U-HAUL: Efficient State Migration in NFV

Libin Liu, Hong Xu, Zhixiong Niu, Peng Wang, Dongsu Han*
NetX Lab, City University of Hong Kong
*KAIST

ABSTRACT

Network function virtualization (NFV) enables dynamic scaling of resources to middlebox deployment and management. In NFV, state migration is an important task because operators often need to shift traffic and its associated flow states across NF instances for load balancing. Existing state migration schemes, however, exhibit long delays and high controller overhead.

This paper presents U-HAUL, an efficient state migration system that reduces the state migration overhead. U-HAUL takes advantage of the fact that most flows are short-lived mice flows, and in many cases their processing states will expire before the state migration finishes. Rather than blindly moving states of all the flows, U-HAUL keeps the states of active mice flows on the original NF instance, and only migrates elephant flow states. By reducing the number of flow states to be migrated, U-HAUL greatly reduces the migration delay and its performance penalty. Our evaluation shows that U-HAUL reduces the average migration time by up to 87% and the latency to mice flows by up to 94% compared to OpenNF.

1 Introduction

Network function virtualization (NFV) [8] aims to replace hardware middleboxes with virtual software instances running on commodity servers. Unlike layer 3 forwarding, many middleboxes such as firewall, proxy, and VPN perform stateful packet processing. Operators usually need to dynamically redistribute packet processing across multiple VNFs. Consider a load balancing scenario where a firewall instance is overloaded with traffic. An additional VNF then needs to be spawned to adapt to the workload. The operator must not only direct some traffic to the new instance, but also move the internal flow states associated with the traffic. Without state migration, serious problems might arise—for example, attacks may go undetected because the new VNF does not have the necessary information. Thus, efficient state migration is an important and practical issue in NFV.

Frameworks such as Split/Merge [21] and OpenNF [10] automatically transfer states across VNFs with or without guarantees on packet loss, reordering, and state inconsistency. Existing work, however, migrates states of all flows, which takes hundreds of milliseconds to complete, generates lots of overhead in the control plane, and degrades application performance. We find that OpenNF [10] takes more than 100ms to move per-flow states for 1,000 flows, which significantly increases the flow completion time of mice flows. In addition, moving all flows requires the controller to update many entries in the routing tables, causing significant overhead at both the controller and switches with limited flow table size [7].

Modern networks contain many middleboxes that perform variety of advanced network functions [13, 18, 19, 25]. This paper focuses on middleboxes deployed for flows within data centers and intra-data center WAN. We show that in many cases we do not have to move all the active flows and their states. During state migration, their packets have to be buffered at the controller in order to avoid packet loss or reordering at the new instance. Yet, these mice flows are highly likely to finish transmission long before the migration concludes and the controller releases them to the new instance. For many NFs (e.g., IDS, packet filters) per-flow states are no longer useful when the flow ends. Thus, an intuitive idea is that we can just move the states for elephant flows that are more likely to persist after the migration, and keep mice flows at the original instance without any additional delay. By moving much fewer flows and states, one can greatly reduce the controller overhead, migration delay, and performance penalty to applications.

We present U-HAUL, an efficient NFV state migration system that only moves elephant flow states whenever possible. U-HAUL relies on an underlying state management framework such as OpenNF to provide basic state migration services. It adds three main components: a lightweight elephant detection module (EDM), a new northbound API call for applications to enable elephant-only migration, and a filter API that allows the controller to obtain elephant flow information from EDM.

APSys 2016, August 4-5, 2016, Hong Kong, China.

