Public Review for

Accelerating Network Measurement in Software

Y. Zhou, O. Alipourfard, M. Yu, T. Yang

The paper proposes an idea for how to leverage the benefits of caching in the context of network measurement software systems. They construct a data structure that improves the efficiency of network traffic monitoring in software, and propose a low-level mechanism to improve the efficiency of a variety of schemes, including hash-based and heap-based mechansims. The premise is that current network cards process packets in batches that depend on packet arrival, which is not necessarily efficient. Their scheme, Agg-Evict, aggregates packets in virtual queues, which can use application-level flow semantics to maximize cache efficiency. This approach provides benefits for count-based network measurements such as flow size count or heavy hitter detection. The authors use Agg-Evict as pre-processing module for popular count-based algorithms, and show consistent performance improvements in a variety of scenarios. The authors are releasing their tools as open source in the interest of reproducibility.

Public review written by
KC Claffy
CAIDA

Accelerating Network Measurement in Software

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

Omid Alipourfard
Yale University
omid.alipourfard@yale.edu

Minlan Yu
Harvard University
minlanyu@seas.harvard.edu

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

ABSTRACT

Network measurement plays an important role for many network functions such as detecting network anomalies and identifying big flows. However, most existing measurement solutions fail to achieve high performance in software as they often incorporate heavy computations and a large number of random memory accesses. We present Agg-Evict, a generic framework for accelerating network measurement in software. Agg-Evict aggregates the incoming packets on the same flows and sends them as a batch, reducing the number of computations and random memory accesses in the subsequent measurement solutions. We perform extensive experiments on top of DPDK with 10G NIC and observe that almost all the tested measurement solutions under Agg-Evict can achieve 14.88 Mpps throughput and see up to 5.7× lower average processing latency per packet.

CCS CONCEPTS

• **Networks → Network measurement**;

KEYWORDS

Software packet processing; Network measurement

1 INTRODUCTION

Software packet processing becomes increasingly important to serve various network functions such as load balancing, firewalls, and anomaly detection [2, 20, 30]. Many network functions involve collecting various traffic information about packet numbers or flow numbers such as heavy hitter detection [15], DDoS detection [34], entropy estimation [22]. We call this kind of measurement *count-based measurement* and focus on improving their performance in this paper.

To support count-based measurement, previous work has developed algorithms and data structures that work for software [11, 19, 26]. However, most solutions are designed for specific functions. For example, Count-Min sketch[11] is used for flow size counting; Space-Saving [26] is used for heavy hitter detection; and FM sketch (FM) [16] is used for estimating unique flow counts. SketchVisor [19] provides generic support for various measurement functions. But under high traffic load, SketchVisor needs to activate a fast path with approximate computation that affects measurement accuracy.

Many solutions involve heavy computations or many random memory accesses. For example, both the Count-Min sketch [11] for flow size counting and Reversible sketch (RevSketch) [33] for heavy change detection need three or more hash computations per packet; Space-Saving [26] for tracking top flows and UnivMon [25] for a variety of functions need to update min-heap(s) for each packet. The literature [19] shows that RevSketch spends 95% CPU cycles on hash computations; UnivMon spends 47% CPU cycles on heap maintenance. Moreover, many solutions incur a large number of random memory accesses because they maintain multiple nonadjacent counters for each flow or do heapifying operation for each incoming packet. These random memory accesses increase the probability of cache misses and degrade the packet processing performance.

The key question is *how to speed up measurement performance in software for a variety of count-based measurement solutions.*

We observe that most count-based measurement data structures satisfy the commutable and aggregatable properties: changing the ordering of incoming packets doest not affect the measurement accuracy; and multiple updates on the same flow can aggregate into a single update. With these properties, we can achieve *sublinear processing time* per packet: aggregately updating an entry n times in a batch is much faster than updating the entry separately n times. Take the Count-Min Sketch [11] as an example. It is faster to hash and process the n updates once than hashing and processing one update n times. This not only results in saving computation cycles, it reduces the number of random memory accesses, too. Similarly, for Space-Saving, which uses a min-heap and a hash table to track top flows, an aggregated update means updating the min-heap only once with one tracked flow size in the heap incremented a larger value. The sublinear processing time from aggregated updates is generically true for a variety of measurement data

*Corresponding author: Tong Yang (yangtongemail@gmail.com).

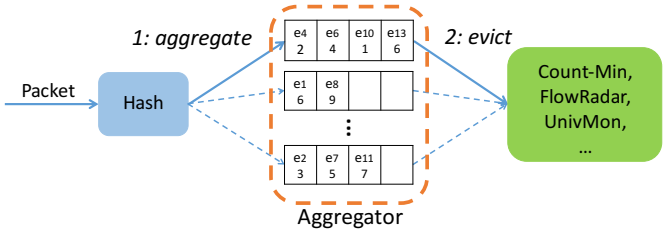


Figure 1: Agg-Evict framework.

structures such as FlowRadar [24], UnivMon [25] (counters arrays with heaps for a lot of functions), RevSketch [33], TwoLevel [41] (combination of RevSketch and bitmaps for detecting DDoS/Superspreader), MRAC [21] (a counter array for measuring flow size distribution), FM [16], and Linear Counting (LC) [35] (a counter array for measuring unique flow number).

In fact, today, software switches already read a batch of packets from the NIC and send them to the application all at once. Such batch processing allows memory prefetching and increases cache locality [28]. However, it is hard to identify the right batch size. We cannot use a large batch size (*i.e.*, 1024 packets or more). This is because the large number of packets cannot fit in cache, which leads to bad cache locality and poor performance. On the other hand, with a small batch size, there is only a small number of packets on the same flows per batch (see § 2), missing the opportunities to fully use the benefits of aggregated updates.

To address this dilemma, rather than relying on the chances that packets in the same batch may access the same entry, we propose a framework called Agg-Evict that proactively aggregates packets in a data structure called an **aggregator** so that we can process the aggregated flows more efficiently. Figure 1 shows the two phases of our design: We *aggregate* the packets according to their packet headers and temporarily store the flow IDs[1] with their aggregated frequencies in an aggregator. When the aggregator is full, we *evict* some of the stored flow IDs to the subsequent measurement solutions. Each evicted flow ID often has frequencies larger than one, but only incurs one insertion in the measurement solutions. This fully utilizes the benefits of aggregated updates, cutting down a lot of computations and random memory accesses. Besides, our aggregator is small (around 256KB) and can easily fit in the L2 or L3 cache. During measurement, most operations happen in this small aggregator and thus achieving high cache locality. In addition, we use SIMD instructions [4] in modern CPU to efficiently implement the operations in the aggregator and carefully design the storage layout of flow IDs in the aggregator to guarantee that each packet only needs two or three cache line[2] accesses.

