DIAL: Distributed Elephant Flow Counting on SDN

Zheng Peng, Hao Li and Chengchen Hu
Department of Computer Science and Technology
Xi'an Jiaotong University
Email: pengzheng@stu.xjtu.edu.cn, {hao.li, chengchenhu}@xjtu.edu.cn

***Abstract*—In network measurement, many statistics counters need maintaining on the switch and thus consuming a lot of very precious on-chip memory. In fact, most network flows in the Internet are small flows that don't need large-width counters. In this paper, we present DIAL, a distributed counting approach, which duplicates the counting rules to leverage global memory resources, maximizing the counting efficiency, which is complementary to existing counter solutions. We pose and formulate the problem of finding the optimal placement for duplicated counting rules. After proving its NP hardness, we give some heuristics to fast generate a near-optimal placement. After describing the feasibility of implementation, we carry out some evaluation for DIAL. Our simulated results with Internet traffic and topologies show that DIAL can significantly decrease the memory cost and increase the memory efficiency for both fixed-width and variable-width counter architecture, with acceptable extra overheads, which is a great save of the precious high-speed memory in the switch.**

I. Introduction

Per-flow counting is very essential to support various measurement scenarios [1]. Specifically, a per-flow measurement task associates each flow with a corresponding counter in the switch to record and update the expected statistics (*e.g.*, packet counts and byte counts) of the flow. The high-level applications periodically query the counters to collect the statistics. To realize the wire-speed counting and respond to frequent querying, the counters are kept in the high-speed memory in the switch, *e.g.*, on-chip registers or SRAM [2], which is very expensive though [3]. In addition, as the Software-Defined Networking (SDN) emerges, the operators tend to measure the fine-grained flows instead of the traditional 5-tuple ones, which requires many more individual counters [4].

Nevertheless, existing counting approaches waste the majority of even this limited memory, making it more insufficient. To be specific, they allocate N-bit memory for each counter to count 2^N packets/bytes, where N is a fixed number. As a result, N should be large enough to measure the "elephant flows" with many packets/bytes. However, since the majority of flows in the traffic are "mice flows" with a few packets/bytes [5], [6], many bits of memory are wasted in most counters. In other words, many fewer flows can be measured in a single switch.

To efficiently count the elephant flows, many attempts have been made [2], [7]–[13]. These approaches embed more information into the counters, so that N-bit memory can measure more than 2^N packets/bytes. That is, the switch can allocate a smaller counter for each flow, so as to measure more flows. However, they still assume fixed-width counters, leaving many bits unused for mice flow counters. Recently, a variable-width architecture, *i.e.*, BRICK, has been proposed [3]. BRICK splits the whole memory into small-sized buckets, and bundles each counter into one bucket initially. If the number of packets/bytes exceeds the counter's capacity, BRICK will allocate an extra bucket for this counter as higher-order bits. In sum, each counter is expected to consume "just enough" memory for the measurement. Although BRICK has efficiently utilized all memory in a single switch, the counting on elephant flows may still fail when the switch runs out of the buckets. This is quite possible, because the high-level measurement applications could count the flows anywhere along the routing path, and such randomness may aggregate the tasks of counting elephant flows into a few switches, resulting in memory exceeding.

Some other works propose the adaptive counting approaches based on SDN architecture [14]–[18]. Besides reactively responding to the queries from the applications, SDN switches can proactively report the statistics to the controller, so as to multiplex the switch counters before exceeding. However, the report messages would incur a heavy load on the south-bound interface of SDN controller, since the counters may exceed very frequently as we discussed above.

We observe that when a few switches are heavy-loaded for elephant flows, lots of memory in other switches remains unused. As a result, our basic idea is to maximumly utilize the memory along the routing path for a specific flow as the supplement to the memory in the local switch. To be specific, all the packets entering the network are tagged as "uncounted". Each switch that has idle memory will count the incoming uncounted packet in the local memory, tag it as "counted", and forward it to the next hop. Otherwise, the switch will leave the packet as uncounted, and directly forward it to the next hop, where more memory can be expected for counting. In this way, the elephant flows will be measured jointly by all the switches along the path, while the mice flows are still maintained by small counters in the single switch. We note that, however, this method duplicates many more counting rules along the routing path, which in contrast burdens the flow table.