 ISBN 978-1-4503-4265-0/16/08…$15.00

DOI: http://dx.doi.org/10.1145/2967360.2967363

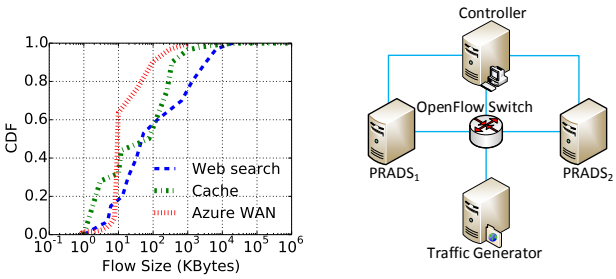
Given a definition of an elephant flow (e.g., size > 3MB), the operator calculates the fraction of elephant traffic based on its offline profiling and measurement, and uses it as the target threshold value. The control application then initiates elephant-only migration. The controller communicates with the EDM using the filter API to detect elephant flows. Information about the selected elephant flows is then sent to the controller as OpenNF *filters*. These filters are then passed to existing OpenNF methods for state migration. We conduct testbed experiments on Emulab to evaluate U-HAUL using realistic traffic distributions. Our preliminary results show that compared to OpenNF, U-HAUL reduces the average migration time by up to 87%, the latency overhead to mice flows by up to 94%, and the buffer usage at the controller by up to 60%.

2 Motivation

We first present our motivation to consider just moving states for elephant flows in NFV state migration. Figure 1(a) shows the flow size distribution in three production networks: a web search cluster [5], a Facebook's cache cluster [23], and the Azure WAN [12]. The first two are data center networks and the third is an inter-data center WAN. For all networks, ∼60% of the flows are less than 100 KB. For Facebook cache cluster and Azure WAN, more than 90% of the flows are less than 1 MB. Mice flows are latency-sensitive and can finish within tens to hundreds of microseconds in 40G or 10G networks.

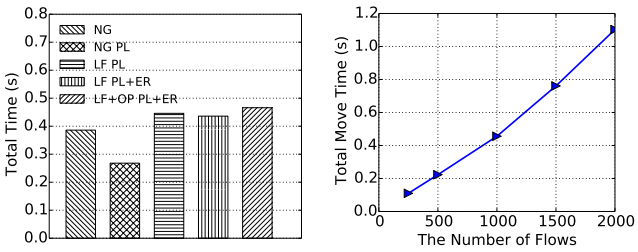
Our observation is that state migration incurs downtime on the order of $O(100)$ms [10, 17], which is significantly larger than the flow completion time of mice flows. This is because migrating states is a control-plane action where complex mechanisms have to be in place to provide critical performance guarantees such as loss-free and order-preserving for NFV applications. To see this, we deploy OpenNF [10], state-of-the-art state migration system on a five-node testbed in Emulab as in Figure 1(b) (more details of the testbed in §4.1). We use five physical machines instead of VMs for maximum performance. We use two PRADS asset monitor instances [3] ($PRADS_1$ and $PRADS_2$) that are OpenNF-enabled. Initially all traffic is sent to $PRADS_1$. After it has created per-flow states for 2,000 flows, we move half of the flows and their states to $PRADS_2$ for load balancing. Figure 2(a) shows that even without any performance guarantees, OpenNF takes at least 268ms to transfer 1,000 states. When applications demand loss-free and/or order-preserving guarantees, the migration time is beyond 400ms even with optimizations.

This motivates us to make a case for U-HAUL that moves just the elephant flow states during NFV state migration. We argue that one can simply leave active mice flows at the original VNF $PRADS_1$ because they will finish the transmission long before state migration ends. This does not harm the working of many NFs, such as NATs, IDS, and packet filters that only require per-flow states. We do not focus on the consistency issues that arise when global or multi-flow state is needed on the new NF instance, which is addressed in some existing work [10]. U-HAUL only migrates per-flow state.



(a) **(b)**

Figure 1: (a) Flow size distributions in a web search cluster [5], a Facebook's cache cluster [23], and Azure WAN [12]; (b) U-HAUL Testbed Topology.



(a) **(b)**

Figure 2: (a) Migration time with no guarantees (NG), loss-free (LF), and loss-free and order-preserving (LF+OP) with and without parallelizing (PL) and early-release (ER) optimizations, using flow size distribution of Figure 1(a); (b) Migration time when moving different numbers of flows with loss-free and order-preserving guarantees, and parallelizing and early-release optimizations in OpenNF.

The benefits of moving just the elephant flow states is twofold. First, it significantly reduces the migration downtime. We repeat the OpenNF migration experiments with varying number of states and find that migration downtime increases with number of states as shown in Figure 2(b).[1] Thus, migrating fewer states can greatly cut the downtime and eliminate the extra latency to mice flows. Second, it also helps minimize migration overhead at both the controller and Openflow switches. The controller has fewer flows to buffer packets and send messages for. Additionally, fewer forwarding rules need to be updated at Openflow switches to adjust routing of flows, further streamlining the entire migration process with flows and states.