Our Agg-Evict framework is applicable to a variety of measurement data structures that satisfy the commutable and aggregatable properties and can improve their performance significantly. We implement nine existing measurement solutions in software, and perform extensive experiments on top of DPDK [3] using CAIDA traces [5]. We observe that almost all the tested measurement solutions under our framework can achieve 14.88 Mpps throughput and see up to 5.7× lower average processing latency per packet. We have released the source code, datasets, and detailed descriptions of how to replicate our results at GitHub [1].

2 DILEMMA IN BATCH PROCESSING

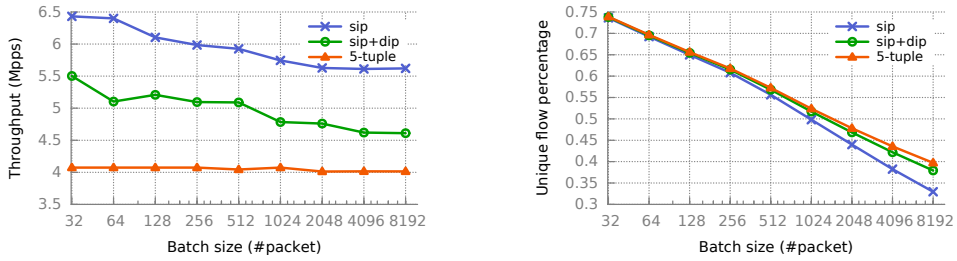
Although batch processing improve the software performance (*e.g.*, by allowing memory prefetching and increasing cache locality [28]), there is a dilemma of how to identify the right batch size. On the one hand, we cannot use a large batch size, because in practice the throughput of batching processing will degrade as the batch size increases. To demonstrate such trend of throughput degrading, we use FlowRadar to process real traffic with different batch sizes (*i.e.*, we send packets to FlowRadar as a batch, see §4.1). Figure 2(a) shows that the increasing batch size leads to throughput degrading regardless of the types of flow IDs. Specifically, using source IP and source-destination IP as the flow ID, the throughput of FlowRadar decreases to 87.4% and 83.8%, seperately, when the batch size increases from 32 to 8192 packets. This is because the large memory size of total packets in one batch cannot fit in cache very well, leading to slightly bad cache locality. We only observe slight throughput degrading when using 5-tuple; the reason might be that processing and storing 5-tuple in FlowRadar need more hash computations and cost more memory (*i.e.*, cache does not work well), making the cache locality less dominant.

On the other hand, we should not use a small batch size; otherwise, we cannot fully use the benefits of aggregated updates. The benefits of aggregated updates (*i.e.*, sublinear processing time) can be indicated by the percentage of unique flows per batch. Figure 2(b) shows a larger batch size to have a lower unique flow percentage. Specifically, using source IP as the flow ID, the benefits increase to 2.23 times when the batch size increases from 32 to 8192 packets. The reason is that in a large batch, there are more chances to see the same flows.

In the next section, we will show how Agg-Evict framework solves this dilemma, making measurement solutions operate at the ideal points of the two figures.

[1]The Flow ID, uniquely identifying a flow, can be the source IP, destination IP, 5-tuple, and so on extracted from packet headers.

[2]Cache line is the minimum memory unit that one memory operation can read/write.



(a) Throughput (FlowRadar).

(b) Avg. unique flow percentage per batch.

Figure 2: Impact of batch size (on CAIDA [5] traces).

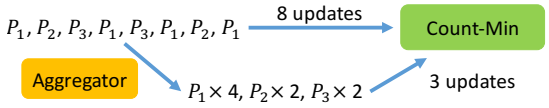


Figure 3: Demonstration of Agg-Evict design.

3 THE AGG-EVICT FRAMEWORK

We first discuss the count-based measurement that Agg-Evict framework focuses on in §3.1. Next, we discuss the commutable and aggregatable properties that our design relies on to improve the performance of count-based measurement solutions in §3.2. We then describe our Agg-Evict design in §3.3, and discuss two important operations of our design in detail in §3.4 and 3.5, respectively.

3.1 Count-Based Measurement

Agg-Evict framework focuses on count-based measurement, which refers to the measurements that collect various traffic information about packet numbers or flow numbers. We summarize count-based measurement tasks and typical solutions into Table 1. For many more sketches, please refer to the literature [10, 12–14, 18, 36–40, 42]

Table 1: Count-based measurement.

Tasks	Typical solutions
Flow size counting	Count-Min [11], FlowRadar [24]
Heavy hitter detection	Count-Min [11], FlowRadar [24], Space-Saving [26], UnivMon [25]
Heavy change detection	RevSketch [32], FlowRadar [24], UnivMon [25]
Superspreader/DDoS detection	TwoLevel [41], UnivMon [25]
Flow size distribution	MRAC [21], FlowRadar [24]
Cardinality estimation	FM [16], LC [35]
Entropy estimation	FlowRadar [24], UnivMon [25]

3.2 Commutable and Aggregatable Properties

Most count-based measurement solutions are commutable and aggregatable in terms of accuracy: *commutable* means processing item a before b is equal to processing b before a; *aggregatable* means processing the same item a n times separately can be aggregated into processing a n times in a batch (**aggregated update**). This is because we maintain counters or bitmaps for flow number or packet number, and such maintenance (*i.e.*, addition and setting bits on) is commutable and aggregatable. Let $T(n)$ denote the time of processing an item with frequency of n. Aggregated updates can bring potential benefits of *sublinear processing time*: $T(n) \ll n \times T(1)$ when $n \gg 1$. That is, aggregately updating an entry n times in a batch is much faster than updating the entry separately n times.