Based on the above insights, we propose *DIstributed elephAnt fLow counting (DIAL)*, an enhanced SDN-based counting scheme, to address the inefficient counting problem for elephant flows. DIAL adaptively adds supplementary measurement rules to exploit the idle memory, while minimizing the impact on the flow tables. Note that DIAL focuses on scheduling the global memory resources, and as a result, it is complementary to all existing single switch optimization.

In sum, we make the following contributions in this paper.

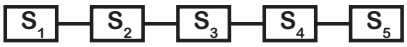

Fig. 1. Example topology.

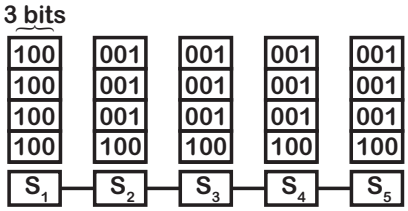

Fig. 2. Example of the counter arrays for the traditional large fixed-width counter solution.

- We present DIAL, a distributed counting approach, which duplicates the counting rules to leverage the global memory resources, maximizing the counting efficiency.
- We pose and formulate the problem of finding the optimal placement for duplicated counting rules. After proving its NP hardness, we give some heuristics to fast generate a near-optimal placement.
- We describe the feasibility of implementation, and carry out some evaluation on DIAL. The results demonstrate that DIAL can significantly decrease the memory cost (up to 93% and 78%) for both fixed-width and variable-width counter architecture, with acceptable extra overheads.

II. Distributed Counting for Elephant Flow

A. Elephant Flow Counting Problem

We use a simple example to explain why counting elephant flows could be a problem. Considering a simple network shown in Fig. 1, where 5 switches S_1–S_5 are linearly connected. We assume there are 20 flows in this network, 4 of which are the elephant flows with 4 bytes at most, and the rest are the mice flows with only 1 byte, following the 80/20 rule. We also assume a simple placement of the counting tasks: each switch counts 4 flows. Next, we specifically consider the following two distributions of the elephant flows. The first distribution evenly places the elephant flows, *e.g.*, 4 of the 5 switches count 1 elephant flow respectively, namely "average case", while the second distribution assumes the 4 elephant flows are gathered in a single switch, namely "dominant case". Based on the above typical cases, we analyze the memory usage of different counting approaches.

The fixed-width approach always assumes the worst case, *i.e.*, each switch will allocate 4 counters for elephant flows, no matter in average or dominant case. As a result, it takes 60 bits in total, since a 4-byte elephant flow takes 3 bits for counting, as shown in Fig. 2. BRICK can largely save the memory in the average case, because it can flexibly extend the 1-bit bucket. As shown in Fig. 3, it consumes 30 bits in total to handle all possible average cases. However, in dominant cases, BRICK also has to allocate 12 bits for each switch, since it cannot predict which switch will be the hot spot, as shown in Fig. 4. In other words, in this simple example, BRICK will consume

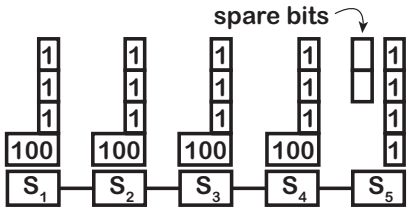

Fig. 3. Example of the average case of the counters for BRICK.

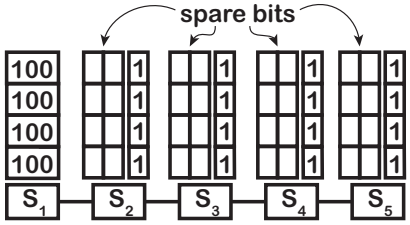

Fig. 4. Example of the dominant case for BRICK.

the same memory with the fixed-width approach, if it wants to avoid exceeding the switch memory in any possible case.

We note that in practice, we have bare knowledge of the flow numbers and sizes. This fact will benefit the actual performance of BRICK, since it can adaptively extend the counter. However, the above analysis also shows that BRICK may still fail the counting if too many elephant flows are counted in a single switch.

