapify operations that reorganize the counters in the heap.

To understand the impact of skews with different table sizes, we pick three table sizes: 0.125 MB, 2MB, and 32 MB, which fits in L2 cache, L3 cache, and memory respectively. Figure 3(b) shows that with all table sizes, the throughput for the hash table increases with higher skews because more packets are handled by a faster cache. When the skew increases over 1.5 for the 32 MB table size, even the 95% of accesses could be handled in the L3 cache.

We also use our model to understand the impact of skews on other measurement tasks. Remember that for the example in Section 3.2, to reach 14.8 Mpps, 70th percentile of entries must fit into the cache. Figure 2(a) shows that for the skew of 1 and 44 bytes per entry, cache handles 70th percentile of 5.5 million entries. When the skew increases to 1.05, cache handles 70th percentile of 100 million entries, which means that we can support all the measurement tasks in Table 2 and 3 up to 40 Gbps on a core.

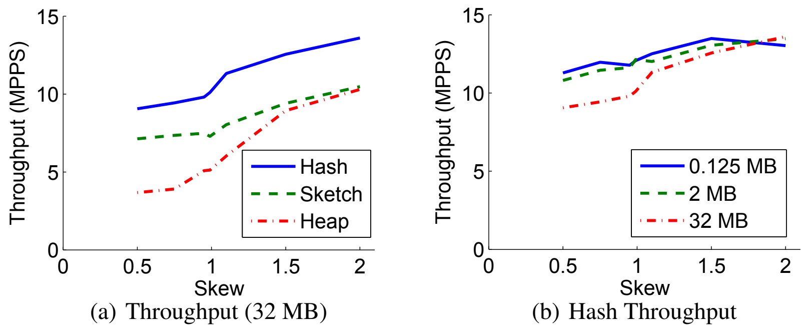
3.4 Across multiple cores

Measurement tasks are never standalone programs. With a pipeline of network functions, it becomes hard for a single core to sustain line rate processing. We can split the traffic on multiple cores where each core executes the pipeline for part of traffic [17]. This means that we may have separate measurement functions run on separate cores but synchronize the states across cores. For example, to detect heavy hitters, there are two options: (a) multiple cores access the shared data structure. Using locks for consistency has huge overhead [8]. An advantage of the simple hash table is that it is easy to implement hash tables in a lockless fashion. For example, we can use compare-and-swap to update a counter in an atomic way. In contrast, it is harder to implement lockless access for more complex data structures such as a heap. (b) each core maintains a separate data structures and merge them when reporting [4].

We compared these two options on two cores: (a) a shared hash table based on compare-and-swap and (b) two separate hash tables. Figure 4 shows that over different sizes, the per packet delay of separate hash tables is lower than a shared hash table in average and at the tail. For example, when the hash table size is 32 KB, the average (95%) delay of the shared hash table is 2x (2.7x) of the separate one. This is because even with lockless data structures, sharing data among cores can increase the delay of accessing data in L3 from 16 ns to 30 ns [32]. It is not useful to save memory with the shared data structure, because it is the working set (affected by traffic skew and access pattern) that matters instead of the total memory usage.

The delay for the separated case increases slightly with larger sizes. In contrast, the delay of the shared case decreases with larger data structure. This is because it is less likely for two cores to access the same entry when there are more entries in a large hash table.

[4]Note that the report frequency is usually low (e.g., once per second) and does not cause performance issue.



(a) Throughput (32 MB) (b) Hash Throughput

Figure 3: Performance comparison of simple hash table for different traffic skews (Legend denotes data structure sizes in MB)

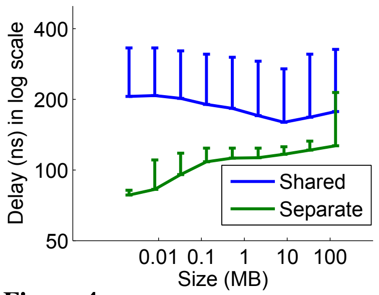
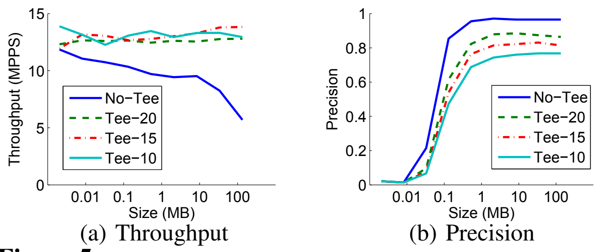


Figure 4: Average and 95% latency of shared and separated counters on multiple cores



(a) Throughput (b) Precision

Figure 5: Benefit of "tee" (Legend numbers are sampling rate (10%, 15%, and 20%))

In summary, we do not need to use shared data structure among the measurement components on different cores. Instead, we just implement the simple hash table on each core and merge them in the report time.

4. OPTIMIZE MEASUREMENT FUNCTION IN SOFTWARE

Even with the simple hash table, there are still cases where the system performance has bottleneck because of traffic changes or other resource-intensive functions in the pipeline. In this section, we discuss new directions for optimizing measurement functions in software: dynamically tuning the packet-processing pipeline for measurement functions; or redesigning measurement solutions to fit better with software architecture.