We now show why most measurement solutions can get sublinear processing time from aggregated updates. There are mainly two classes of measurement solutions: hash-based and heap-based [7][3]. Hash-based solutions include: Count-Min [11], FlowRadar [24], Reversible Sketch (RevSketch) [33], TwoLevel [41] (combination of RevSketch and bitmaps for detecting DDoS/Superspreader), MRAC [21] (a counter array for measuring flow size distribution), FM Sketch (FM) [16], and Linear Counting (LC) [35] (a counter array for measuring unique flow number). Take the Count-Min sketch as an example. Here, we use Count-Min to count the packet number of each unique flow. Normally, for each incoming item, we first compute d hash functions and then increment the d hashed counters by one. Aggregating n updates of an item means computing d hash and then incrementing the d hashed counters by n. Compared with n normal updates, aggregated update saves $n-1$ hash computations and $n-1$ memory accesses, thus getting sublinear processing time.

[3]The sketch-based and sampling-based solutions in the literature [7] can be considered both as hash-based, since both of their major operations are doing hash and then updating entry (or entries).

Heap-based solutions include: Space-Saving [26] and UnivMon [25] (counters arrays with heaps for a lot of functions). Space-Saving [26] needs to compute a hash function and update a min-heap for each incoming item. Processing an item with frequency of n means computing one hash, and then doing a deeper heapifying operation than normal update. Compared with n normal updates, aggregated update saves $n-1$ hash computations, and transforms n heapifying operations into one deeper heapifying operation. The cost of this deeper heapifying operation is constrained by the heap depth h, which is a fixed value in Space-Saving. This fixed value is a relatively small (often around 10), since it is logarithmic to the number of tracked top flows. That is, this deeper heapifying operation will not exceed h steps, while the normal heapifying operation needs at least one step. Thus, when $n \geqslant h$, the cost of this deeper heapifying operation will be lower than the cost of n normal heapifying operations, leading to sublinear processing time. If we additionally consider the saved $n-1$ hash computations, n even does not need to be larger than h to make $T(n) \ll n \times T(1)$ hold. UnivMon [25] uses a universal sketch to simultaneously collect different types of traffic statistics and relies on multiple heaps to maintain top flows. Similar to Space-Saving, it also gets sublinear processing time from aggregated updates.

3.3 Overall Design

In order to do aggregated updates to save computations and memory accesses, we design an Agg-Evict framework that works for all measurement solutions with the commutable and aggregatable properties. This framework separates the measurement into two phases: aggregation and eviction. In the aggregation phase, we proactively *aggregate* flow IDs across all the incoming packets. We store the unique flow IDs and their aggregated frequencies in a data structure called **aggregator** using a few and consecutive memory accesses (high cache locality). In the eviction phase, when the aggregator is full, we *evict* some flow IDs stored in it with their aggregated frequencies into the existing measurement solutions at a time, doing aggregated updates (the aggregated frequencies may not be needed for some solutions like Bloom Filter [9]). In this way, we get sublinear processing time (§3.2).

We use an example to demonstrate this two-phase design and its benefits (see Figure 3). In a Count-Min sketch, suppose we have 8 incoming packets with flow IDs: $P_1, P_2, P_3, P_1, P_3, P_1, P_2, P_1$. Normally, we need to update the sketch eight times, incurring $8 \times d$ hash computations and $8 \times d$ random memory accesses. With Agg-Evict, we first aggregate the 8 flow IDs into 3 unique flow IDs with separate aggregated frequencies: $P_1 \times 4$, $P_2 \times 2$, $P_3 \times 2$, and then evict the aggregated results into the Count-Min sketch. We only incur 3 updates (*i.e.*, $3 \times d$ hash computations and $3 \times d$ random memory accesses). In this way, we improve the processing speed of the Count-Min sketch by $\frac{8}{3}$ times, with additional small overhead of aggregation operations. We will explain how we make the overhead of aggregation operations small in §3.4.

We now describe the aggregator design and Agg-Evict workflow in detail. Figure 1 shows the data structure of the aggregator. It has k arrays of key-value (KV) pairs and each array has l KV pairs. The key part in each KV pair stores flow ID and the value part stores the corresponding aggregated frequency. Simply put, there are k KV arrays, each of which has the same data structure as a small linear hash table with length of l. We arrange the storage layout of the aggregator to make each KV array store the l flow IDs consecutively, which enables high cache locality as we will discuss in §3.4). For each incoming packet, we compute a hash value based on its flow ID, and use this value to locate a KV array. For convenience, we call this KV array the **hashed KV array** for this packet. Within this hashed KV array, we do not perform any hash operation. Instead, we perform the following operations:

(1) **Aggregation:** if the flow ID of the incoming packet matches the key part of one KV pair (**lookup**), we increment the corresponding value part by one; otherwise (no match), if there are empty pairs in this array, we initiate one arbitrary empty pair with the flow ID and frequency of one;
(2) **Eviction:** otherwise (no match and no empty pairs), we evict one pair in this array by some eviction policy (covered in §3.5). Specifically, we set the key part to the flow ID of the incoming packet and value part to one. The original flow ID and aggregated frequency in the evicted pair will be sent to the subsequent measurement solutions, aggregately updating existing solutions as a batch.

Since the hash computation locates in the critical path of packet processing, we need to do it efficiently to guarantee high processing speed. Thus, we choose the simple modulo operation as the hash function (*i.e.*, $flow\ ID\ \%\ k$). Using modulo operation as hash function may cause load imbalance across the k KV arrays. However, our experiments show that it incurs much smaller implementation overhead compared with more advanced hash functions, leading to higher processing speed. Due to space limitation, we do not show the related figures.

There are several challenges during aggregation and eviction. In aggregation, for each incoming packet, we need to compare its flow ID with the l existing keys stored in the hashed KV array. If we just use linear search to do this comparison, there are many branch statements, leading to low

performance in software. Instead, we anticipate a more efficient comparison scheme without any branch statement to guarantee the performance. In eviction, we need to choose one pair to evict, which should guarantee high aggregation level without incurring much implementation overhead. In the following subsections, we will show how we address these challenges.

3.4 Aggregation with SIMD and High Cache Locality

Algorithm 1: Lookup in a KV array with 16 pairs via the SSE2 SIMD instructions (assuming 4-byte key).