B. Basic Idea of DIAL

We propose DIAL based on a simple observation: recall the worst case of BRICK shown in Fig. 4, when the gathered elephant flows impact the memory in a single switch, much memory remains unused in others. Therefore, our basic idea is to exploit the spared memory in other switches as a supplement to the local memory, so that the elephant flows can be jointly counted by multiple switches instead of a single one. Still considering the aforementioned example, we use f_1–f_{20} to denote the 20 flows in the network, where f_1–f_4 are the 4-byte elephant flows. DIAL allocates a set of 1-bit counters for each switch, and when a specific counter is exceeded, DIAL will use a spared counter in other switches to continue counting the flow. Fig. 5 depicts one of the dominant cases, where all elephant flows are originally counted in the first switch. It can be seen that 8 bits for each switch are sufficient. For this worst case, the total memory cost is 40 bits, which is lower than that of the fixed-width approach and BRICK. Plus, DIAL in the average case has a similar cost.

Though the above analysis shows the potential benefit of DIAL, we note that DIAL could bring much overhead if being simply applied. The reason is that to implement the supplementary counters along a routing path, the original counting rule must be duplicated. To be specific, DIAL has to duplicate all the counting rules to all the switches, in case any of them becomes an elephant flow. This duplication makes DIAL impracticable for two reasons. First, the number

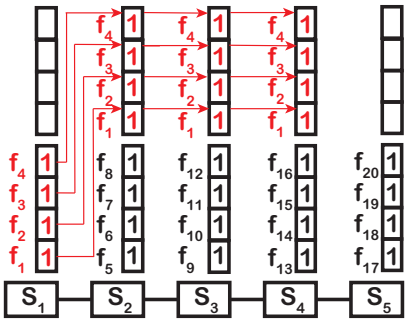

Fig. 5. Example of the dominant case for DIAL.

of counting rules increases with the number of switches, which largely consumes the flow entries in the flow table. For example in Fig. 5, the 20 flows will consume 100 flow entries. Second, the counter memory is occupied right after the counting rules are installed. That is, the counting memory cost will also increase with the number of switches. For example in Fig. 5, even assuming we have the pre-knowledge that there are only 4 elephant flows and that 8 bits for each switch are sufficient, DIAL still has to allocate 100 bits, since we don't know which four could grow to be elephant.

III. Adaptive Counting Rule Duplication

A. Counting Rule Duplication Problem

Recall the 80/20 rule indicating only a small portion of flows are elephant flows, duplicating all the counting rules will waste lots of flow entries. A straightforward method is to adaptively duplicate the counting rules for those flows that cannot be counted in their original switches. However, considering there could be multiple flows requiring extra counting rules simultaneously, the arbitrary duplication may produce the hot spot where the switch runs out of its flow table entries and/or counting memory. That is, we must carefully decide where to duplicate the rule for each flow, namely counting rule duplication problem (CRDP).

We formalize CRDP as follows. Assume there are N switches in total, and M flows requiring extra counting rules. If we duplicate a counting rule for flow M_i to switch N_j, it will consume a_{ij} counting memory along with b_{ij} flow table entries. If M_i cannot be assigned to N_j, *i.e.*, N_j is not in the routing path of M_i, a_{ij} and b_{ij} will be infinite. Note that b_{ij} does not always equal to 1, because the counting rule may have to be further divided to fit into the original flow table. In sum, CRDP can be formalized by the following model:

$$\text{minimize} \quad \sum_{i=1}^{M} a_{ij}x_{ij} \sum_{i=1}^{M} b_{ij}x_{ij} \quad j = 1, ..., N \qquad (1)$$

$$\text{subject to} \quad \sum_{i=1}^{M} a_{ij}x_{ij} \leq A_j, \quad j = 1, ..., N \qquad (2)$$

$$\sum_{i=1}^{M} b_{ij}x_{ij} \leq B_j, \quad j = 1, ..., N \qquad (3)$$

$$\sum_{j=1}^{N} x_{ij} = 1, \quad i = 1, ..., M \qquad (4)$$

Eq. (1) is to minimize the cost of counting memory and flow entries.[1] Eq. (2) and Eq. (3) constrain the counting memory and number of flow entries. Eq. (4) defines a 0-1 variable to represent that each rule can only be duplicated to one switch. Actually, a_{ij} is a constant value for all assignment (M_i to N_j), since DIAL allocates same-size counters for each flow. As a result, we could rewrite the objective to minimize the second factor, *i.e.*, the number of flow entries. In this way, CRDP captures the well-known Multi-Resource Generalized Assignment Problem (MRGAP), which has been proved to be NP-hard [19].