Note, in some cases it is necessary to move states for all flows, for example when the VNF crashes. U-HAUL does not apply in these cases.

3 Design and Implementation

Figure 3 shows the overall design of U-HAUL based on OpenNF. A controller manages both the NF state and flow routing at switches. U-HAUL consists of three components: an elephant detection module (EDM) at each VNF, a new northbound API call between applications and the controller,

[1]This is likely due to the increased complexity of providing loss-free and order-preserving guarantees with more packets.

and a new filter API between the controller and the EDM to obtain elephant flow information. In general these components work as follows: When the control application initiates state migration using the U-HAUL API call, the elephant detection module selects flows to be migrated out of all active flows and transfers their information to controller via the filter API. The controller then invokes the OpenNF methods to iteratively move the states of these to the destination VNF.

Note that during state migration, the original NF instance stops establishing new states for flows it has not seen yet. All packets that hit the original instance but do not have any matching state are forwarded to the new instance which processes them normally. For example for a persistent TCP connection that does not always have traffic, U-HAUL would not regard it as new flow, which guarantees the correctness of NFs.

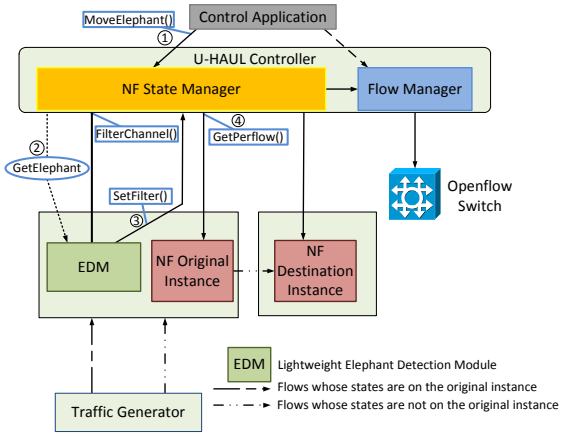


Figure 3: U-HAUL Overview

3.1 Elephant Detection

We describe the design of the elephant detection module (EDM) in this section.

Flow Monitor: Elephant detection relies on a *flow monitor* to collect information of active flows at each VNF. Here we consider active to mean existing within the last 100ms time window. The monitor maintains a *flow information table* (FIT), where each entry stores the flow ID hashed from five-tuples, its current size, and a timestamp of its last packet.

There are three operations associated with the FIT: insertion, update, and eviction.

Insertion: When a packet arrives at the VNF, if the flow is not present in the FIT, a new entry with the flow ID and timestamp is added.

Update: When a packet arrives and its flow is present in the table, the size and timestamp of the flow is updated.

Eviction: Every 100ms, an eviction pointer goes through the FIT to evict entries whose timestamps are not within the last time window. This helps maintain the FIT size in a reasonable level as we will show in §4.4. We have tried 20ms, 50ms, 100ms as time window values to evict flow entries. We choose 100ms because it is able to cover all active flows that indeed cause overload for NF instance.

The flow monitor adds delay to the VNF's packet processing. This overhead can be minimized by implementing it using high-performance packet I/O frameworks such as DPDK [1] or netmap [22] with microsecond level delay [11].

Traffic Proportion based Detection: Elephant flow detection is usually used to optimize per-flow routing in data center networks, such as in Hedera [4] and Mahout [6]. These systems use a size-based approach: a flow is classified as an elephant when it has sent more than a certain amount of bytes.

However, in U-HAUL, we need to tell whether an active flow will be an elephant or not based on its current size at the time of migration, and the size-based approach cannot detect elephant flows that have not sent enough bytes. Thus, we adopt a detection approach based on proportion of traffic instead. First, the operator monitors the traffic of its network and calculates the proportion of traffic ρ that comes from elephant flows. This is the target threshold value configured at each EDM. At the time of migration, the EDM queries its FIT for all active flows and obtains the total current size S. It then selects flows one by one based on their current size in the decreasing order and calculates the total size of selected flows S', until the proportion of traffic from the selected flows S'/S is larger than or equal to ρ.