Input: The flow ID e of incoming packet, the start address p of the key part array in the hashed KV array.
Output: Return the index of the matched key or -1.

```
   /* load flow ID into a 128-bit SSE2 register   */
1  const __m128i item = _mm_set1_epi32 (e);
   /* convert address type                        */
2  __m128i * keys_p = (__m128i *) p;
   /* compare the flow ID with the 16 keys        */
3  __m128i a_comp = _mm_cmpeq_epi32(item, keys_p[0]);
4  __m128i b_comp = _mm_cmpeq_epi32(item, keys_p[1]);
5  __m128i c_comp = _mm_cmpeq_epi32(item, keys_p[2]);
6  __m128i d_comp = _mm_cmpeq_epi32(item, keys_p[3]);
   /* get the final matching results              */
7  a_comp = _mm_packs_epi32(a_comp, b_comp);
8  c_comp = _mm_packs_epi32(c_comp, d_comp);
9  a_comp = _mm_packs_epi32(a_comp, c_comp);
10 int matched = _mm_movemask_epi8(a_comp);
   /* return index or -1 according to matched     */
11 return (matched ≠ 0 ? TZCNT(matched) : −1);
```

We leverage SIMD instructions [4] supported by modern CPUs to do efficient lookup and consider the cache line access of each lookup to purse high cache locality. We start with the assumption that the flow ID or key is only 4 bytes (*e.g.*, source IP or destination IP), and then discuss how to handle large flow IDs (*e.g.*, source-destination IP pair, five-tuple, *etc.*) at the end of this section.

SIMD-based lookup: The key idea is to do comparison between the flow ID of incoming packet with each key stored in the hashed KV array in parallel via SIMD, and encode the l (*i.e.*, the number of KV pairs in each KV array) comparison results into l bits in one integer to avoid branch statements. For a better demonstration, we show the specific implementation with $l = 16$ in Algorithm 1.

In this algorithm, the incoming packet has a flow ID e. Each SSE2 register has 128 bits, which is specified by SIMD instructions in CPU. Line 1-2 load 16 copies of e and the 16 keys into 4 SSE2 registers, respectively. Line 3-6 complete 16 comparisons in four-way parallel. *_mm_cmpeq_epi32* instruction compares the four keys in *item* and the four keys in *keys_p[0]* for equality. If two keys are equal, the corresponding 32 bits in *a_comp* will be set on; otherwise, these 32 bites will be set off. Line 4-6 work similarly.

Line 7-10 encode the 16 comparison results into 16 bits in one integer. *_mm_packs_epi32* instruction considers each key from the two SSE2 registers as a 32-bit integer, converts each 32-bit integer to a 16-bit integer by signed saturation, and then packs these eight 16-bit integers into the 128-bit SSE2 register *a_comp* one-by-one. After line 7-9, the matching results of 16 keys in original key part array are indicated by the most significant bits of the 16 consecutive signed 8-bit integers in *a_comp*. In line 10, *_mm_movemask_epi8* instruction creates a 16-bit mask from the most significant bits of the 16 consecutive signed 8-bit integers in *a_comp*, and zero extends this mask to a 32-bit integer *matched*.

Line 11 returns the index of the matched key or -1. If *matched* is not zero, there must be a **matching case** happening among the 16 keys. In this case, we use *TZCNT* instruction to get the number of trailing 0-bits in *matched*. Otherwise, we return *-1*. All the involved SIMD instructions in Algorithm 1 can finish data manipulation in only one CPU cycle. Thus, this implementation is very efficient, and only costs around 10 cycles for each packet.

Due to the constraint of existing SSE2 instruction design, the above implementation is specific to the case of $l = 16$, and cannot be easily expanded for larger l by using more instructions of *_mm_cmpeq_epi*32 and *_mm_packs_epi*32. Thus, if l is large than 16, say 32, we need to run Algorithm 1 twice; if l is indivisible by 16, we need to make some corner checks.

Guaranteeing high Cache locality: Our goal is to minimize the memory access overhead per packet in the aggregator to keep the efficiency of lookup operation. We achieve this by constraining l to a specific value to make the l 4-byte keys exactly fit in one cache line in modern CPU. Typical cache line of modern CPU is 64 bytes [27], which could exactly accommodate 16 4-byte keys. Therefore, we set l to 16 in order to both reduce the number of cache line accesses per packet (*i.e.*, high cache locality) and only run Algorithm 1 once to minimize the lookup overhead.

Handling large flow IDs: Due to the constraint of existing SSE2 instruction design, Algorithms 1 cannot be easily expanded to support keys larger than 4 bytes. However, when the flow ID exceeds 32 bits, we still want to employ the SIMD-based lookup to get high performance. We address this problem by transforming large flow IDs into 32-bit fingerprints, storing these fingerprints as the key part in the aggregator, and using fingerprints to perform the aggregation operations in Algorithm 1. In this way, we can still keep each key part in KV arrays to four bytes, perform efficient

SIMD-based lookup operation on KV arrays, and guarantee high cache locality. Besides, in the case of using fingerprints, to provide flow IDs for the subsequent measurement solutions during eviction, we also need to store the original flow IDs in an array with $k \times l$ elements. We call this array the **flow ID array**.

We now discuss some details during handling large flow IDs, including fingerprint generation and solving fingerprint collisions. We try many different ways to generate 32-bit fingerprints: modulo, bit shifting, XOR, *etc.* We finally find that XOR (*e.g.*, $srcIP \oplus dstIP$) works best among them in terms of both efficiency and probability of fingerprint collisions. This choice still needs further investigations but it works for us well empirically. In terms of fingerprint collisions, the flow ID array can completely help solve it. Specifically, once the matching case happens, we check whether the incoming flow ID actually matches with the flow ID stored in the corresponding position of the flow ID array. If the two flow IDs are the same, we just increment the corresponding value part in the hashed KV array by one. Otherwise (the fingerprint collision happens), we evict the original KV pair and set its value part to one. We note that our fingerprint-based scheme for handling large flow IDs does not influence the accuracy of measurement solutions since we solve the fingerprint collisions completely. We will show the impact of such scheme on processing speed of measurement in §4.3.

3.5 Eviction with GRR Policy

The eviction policy decides which pair we should evict when the hashed KV array is full. It determines the aggregation level of the aggregator and thus influences the whole performance. A natural solution is LRU (Least Recently Used), because it or its variants have been widely used in modern cache design. LRU maintains a timestamp for each KV pair, recording the last time when this KV pair was updated. Once we need to evict a KV pair, we need to scan the l KV pairs in the hashed KV array, find the KV pair owning the smallest time stamp, and evict this pair. Thus, the implementation overhead of LRU is relatively high, since we need to scan l time stamps per eviction.