We note that M could be relatively small in practice due to the limited number of elephant flows. However, the first-time "duplication", *i.e.*, the initial counting rule placement, involves all flows in the network, incurring a large overhead of computing optimal solution. As a result, we still need to seek near-optimal solution for CRDP.

B. Heuristics of Near-Optimal Duplication

Due to the NP-hardness of CRDP, we propose some simple heuristics to approximately approach the optimal duplication solution under the acceptable time consumption. Specifically, we consider the following requirements of duplication placement: (1) the consumption of the two resources (counting memory and #flow entries) should not go beyond their capacities, and (2) the consumption of the two resources should be balanced for all switches.

The heuristics is to greedily find the lightest-loaded switch to duplicate the counting rule for each flow that exceeds its original counter, namely Lightest-Loaded-First duplication (LLF). We note that, for a certain flow, the number of candidate switches for duplication is actually the number of hops of this flow. That is, flows with longer path have more options to place the duplicated rules. As a result, for a bunch of flows to be duplicated, LLF will first decide the duplication place for the flows with shorter path.

Algorithm 1 depicts the whole process of finding a near-optimal duplication solution. To be specific, `multi_place` works out the priority of all the duplicating flows by their routing paths' length, and `place` finds the lightest-loaded switch from the candidates for each duplicating flows sequentially.

For example in Fig. 6, each switch counts four flows in the beginning. Suppose that the routing paths of f_1 and f_2 are $S_1 \rightarrow S_2$ and $S_1 \rightarrow S_2 \rightarrow S_3$, respectively. If afterwards both f_1 and f_2 in S_1 need to duplicate their counting rules, `multi_place` will first arrange to `place`(f_1) that has a shorter routing path. For f_1, only S_2 is available, so it duplicates the rule to and creates a 1-bit counter in S_2. Next, `place`(f_2) is called, and here S_3 is lighter loaded than S_2, so the duplicated rule will be placed in S_3.

Pre-duplication. Ideally, the duplication happens only when the original counter exceeds. However, this process has to wait for the controller to issue the new counting rule, which will incur large latency through each duplication. We employ

[1]The weights of the two parts are unknown, so we multiply (not add) them.

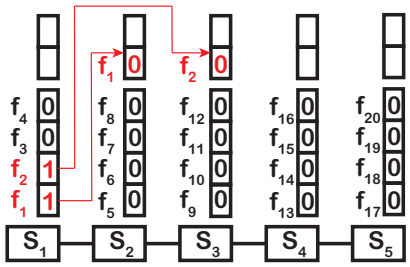

Fig. 6. Example of the duplication for DIAL.

Algorithm 1: Lightest-loaded-first duplication

Input: **Output**: the set of flows to be duplicated: fs

```
1  multi_place(fs) begin
2      sort fs in ascending order of the number of available
       switches along the routing path of each flow;
3      foreach f ∈ fs do
4          place(f);

5  place(f) begin
6      ss ← {s | s is along the routing path of f and there is no
       rule of counting f in s};
7      sort ss in ascending order of the load of each switch;
8      lis ← the lightest-loaded switch of ss;
9      install the rule of counting f in lis;
10     create the counter of f in lis;
```

a pre-duplication process into DIAL to mitigate the latency problem. Specifically, after initially placing the counting rules, we directly duplicate them by re-running `multi_place` of LLF in Algorithm 1. That is, all the counting rules are at least duplicated once. When a flow exceeds its first counter, a third one will be allocated immediately. As a result, the controller has plenty of time to issue the new counting rule, as long as the second counter is not full. In this way, elephant flows are properly counted without employing extra latency.