Dynamic pipeline tuning. The performance of a packet-processing pipeline (either the measurement function or other network functions in the same pipeline) may change with traffic properties (e.g., skews) [39]. To ensure stable performance, we propose a dynamic pipeline tuning system with two key insights of measurement functions: (a) We can trade-off measurement accuracy for performance. For example, when traffic is less skewed, we may change the data structure size to ensure stable throughput and lower per packet delay. In Figure 3(b), when the skew changes to 0.5, we can tune the hash table size to 2 MB to maintain the throughput above 10 Mpps. (b) We can move measurement functions to another core to reduce the delay of critical path of packet-processing pipeline or reduce L2 cache contention. We replace the measurement function in the pipeline with a "tee" component that copies sampled packets to a ring buffer. We then run the measurement function on a separate core that reads packets from the ring. Figure 5 shows the performance of a pipeline with a tee component with different sampling rates in finding heavy hitters for traffic with skew 0.5. Because of the low skew, the throughput of the simple hash table ("Not-Tee") with large data structure is low, which causes packet loss. With the tee approach, for a hash table with size 2 MB, the tee component that samples 15% of packets can sustain 13 Mpps and reach 75% precision while without tee, the throughput is 9.4 Mpps and precision is 97%. We leave it to future work on how to dynamically adapt both size and sampling rate to reach the best throughput and accuracy.

Customize measurement programs to fit software architecture. We can also customize measurement solutions to leverage key features in the software architecture such as prefetching and Single Instruction, Multiple Data (SIMD). For example, we can use SIMD instructions to calculate the hash of four packets together, and then, similar to [26], use prefetching to ensure that the relevant data for measurement are available in the cache.

5. CONCLUSION

In conclusion, this paper re-evaluates the measurement algorithms used for reducing memory usage in the new context of software servers. Our experiments and analysis show that a simple hash table actually works fine for a variety of measurement scenarios. To handle those scenarios that a simple hash table does not work (e.g., near uniform traffic), we discuss ways to dynamically tune the packet processing pipeline for measurement functions or redesigning measurement solutions to fit better in software. We hope this paper can shed light on future efforts on managing states for NFVs and designing measurement algorithms.

6. ACKNOWLEDGMENT

We thank the anonymous reviewers for their helpful feedbacks. This work is supported in part by the grants CNS-1453662, DHS-D15PC00184, and a Google faculty research award.

7. REFERENCES

[1] http://www.7-cpu.com/cpu/Haswell.html.

[2] Benchmarks : Intel mobile haswell (crystalwell): Memory sub-system. http://www.sisoftware.net/?d=qa&f=mem_hsw.

[3] CAIDA Anonymized Internet Traces 2012. http://www.caida.org/data/passive/passive_2012_dataset.xml.

[4] DPDK. http://dpdk.org.

[5] AT&T Domain 2.0 vision white paper. http://goo.gl/YqSjkA, 2013.

[6] M. Alizadeh, A. Greenberg, D. A. Maltz, J. Padhye, P. Patel, B. Prabhakar, S. Sengupta, and M. Sridharan. Data Center TCP (DCTCP). *SIGCOMM computer communication review*, 41(4):63–74, 2011.

[7] T. Benson, A. Akella, and D. A. Maltz. Network Traffic Characteristics of Data Centers in the Wild. In *IMC*, 2010.

[8] S. Boyd-Wickizer, M. F. Kaashoek, R. Morris, and N. Zeldovich. Non-scalable Locks Are Dangerous. In *Linux Symposium*, 2012.

[9] M. Charikar, K. Chen, and M. Farach-Colton. Finding Frequent Items in Data Streams. In *Automata, Languages and Programming*, pages 693–703. Springer, 2002.

[10] B. Claise. Cisco systems NetFlow services export version 9. 2004.

[11] G. Cormode and M. Hadjieleftheriou. Finding Frequent Items in Data Streams. *VLDB Endowment*, 1(2), 2008.

[12] G. Cormode and S. Muthukrishnan. An Improved Data Stream Summary: The Count-Min Sketch and its Applications. *Journal of Algorithms*, 55(1), 2005.

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

[14] G. Cormode and S. Muthukrishnan. Summarizing and Mining Skewed Data Streams. In *SIAM International Conference on Data Mining*, 2005.

[15] A. Das. Intel's Embedded DRAM: New Era of Cache Memory, Overcoming SRAM's scaling. http://www.eetimes.com/author.asp?doc_id=1323410, 2014.

[16] M. Dobrescu, K. Argyraki, G. Iannaccone, M. Manesh, and S. Ratnasamy. Controlling Parallelism in a Multicore Software Router. In *PRESTO*, 2010.

[17] M. Dobrescu, K. Argyraki, and S. Ratnasamy. Toward Predictable Performance in Software Packet-Processing Platforms. In *NSDI*, 2012.

[18] N. Duffield, C. Lund, and M. Thorup. Estimating Flow Distributions from Sampled Flow Statistics. In *SIGCOMM*, 2003.