To reduce the overhead, we propose a new eviction policy called GRR (Global Round Robin) with extremely low implementation overhead. In GRR, for the k KV arrays, we maintain a global eviction index ranging from 0 to $m-1$ that indicates the position where the next eviction will happen. Once we need to evict a KV pair regardless of on which hashed KV array, we will evict the KV pair indicated by this index, and then increment this index by one. Once this index reaches l, we will set it to 0. We maintain one eviction index globally instead of per KV array in order to get high cache locality, since one global index can often be in cache while per-array indexes cannot. The GRR eviction policy resembles a fully randomized policy[4]. The shortcoming of this policy is that it may evict some packets too early to achieve a high aggregation level. However, as we will show in §4.4, it can help Agg-Evict achieve higher performance than LRU for most measurement solutions due to its extremely low implementation overhead. Note that different eviction policies do not influence the accuracy of measurement solutions, since they only influence the packet incoming order that existing measurement solutions see, while these solutions have commutable property (§3.2).

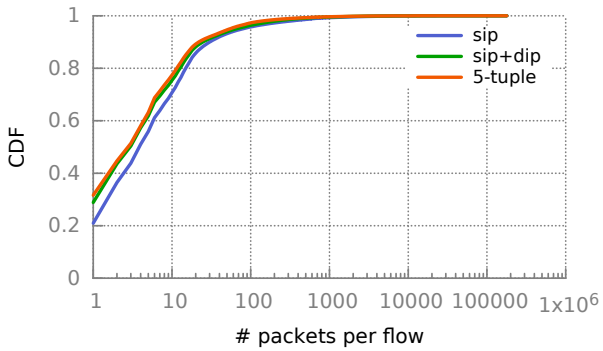


Figure 4: CDF of number of packets per flow in CAIDA traces.

4 PERFORMANCE EVALUATION

4.1 Experimental Setup

Traffic Traces: We use a one-minute trace from Equinix data center at Chicago from CAIDA [5] with 28.82M TCP and UDP packets. We generate 14×2M TCP and UDP packets with payload size of 64 bytes by preserving the 5-tuple as in the original CAIDA traces. The CDF of number of packets per flow in this trace is shown in Figure 4. When using source IP to identify flows, these 28M packets contain 0.59M flows with skewness of around 1.1. Still keeping 64 bytes payload size, we generate multiple traffic traces following a Zipfian distribution [31] with different skewness[5], each of which consists of 14×2M packets using the 5-tuple from CAIDA traces. We use the smallest payload size (*i.e.*, 64 bytes) to create the maximum workload for a measurement solution on a 10G NIC.

Testbed: We use two Ubuntu servers that run on Intel Xeon E5-2650 v4 processors. The processor has 12 cores, 256KB L2 cache, and 30MB shared L3 cache. The servers are connected by two Intel 82599ES 10G NICs and a switch. We use DPDK [3], a set of data plane libraries for packet processing, to

[4]We discard the fully randomized policy, because producing random number can be a relatively costly process in software (approximately equal to one hash function computation).

[5]The skewness refers to the parameter s in the Zipfian distribution [31].

send and receive packets from the NIC. At the receiver side, we used data prefetching and batch processing (i.e., receiving packets at NIC in a batch) provided by DPDK libraries, and additionally implemented another batch processing (i.e., sending packets to measurement algorithms in a batch) manually by using a small packet receiving buffer in memory. We set both batch sizes to 32 by default.

Tested measurement solutions: We summarize the nine tested count-based measurement solutions in Table 1. All related source code, datasets, and detailed descriptions of how to replicate our results are provided at GitHub [1].

Parameter Settings: We set the length of the measurement epoch to 2M packet, which translates to a 130ms time window on a 10G NIC with 64-byte payload in each packet. For all the algorithms, we use source IP as flow ID, allocate 258KB memory to the aggregator ($k = 2000$), and leverage GRR eviction policy. The default settings of nine tested measurement algorithms are as follows:

- *Count-Min, RevSketch, and MRAC*: We use 4 counter arrays each of which has 65536 32-bit counters.
- *FlowRadar*: We use 262144 cells in its IBLT (*i.e.*, invertible bloom lookup tables [17]), use a Bloom filter with memory equal to one-third of the IBLT memory, and use 4 hash functions in both IBLT and Bloom filter.
- *Space-Saving*: We track the top 128 flows.
- *TwoLevel*: We use 4 counter arrays each of which has 65536 32-bit counters in RevSketch, and use 4 additional counter arrays with 4096 12-bit small bitmaps in the second level modified Count-Min.
- *UnivMon*: We use a 4-level sketch with 65536×4, 32768×4, 16384×4, 8192×4 counters for each level. We track the top 128 flows in its heap.
- *FM*: We use 4 32-bit counters.
- *LC*: We use 4 bit arrays each of which has 65536×32 bits.

Metrics: We consider three metrics: throughput, average latency, and tail latency (*i.e.*, 99th percentile latency). When measuring these metrics, we manually adjust the sending rate to reach the maximum value with no packet losses (*i.e.*, zero packet loss tests [6]). The throughput is constrained by the 10G NIC (*i.e.*, 14.88 Mpps at 64-byte payload). As such, we also show the average latency, a good indicator of the ideal throughput assuming infinite bandwidth. The tail latency (shown as error bars in latency figures) reflects the variance of packet processing time. High packet variance results in long queues in NIC, which leads to packet drops – not enough space in the queue – even when the average latency is low. We do not show accuracy figures as our Agg-Evict framework does not impact the accuracy of measurement solutions.