Fig. 7 depicts a possible result after the pre-duplication, in which the assumptions are the same as those in Fig. 5. After duplicating all the counting rules once, each switch counts eight flows now. It can be seen that f_1 has two counters in S_1 and S_2 at the start, and the one in S_1 is counting the first byte of f_1. When f_1 exceeds its counter in S_1, `overflow_report` in Algorithm 2 will `place`(f_1) in another switch, S_3. Meanwhile, the elephant flow f_1 can be properly and continuously counted in S_2.

IV. Enforce DIAL into SDN

In this section, we highlight some key designs of enforcing DIAL into real SDN environment.

Tag packets. To avoid redundant counting in the switches along the routing path of a specific flow, the first switch that counts the flow should tag the packet as "counted". In OpenFlow, we can tag packets using an unused field of packet, *e.g.*, VXLAN. Such field will be initialized with 0, denoting "uncounted". All the counting rules should only match the uncounted packets. In real time matching, if a packet is

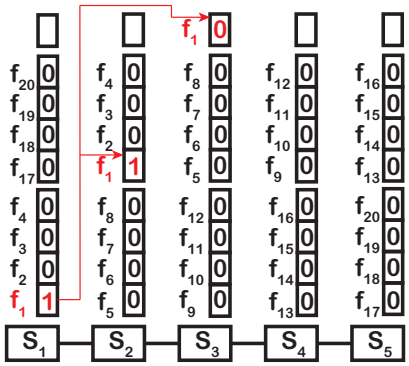

Fig. 7. Example of the pre-duplication for DIAL.

matched with a counting rule, the rule should modify the field to be 1, *i.e.*, the packet has been counted.

Update results. In the measurement system, the controller will collect the counters of all switches periodically. To be specific, at the end of each measurement period, switches will report their counter values to the controller by calling `upload` in Algorithm 2. This can be implemented by Statistics messages in OpenFlow for querying the counters.

Handle exceptions. In real-time counting, if a counter exceeds, `overflow_report` in Algorithm 2 will be triggered to set the switch to be "FULL". If it doesn't exceed, this packet of the flow will remain "uncounted" and pass the switches which contain no rule or have been "FULL" for the flow. If all switches of a flow are full or have installed the counting rule, then `upload` will be triggered to upload the statistics of the flow in all switches to the controller counters and then clear the counter values in the switches to continue counting. During the counting process of DIAL, when a counter exceeds, the duplication happens as `overflow_report` is triggered. In this procedure, an integer is used to help index the switch which incurs the overflowing. Then the flow will set the switch "FULL" and call `place` to find another switch to count it on.

In OpenFlow, `overflow_report` and `upload` launched by switch-end can be implemented with Flow-monitor, Experimenter, and Packet_in messages in special cases.

V. Evaluation

A. Settings

In this section, we evaluate the performance of DIAL. Since DIAL is complementary to the optimization in a single switch, we involve the fixed-width and variable-width approaches in our evaluation. The placement of the counting rules is critical for the above two approaches. We specifically simulate three cases: (1) randomly select the original counting switch for each flow, (2) choose the most empty switch to count the flow (*i.e.*, LLF), and (3) apply DIAL with adaptive rule duplication.

We use three public real-world Internet traffic traces collected in 2016 from CAIDA [20] in our evaluation. Each trace has about 1M flows and the largest flow needs a 28/29-bit width counter. As shown in Table I, each trace has a theoretical optimum (T. O.) memory cost which is calculated by the sum of every theoretical minimal counter width of each flow.

Algorithm 2: Exception handling

Input: **Output**: the flow to be handled: f

```
upload(f) begin
    foreach s along the routing path of f do
        if the rule of counting f has been installed in s then
            upload the counter value of f in s to the controller
            and update the corresponding controller counter;
            reset the counter value of f in s to zero;
            set the state of s along the routing path of f to
            "COUNTING";
overflow_report(f, i) begin
    if a packet leads to an overflow then
        set the state of the i-th switch along the routing path of
        f to "FULL";
        place(f);
```

We employ two real-world network topologies from the Internet Topology Zoo [21], *i.e.*, CERNET (36 switches) and ChinaNet (38 switches). We further involve a simulated topology based on fat-tree ($k = 4$). Given the topologies, we simulate the hosts by evenly partitioning all IPv4 addresses[2] and distributing them to the edge switches. The routing path for each packet class (source and destination IP pair) from the traces is then generated by the shortest-path algorithm.