[19] D. Egloff and M. Leippold. Quantile Estimation with Adaptive Imprtance Sampling. *The Annals of Statistics*, 38(2):pp. 1244–1278, 2010.

[20] C. Estan and G. Varghese. New Directions in Traffic Measurement and Accounting. In *SIGCOMM*, 2002.

[21] É. Fusy, G. Olivier, and F. Meunier. Hyperloglog: The Analysis of a Near-optimal Cardinality Estimation Algorithm. In *Analysis of Algorithms (AofA)*, 2007.

[22] A. Ghodsi, V. Sekar, M. Zaharia, and I. Stoica. Multi-resource Fair Queueing for Packet Processing. In *SIGCOMM*, 2012.

[23] M. Greenwald and S. Khanna. Space-efficient Online Computation of Quantile Summaries. In *SIGMOD*, 2001.

[24] N. Hohn and D. Veitch. Inverting Sampled Traffic. In *IMC*, 2003.

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

[26] A. Kalia, D. Zhou, M. Kaminsky, and D. G. Andersen. Raising the Bar for Using GPUs in Software Packet Processing. In *NSDI*, 2015.

[27] F. Khan, N. Hosein, C.-N. Chuah, and S. Ghiasi. Streaming Solutions for Fine-Grained Network Traffic Measurements and Analysis. In *ANCS*, 2011.

[28] E. Kohler, R. Morris, B. Chen, J. Jannotti, and M. F. Kaashoek. The Click Modular Router. *Transaction on Computer Systems*, 18(3):263–297, Aug. 2000.

[29] F. Korn, S. Muthukrishnan, and Y. Wu. Modeling Skew in Data Streams. In *SIGMOD*, 2006.

[30] A. Kumar, M. Sung, J. J. Xu, and J. Wang. Data Streaming Algorithms for Efficient and Accurate Estimation of Flow Size Distribution. In *SIGMETRICS*, 2004.

[31] A. Lall, V. Sekar, M. Ogihara, J. Xu, and H. Zhang. Data Streaming Algorithms for Estimating Entropy of Network Traffic. In *SIGMETRICS/Performance*, 2006.

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

[33] P. Li and C.-H. Zhang. A New Algorithm for Compressed Counting with Applications in Shannon Entropy Estimation in Dynamic Data. In *COLT*, 2011.

[34] G. S. Manku, S. Rajagopalan, and B. G. Lindsay. Random Sampling Techniques for Space Efficient Online Computation of Order Statistics of Large Datasets. In *SIGMOD*, 1999.

[35] J. Martins, M. Ahmed, C. Raiciu, V. Olteanu, M. Honda, R. Bifulco, and F. Huici. ClickOS and the Art of Network Function Virtualization. In *NSDI*, 2014.

[36] A. Metwally, D. Agrawal, and A. El Abbadi. Efficient computation of frequent and top-k elements in data streams. In *ICDT*, 2005.

[37] M. Mitzenmacher, T. Steinke, and J. Thaler. Hierarchical heavy hitters with the space saving algorithm. *arXiv preprint arXiv:1102.5540*, 2011.

[38] M. Moshref, M. Yu, and R. Govindan. Resource/Accuracy Tradeoffs in Software-Defined Measurement. In *HotSDN*, 2013.

[39] A. Nucci and S. Ranjan. Method and apparatus for worm detection and containment in the internet core, 2010. US Patent 7,712,134.

[40] P. Patel, D. Bansal, L. Yuan, A. Murthy, A. Greenberg, D. A. Maltz, R. Kern, H. Kumar, M. Zikos, H. Wu, C. Kim, and N. Karri. Ananta: Cloud Scale Load Balancing. In *SIGCOMM*, 2013.

[41] R. Schweller, Z. Li, Y. Chen, Y. Gao, A. Gupta, Y. Zhang, P. A. Dinda, M.-Y. Kao, and G. Memik. Reversible Sketches: Enabling Monitoring and Analysis over High-speed Data Streams. *Transaction on Networking*, 15(5), Oct. 2007.

[42] V. Sekar, M. K. Reiter, and H. Zhang. Revisiting the Case for a Minimalist Approach for Network Flow Monitoring. In *IMC*, 2010.

[43] A. Vahdat. Enter the Andromeda zone - Google Cloud Platform's Latest Networking Stack. http://goo.gl/smN6W0.

[44] S. Venkataraman, D. Song, P. B. Gibbons, and A. Blum. New Streaming Algorithms for Fast Detection of Superspreaders. In *NDSS*, 2005.

[45] A. Verma, L. Pedrosa, M. Korupolu, D. Oppenheimer, E. Tune, and J. Wilkes. Large-scale cluster management at Google with Borg. In *EuroSys*, 2015.

[46] R. Wang, D. Butnariu, and J. Rexford. OpenFlow-based Server Load Balancing Gone Wild. In *Hot-ICE*, 2011.

[47] M. Yu, L. Jose, and R. Miao. Software Defined Traffic Measurement with OpenSketch. In *NSDI*, 2013.

[48] Y. Zhang. An Adaptive Flow Counting Method for Anomaly Detection in SDN. In *CoNEXT*, 2013.