4.2 Benefits of Agg-Evict

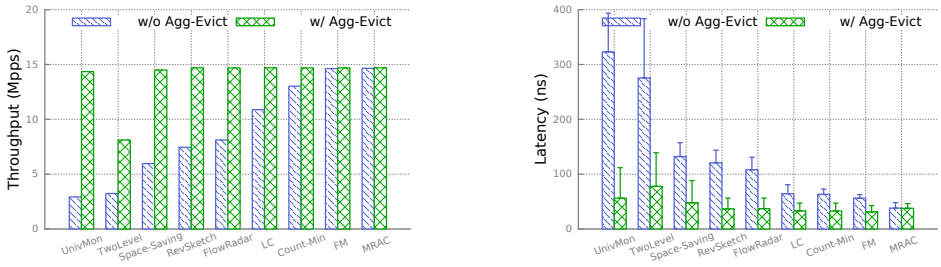
Agg-Evict improves the throughputs and reduces the latencies of all nine tested measurement solutions. Figure 5(a) and 5(b) show the throughputs and latencies of the nine measurement solutions with and without Agg-Evict. With Agg-Evict, all the nine solutions with the exception of TwoLevel algorithm achieve 14.88 Mpps throughput. TwoLevel itself is very slow and uses source-destination IP as flow ID. After using Agg-Evict, we still see 2.5× throughput improvement and 3.5× average latency reduction. With Agg-Evict, the average latencies of RevSketch, FlowRadar, LC, Count-Min, FM and MRAC go below 40 ns – enough to support 16.8 Gbps traffic. SketchVisor [19] uses traffic traces with average packet size of 769 bytes, while here we use average packet size of 84 bytes. After multiplied (divided) by the scaling factor $769/84 \approx 9.2$, our throughput (latency) values are much higher (lower) than SketchVisor.

4.3 Sensitivity Analysis

Agg-Evict improves the throughput and reduces the latency of FlowRadar regardless of the types of flow ID. Larger flow IDs incur extra computation and memory access overhead for measurement solutions. We show this impact in Figure 6(a) and 6(b) by using different flow IDs. We find that even in the case of 5-tuple as flow ID, Agg-Evict can still help FlowRadar achieve throughput larger than 10 Mpps – enough to serve a 10G NIC at the average packet size of 119 bytes (still much smaller than the average packet size in real data centers [8]).

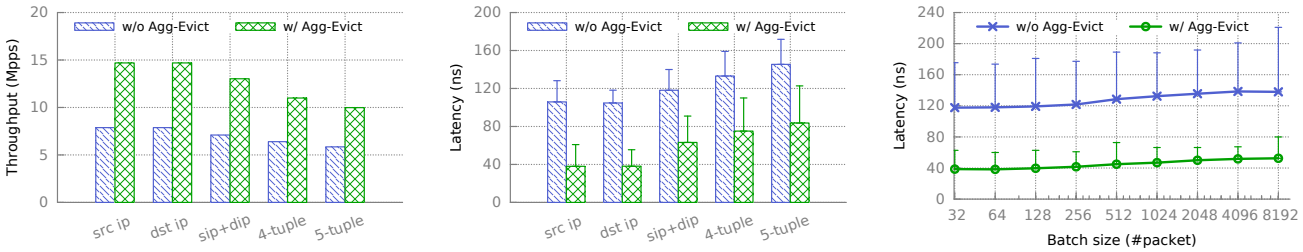
Agg-Evict reduces the processing latency of FlowRadar under different batch sizes. We vary the batch size when sending packets to FlowRadar, and compare the processing latency with and without Agg-Evict, as shown in Figure 6(c). We also vary the batch size when receiving packets from the NIC and observe similar trend as Figure 6(c); we do not plot these results in the interest of space. When batch size increases, the processing latency with and without Agg-Evict both increase gradually. Agg-Evict helps FlowRadar always achieve lower latency and more stable performance (*i.e.*, smaller gap between average and 99th percentage latencies).

Agg-Evict improves the throughput and reduces the latency of FlowRadar under different traffic skewness and aggregator sizes. In order to validate the effectiveness of Agg-Evict in a variety of cases, we vary traffic skewness and aggregator size, and show their impacts on the processing latencies of FlowRadars with and without Agg-Evict in Figure 7(a), 7(b) and 7(c). We observe that in all cases, Agg-Evict reduces the latency of FlowRadar. The reason why throughput or latency on the synthetic trace with traffic skew of 1.1 is lower or higher than CAIDA trace is that the



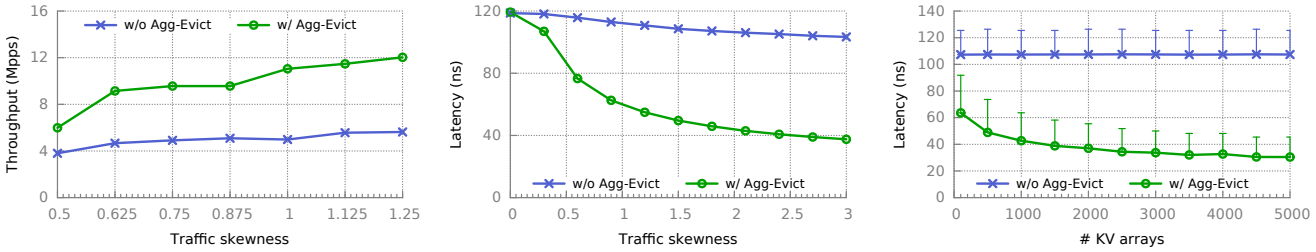
(a) Throughput improvement. (b) Avg. and 99th percentile latency reduction.

Figure 5: Benefits of Agg-Evict on nine typical network measurement solutions.



(a) Throughput: varying flow IDs (FlowRadar). (b) Latency: varying flow IDs (FlowRadar). (c) Latency: varying batch size (FlowRadar).

Figure 6: Impact of flow ID and batch size.



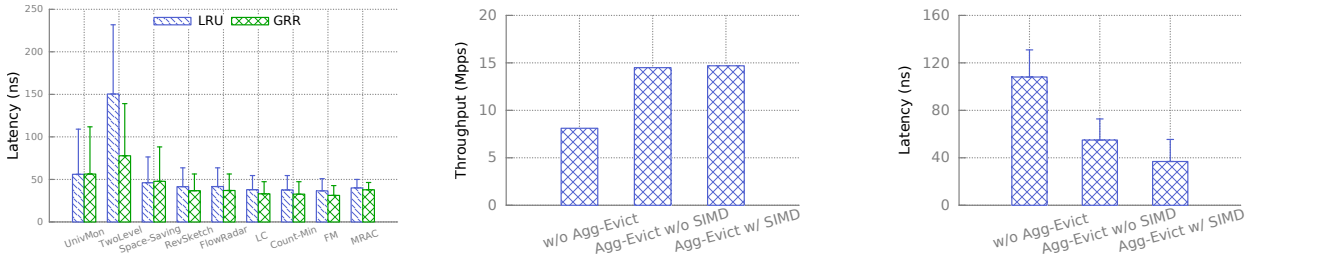
(a) Throughput: varying skewness (FlowRadar). (b) Latency: varying skewness (FlowRadar). (c) Latency: varying aggregator size (FlowRadar).