We consider the following two metrics: (1) the total memory cost, and (2) the memory efficiency. Here the memory efficiency is defined as the ratio of the theoretical optimum to the actual memory cost in total, since all the involved approaches are inevitable to waste some of the memory. We assume the ideal cases for fixed- and variable-width approaches when calculating the total memory cost: for fixed-width approach, we do not set the counter width beforehand, but use the actual consumed memory to calculate the total cost, which avoids the memory exceeding or unnecessary memory waste; for variable-width approach, we assume it has a set of 1-bit buckets without extra overhead (*e.g.*, pointer to link multiple buckets), which could maximize the memory efficiency in a single switch. For DIAL, we assume a threshold of 13-bit width, which means the global counting mechanism will be triggered if the local counter exceeds 13-bit memory. This threshold can be adaptively adjusted for different situations.

B. Results

Fig. 8 depicts the total memory cost of all six approaches. It can be seen that DIAL can largely decrease the memory cost, no matter combined with fixed- or variable-width approach. Specifically, if we apply DIAL to the fixed-width approach, the memory cost will be reduced by 82%–93% and 57%–88%, compared to random and LLF placement, respectively. The variable-width approach has drawn a similar trend by applying DIAL: the memory cost will be reduced by 58%–78% and 7%–54%, respectively.

[2]class A, B and C are included but class D and E are excluded [22]

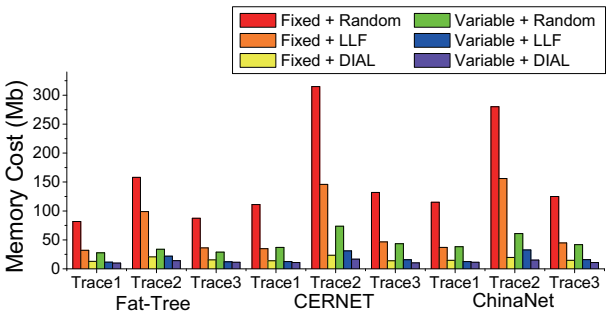

Fig. 8. Memory cost

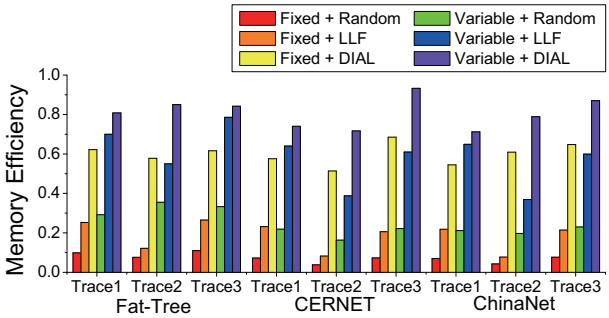

Fig. 9. Memory efficiency

Additionally, the memory efficiency of these six approaches is shown in Fig. 9. It can be observed that DIAL can significantly increase the memory efficiency for both fixed- and variable-width counter approaches. To be specific, when DIAL is applied to the fixed-width approach, the results of memory efficiency reveal that it will bring about 459%–1320% increases if compared to random placement, and 133%–688% rises if compared to LLF placement. The effect of using DIAL is also satisfying with the variable-width counter approach: the memory efficiency will be raised by 139%–338% and 55%–207%, compared to the two placements respectively.

We further measure the overhead of extra Packet_in messages when handling exceeding exceptions, as is denoted in the "Costs" field in Table I. We find that the "Costs" count is at the same order of magnitude with the flow count. We know that in periodic network measurement in SDN, one flow has to lead the switch to communicate with the controller at least once when submitting the result at the end of the measurement period. In DIAL, the extra message cost is introduced, which leads to an increase in the number of the same messages *e.g.*, Packet_in costs in OpenFlow, but that is still at the same order of magnitude, so we consider that it is endurable with high bandwidth SDN link. For example, the system in [23] has the 1Gbps SDN link.

VI. Related Work

Fixed-width counter. In flow statistics field, many approaches have been proposed to save the on-chip memory. DISCO [12] elaborately designs the counter update rule so that the increment of the updated counter is less than the actual packet length. ANLS [13] uses a sampling technique and dynamically sets a smaller sampling rate when the counter value

TABLE I
SOME RESULTS OF DIAL

Topology	Trace	#Flow	T. O.	Costs
Fat-Tree	Trace1	0.9M	8.08Mb	2.0M
Fat-Tree	Trace2	1.5M	12.0Mb	3.5M
Fat-Tree	Trace3	1.0M	9.61Mb	2.7M
CERNET	Trace1	0.9M	8.08Mb	1.9M
CERNET	Trace2	1.5M	12.0Mb	3.4M
CERNET	Trace3	1.0M	9.61Mb	2.2M
ChinaNet	Trace1	0.9M	8.08Mb	1.9M
ChinaNet	Trace2	1.5M	12.0Mb	3.2M
ChinaNet	Trace3	1.0M	9.61Mb	2.3M

is large, to reduce the memory usage. These approaches can save memory cost, but they still apply fixed-width counters, leading to low memory efficiency.

Variable-width counter. To save the memory wasted by fixed counter width, variable-width counter architecture has been proposed. BRICK [3] is the state-of-the-art variable-width counter approach. BRICK splits a counter into several sub-counters, and will allocate an extra sub-counter to a flow only when it exceeds the original memory size. However, in the cases of multiple per-flow tasks, BRICK may fail when handling the hot spots of elephant flows.

DIAL is complementary to these approaches. As a result, a smaller counter/bucket can be set with much lower risk of memory exceeding, so as to further reduce the overall cost. For example, OpenCounter [24] counts unknown flows in SDN, maintaining a counter named LLCRS in the controller for each switch, and aggregates them when queried. The design philosophy of OpenCounter is similar to DIAL. Both of them are distributed counting architecture based on SDN. However, OpenCounter focus on unknown flow counting in controller, with specific data structure LLCRS, whereas DIAL can count any flows in switch, with diverse data structure options, depending on the specific usage, which is a more general counting architecture framework.

VII. CONCLUSION

In this paper, we explored the memory waste problem of per-flow counting. Specifically, we propose a distributed counting approach on SDN, *i.e.*, DIAL. In contrast to only focusing on reducing the local switch memory, DIAL leverages the spared memory along the routing path to make a global optimization. In this work, we formulate the problem of finding the optimal placement for the extra counting rules, and propose simple heuristics to fast generate a near-optimal solution. The evaluation based on real and synthetic traces and topologies has demonstrated that DIAL is effective, efficient and scalable with various local optimization approaches.

ACKNOWLEDGMENT

This work is supported by the National Key Research and Development Program of China (2017YFB0801703), NSFC (61702407, 61672425), and Fundamental Research Funds for the Central Universities.

REFERENCES

[1] B. B. Ran, G. Einziger, R. Friedman, Y. Kassner, B. B. Ran, G. Einziger, R. Friedman, Y. Kassner, B. B. Ran, and G. Einziger, "Randomized admission policy for efficient top-k and frequency estimation," in *IEEE INFOCOM 2017 - IEEE Conference on Computer Communications*, 2017, pp. 1–9.

[2] D. Shah, S. Iyer, B. Prabhakar, and N. Mckeown, "Maintaining statistics counters in router line cards," *IEEE Micro*, vol. 22, no. 1, pp. 76–81, 2002.

[3] N. Hua, B. Lin, J. Xu, and H. Zhao, "Brick:a novel exact active statistics counter architecture," in *ACM/IEEE Symposium on Architecture for Networking and Communications Systems, ANCS 2008, San Jose, California, Usa, November*, 2008, pp. 89–98.

[4] T. P. Xuan and F. Kensuke, "Sdn-mon: Fine-grained traffic monitoring framework in software-defined networks," *Journal of Information Processing*, vol. 58, pp. 182–190, 2017.