Figure 7: Impact of traffic skew and aggregator size.

appearance order of packets belonging to the same flows in synthetic trace is randomized, which largely degrades the aggregating performance of Agg-Evict. When traffic skewness increases, the latency of FlowRadar without Agg-Evict declines gradually, while the one with Agg-Evict declines drastically. When aggregator size increases, average and 99th percentage latencies decline, and the gap between them gradually shrinks, which means more stable performance.

4.4 Microbenchmark

For hash-based measurement solutions, GRR achieves better performance than LRU, while for heap-based solutions, LRU is a litter better. Figure 8(a) compares the latencies of nine measurement solutions under different eviction policies. For UnivMon and Space-Saving, they all maintain heaps to track top flows. Due to the heapifying operations, processing an item with frequency larger than one will cost more than processing an item with frequency of



(a) Latency: eviction policy.

(b) Throughput: SIMD instructions (FlowRadar).

(c) Latency: SIMD instructions (FlowRadar).

Figure 8: Comparison of different design choices.

only one. This will amplify the benefits of higher aggregation level brought by LRU.

SIMD-based lookup for Agg-Evict achieves lower latency than linear-search-based lookups. Figure 8(b) and 8(c) show the throughput and latency of Agg-Evict with and without SIMD instructions. Agg-Evict without SIMD instructions employs linear search to check the matching case. We observe that SIMD instructions help lower the latency by around 20 ns. This is due to the higher efficiency of Algorithm 1 than normal linear search. We also observe that FlowRadar under Agg-Evict (with SIMD) achieves 2.92× lower average latency than pure FlowRadar, which corresponds to doing measurement at the ideal points in Figure 2(a) and 2(b) – $1/0.874 \times 2.23 = 2.55$ times better performance.

5 RELATED WORK

Measurement data structures: We summarize the count-based measurement tasks and typical data structures in Table 1. Our Agg-Evict framework can support all the algorithms in this table and improve their processing speed without influencing their accuracy.

Measurement designs in software: The literature [7] observes that in software, simple hash tables work better than more advanced measurement algorithms for a variety of count-based measurement tasks if we do not consider the issue of memory size. Agg-Evict framework can also improve the performance of simple hash tables, and is orthogonal to this literature. SketchVisor [19] finds that many existing measurement solutions cannot achieve satisfying processing speed in software. It augments software count-based measurement algorithms with a *fast path*, which is activated under high traffic load to provide fast measurement with accuracy degradations. In contrast, Agg-Evict improves the processing speed of measurement solutions purely in "fast path", and does not degrade the accuracy of measurement results.

Aggregation mechanism: Marple [29] aggregates statistic information in fast SRAM of switch hardware and periodically merges it into slow DRAM of remote servers to complete line-rate measurement. Agg-Evict improves the performance of count-based measurement by reducing computations and memory accesses and enhancing cache locality. MapReduce schedulers [23] that aggregate keys in data streams for fast in-memory processing share commonalities with our Agg-Evict.

6 CONCLUSION

In this paper, we present a generic framework, namely Agg-Evict, to accelerate count-based network measurement in software. Through aggregating incoming packets on the same flows, Agg-Evict reduces a large amount of computations and random memory accesses of subsequent measurement solutions. Extensive experiments performed on top of DPDK further validate the feasibility and effectiveness of Agg-Evict. We believe that Agg-Evict can be applied to and help accelerate many more network measurement solutions in the future.

7 ACKNOWLEDGMENTS

We thank SIGCOMM CCR reviewers and editor for their insightful comments. This work is supported by Primary Research & Development Plan of China (2016YFB1000304), National Basic Research Program of China (973 Program, 2014CB340405), NSFC (61672061), the OpenProject Funding of CAS Key Lab of Network Data Science and Technology, Institute of Computing Technology, Chinese Academy of Sciences, NSF 1834263 and 1712674.

REFERENCES

[1] Source code and experiment details related to agg-evict. https://github.com/zhouyangpkuer/Agg-Evict.