[5] Pareto principle. [Online]. Available: https://en.wikipedia.org/wiki/Pareto_principle

[6] L. Guo and I. Matta, "The war between mice and elephants," in *International Conference on Network Protocols*, 2001, pp. 180–188.

[7] S. Ramabhadran and G. Varghese, "Efficient implementation of a statistics counter architecture," in *Acm Sigmetrics International Conference on Measurement & Modeling of Computer Systems*, 2003, pp. 261–271.

[8] M. Roeder and B. Lin, "Maintaining exact statistics counters with a multi-level counter memory," in *Global Telecommunications Conference, 2004. GLOBECOM '04. IEEE*, 2004, pp. 576–581 Vol.2.

[9] Q. Zhao, J. Xu, and Z. Liu, "Design of a novel statistics counter architecture with optimal space and time efficiency." *Acm Sigmetrics Performance Evaluation Review*, vol. 34, no. 1, pp. 323–334, 2006.

[10] A. Cvetkovski, "An algorithm for approximate counting using limited memory resources," *Acm Sigmetrics Performance Evaluation Review*, vol. 35, no. 1, pp. 181–190, 2007.

[11] R. Stanojevic, "Small active counters," *Proceedings - IEEE INFOCOM*, pp. 2153–2161, 2007.

[12] C. Hu, B. Liu, H. Zhao, and K. Chen, "Disco: Memory efficient and accurate flow statistics for network measurement," in *IEEE International Conference on Distributed Computing Systems*, 2010, pp. 665–674.

[13] C. Hu, B. Liu, S. Wang, J. Tian, Y. Cheng, and Y. Chen, "Anls: Adaptive non-linear sampling method for accurate flow size measurement," *IEEE Transactions on Communications*, vol. 60, no. 3, pp. 789–798, 2012.

[14] Y. Zhang, "An adaptive flow counting method for anomaly detection in sdn," in *ACM Conference on Emerging NETWORKING Experiments and Technologies*, 2013, pp. 25–30.

[15] Z. Hu and J. Luo, "Cracking network monitoring in dcns with sdn," in *Computer Communications*, 2015, pp. 199–207.

[16] M. Jarschel, T. Zinner, T. Hohn, and P. Tran-Gia, "On the accuracy of leveraging sdn for passive network measurements," in *Telecommunication Networks and Applications Conference*, 2013, pp. 41–46.

[17] N. L. M. V. Adrichem, C. Doerr, and F. A. Kuipers, "Opennetmon: Network monitoring in openflow software-defined networks," in *Network Operations and Management Symposium*, 2014, pp. 1–8.

[18] J. Suh, T. T. Kwon, C. Dixon, W. Felter, and J. Carter, "Opensample: A low-latency, sampling-based measurement platform for commodity sdn," in *IEEE International Conference on Distributed Computing Systems*, 2014, pp. 228–237.

[19] B. Gavish and H. Pirkul, "Algorithms for the multi-resource generalized assignment problem," *Management Science*, vol. 37, no. 6, pp. 695–713, 1991. [Online]. Available: https://doi.org/10.1287/mnsc.37.6.695

[20] (2016) Caida anonymized internet traces 2016. [Online]. Available: http://www.caida.org/data/passive/passive_2016_dataset.xml

[21] S. Knight, H. Nguyen, N. Falkner, R. Bowden, and M. Roughan, "The internet topology zoo," *Selected Areas in Communications, IEEE Journal on*, vol. 29, no. 9, pp. 1765 –1775, october 2011. [Online]. Available: http://www.topology-zoo.org/

[22] Classful network. [Online]. Available: https://en.wikipedia.org/wiki/Classful_network

[23] M. Liu, L. Luo, J. Nelson, L. Ceze, A. Krishnamurthy, and K. Atreya, "Incbricks: Toward in-network computation with an in-network cache," in *International Conference*, 2017, pp. 795–809.

[24] C. Callegari, S. Giordano, M. Pagano, and G. Procissi, "Opencounter: Counting unknown flows in software defined networks," in *International Symposium on PERFORMANCE Evaluation of Computer and Telecommunication Systems*, 2015, pp. 1–7.