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

[3] Data Plane Development Kit (DPDK). http://dpdk.org/.
[4] Intel SSE2 Documentation. https://software.intel.com/en-us/node/683883.
[5] The CAIDA anonymized 2016 internet traces. http://www.caida.org/data/passive/passive_2016_dataset.xml.
[6] Zero packet loss tests. https://www.ietf.org/rfc/rfc2544.txt.
[7] O. Alipourfard, M. Moshref, and M. Yu. Re-evaluating measurement algorithms in software. In *HotNets*, page 20. ACM, 2015.
[8] T. Benson, A. Akella, and D. A. Maltz. Network traffic characteristics of data centers in the wild. In *IMC*, pages 267–280. ACM, 2010.
[9] B. H. Bloom. Space/time trade-offs in hash coding with allowable errors. *Communications of the ACM*, 13(7):422–426, 1970.
[10] G. Cormode. Sketch techniques for approximate query processing. *Foundations and Trends in Databases. NOW publishers*, 2011.
[11] G. Cormode and S. Muthukrishnan. An improved data stream summary: the count-min sketch and its applications. *Journal of Algorithms*, 55(1):58–75, 2005.
[12] H. Dai, L. Meng, and A. X. Liu. Finding persistent items in distributed, datasets. In *Proc. IEEE INFOCOM*, 2018.
[13] H. Dai, M. Shahzad, A. X. Liu, and Y. Zhong. Finding persistent items in data streams. *Proceedings of the VLDB Endowment*, 10(4):289–300, 2016.
[14] H. Dai, Y. Zhong, A. X. Liu, W. Wang, and M. Li. Noisy bloom filters for multi-set membership testing. In *Proc. ACM SIGMETRICS*, pages 139–151, 2016.
[15] X. Dimitropoulos, P. Hurley, and A. Kind. Probabilistic lossy counting: an efficient algorithm for finding heavy hitters. *ACM SIGCOMM CCR*, 38(1):5–5, 2008.
[16] P. Flajolet and G. N. Martin. Probabilistic counting algorithms for data base applications. *Journal of computer and system sciences*, 31(2):182–209, 1985.
[17] M. T. Goodrich and M. Mitzenmacher. Invertible bloom lookup tables. In *Proceedings of the 49th Annual Allerton Conference on Communication, Control, and Computing*, pages 792–799. IEEE, 2011.
[18] S. Heule, M. Nunkesser, and A. Hall. Hyperloglog in practice: algorithmic engineering of a state of the art cardinality estimation algorithm. In *Proceedings of the 16th International Conference on Extending Database Technology*, pages 683–692. ACM, 2013.
[19] Q. Huang, X. Jin, P. P. Lee, R. Li, L. Tang, Y.-C. Chen, and G. Zhang. Sketchvisor: Robust network measurement for software packet processing. In *SIGCOMM*, pages 113–126. ACM, 2017.
[20] B. Krishnamurthy, S. Sen, Y. Zhang, and Y. Chen. Sketch-based change detection: methods, evaluation, and applications. In *Proc. ACM IMC*, pages 234–247. ACM, 2003.
[21] A. Kumar, M. Sung, J. J. Xu, and J. Wang. Data streaming algorithms for efficient and accurate estimation of flow size distribution. In *ACM SIGMETRICS Performance Evaluation Review*, volume 32, pages 177–188. ACM, 2004.
[22] A. Lall, V. Sekar, M. Ogihara, J. Xu, and H. Zhang. Data streaming algorithms for estimating entropy of network traffic. In *ACM SIGMETRICS Performance Evaluation Review*, volume 34, pages 145–156. ACM, 2006.
[23] B. Li, E. Mazur, Y. Diao, A. McGregor, and P. Shenoy. A platform for scalable one-pass analytics using mapreduce. In *Proc. ACM SIGMOD*, pages 985–996. ACM, 2011.
[24] Y. Li, R. Miao, C. Kim, and M. Yu. Flowradar: a better netflow for data centers. In *NSDI*, pages 311–324. USENIX Association, 2016.
[25] Z. Liu, A. Manousis, G. Vorsanger, V. Sekar, and V. Braverman. One sketch to rule them all: Rethinking network flow monitoring with univmon. In *SIGCOMM*, pages 101–114. ACM, 2016.
[26] A. Metwally, D. Agrawal, and A. El Abbadi. Efficient computation of frequent and top-k elements in data streams. In *International Conference on Database Theory*, pages 398–412. Springer, 2005.
[27] S. Meyers. Cpu caches and why you care, 2013. (Cited on slice 20).
[28] M. Moshref, M. Yu, R. Govindan, and A. Vahdat. Trumpet: Timely and precise triggers in data centers. In *SIGCOMM*, pages 129–143. ACM, 2016.
[29] S. Narayana, A. Sivaraman, V. Nathan, P. Goyal, V. Arun, M. Alizadeh, V. Jeyakumar, and C. Kim. Language-directed hardware design for network performance monitoring. In *Proc. ACM SIGCOMM*, pages 85–98. ACM, 2017.
[30] P. Patel, D. Bansal, L. Yuan, A. Murthy, A. Greenberg, D. A. Maltz, R. Kern, H. Kumar, M. Zikos, H. Wu, et al. Ananta: Cloud scale load balancing. In *ACM SIGCOMM CCR*, volume 43, pages 207–218. ACM, 2013.
[31] D. M. Powers. Applications and explanations of Zipf's law. In *Proc. EMNLP-CoNLL*. Association for Computational Linguistics, 1998.
[32] R. Schweller, A. Gupta, E. Parsons, and Y. Chen. Reversible sketches for efficient and accurate change detection over network data streams. In *IMC*, pages 207–212. ACM, 2004.
[33] R. Schweller, Z. Li, Y. Chen, et al. Reversible sketches: enabling monitoring and analysis over high-speed data streams. *IEEE/ACM Transactions on Networking (ToN)*, 15(5):1059–1072, 2007.
[34] V. Sekar, N. G. Duffield, O. Spatscheck, J. E. van der Merwe, and H. Zhang. Lads: Large-scale automated ddos detection system. In *ATC*, pages 171–184. USENIX Association, 2006.
[35] K.-Y. Whang, B. T. Vander-Zanden, and H. M. Taylor. A linear-time probabilistic counting algorithm for database applications. *ACM Transactions on Database Systems (TODS)*, 15(2):208–229, 1990.
[36] K. Xie, X. Li, X. Wang, and et al. On-line anomaly detection with high accuracy. *IEEE/ACM Transactions on Networking*, 2018.
[37] T. Yang, J. Gong, H. Zhang, S. L. Zou, Lei, and X. Li. Heavyguardian: Separate and guard hot items in data streams. In *Proc. ACM SIGKDD*, 2018.
[38] T. Yang, J. Jiang, P. Liu, Q. Huang, J. Gong, Y. Zhou, R. Miao, X. Li, and S. Uhlig. Elastic sketch: Adaptive and fast network-wide measurements. In *Proc. ACM SIGCOMM*, 2018.
[39] T. Yang, A. X. Liu, M. Shahzad, D. Yang, Q. Fu, G. Xie, and X. Li. A shifting framework for set queries. *IEEE/ACM Transactions on Networking*, 25(5):3116–3131, 2017.
[40] T. Yang, L. Wang, Y. Shen, M. Shahzad, Q. Huang, X. Jiang, K. Tan, and X. Li. Empowering sketches with machine learning for network measurements. In *Proc. ACM SIGCOMM workshop on NetAI*, 2018.
[41] M. Yu, L. Jose, and R. Miao. Software defined traffic measurement with opensketch. In *NSDI*, pages 29–42, 2013.
[42] Y. Zhou, T. Yang, J. Jiang, B. Cui, M. Yu, X. Li, and S. Uhlig. Cold filter: A meta-framework for faster and more accurate stream processing. In *Proc. ACM SIGMOD*, 2018.

APPENDIX

We release source code, datasets, and detailed descriptions of how to replicate our results at GitHub [1]. The source code and detailed descriptions are made public under the MIT license. The datasets are generated from CAIDA trace [5].