Effectively our approach relies on relative size of a flow, making it less prone to false negatives from using the absolute size. If we use 3MB as the elephant definition, a flow that has sent 2.5MB is likely to be detected in our approach, whereas the size-based approach will miss it.

3.2 A New Northbound API Call

U-HAUL provides a new northbound API call `moveElephant` for applications to enable elephant-only state migration. It transfers both the state and input (i.e., traffic) for a set of flows from one VNF to another. Its syntax is:

```
moveElephant(src, dst, scope, properties).
```

The implementation extends OpenNF's `move` method. The `scope` argument specifies which class(es) of state (per-flow and/or multi-flow) to move. We only consider per-flow in this paper. The `properties` argument defines whether the move should be loss-free and order-preserving [10].

When the control application issues `moveElephant`, the controller communicates with the elephant detection module (EDM) in order to determine which flows are elephant flows. This is done through the filter API described below.

Filter API: The filter API between the controller and the EDM consists of `FilterChannel` and `SetFilter` functions. When the controller receives `moveElephant` call, it invokes the `FilterChannel` function to send a `getElephant` message to the EDM. The EDM runs its detection logic as discussed in §3.1. It then uses the `SetFilter` function to package the five-tuples of selected flows and send them to the controller to match the *filter*, which is a dictionary specifying values for five-tuples in OpenNF.

The controller uses the matched filters to configure the `moveElephant` operation. For each filter, it invokes the OpenNF `getPerflow` function and pass the filter as the input

to get the per-flow state pertaining to the flow. For a move without guarantees, the controller calls `putPerflow` on the destination VNF instance and `delPerflow` on the original VNF instance to complete per-flow state transfer.

4 Evaluation

We now evaluate the performance of U-HAUL using an Emulab testbed [2]. We answer the following questions:

(1) How well does U-HAUL's elephant detection work?

(2) How much performance benefit can U-HAUL provide compared to OpenNF?

(3) How much overhead does U-HAUL add?

(4) Can the remaining flows actually die out in time when migration finishes?

4.1 Testbed and Setup

Our testbed consists of five servers arranged as shown in Figure 1(b). Each server has two 2.4GHz Intel Xeon 8-Core E5-2630 v3 processors, 64GB DDR4 RAM, and a quad-port Intel X710 10GbE NIC. Two servers run two OpenNF-modified PRADS [3] VNFs separately. One server runs OpenvSwitch as an OpenFlow switch. Another one runs the U-HAUL controller while the fifth server generates traffic. We use the web search workload [5], the Facebook workload [23], and the Azure WAN workload [12] shown in Figure 1(a) to generate traffic: flows arrive and finish dynamically according to a Poisson process with an average load of 1, i.e., 10Gbps.

Methodology: We generate five traces of flows using the three workloads, send flows according to the traces, and choose a random time after 2 seconds into the traces to start migration. The same traces are used for the different schemes compared. Thus for each scenario we repeat experiments for five independent runs. We report the average result of the five runs. We consider elephant flow definitions from 3MB to 10MB for web search workload and Facebook workload, and from 1MB to 2.5MB for Azure WAN workload. We use the corresponding threshold values on proportion of elephant traffic for our EDM.

Schemes Compared: We compare U-HAUL against OpenNF that moves all flows. We choose OpenNF because only it provides loss-free and order-preserving guarantees compared with other solutions mentioned in §5. These guarantees are important to ensure the new NF instances work correctly. U-HAUL also provides these guarantees. In our experiments, we set the guarantee to be loss-free, the scope to be per-flow state, and the optimizations to be parallelizing and early-release for both schemes. This is consistent with the setup in [10].

4.2 Accuracy of Elephant Detection

We first investigate the accuracy of our elephant detection module. We do so by comparing the detection output of our EDM with the trace files that record the actual size of flows.

Figures 4, 5 and 6 show the accuracy results. The average number of active flows is 142.3 in the web search workload, 364.0 in the Facebook workload, and 955.4 in the Microsoft Azure workload. We observe that our method based on

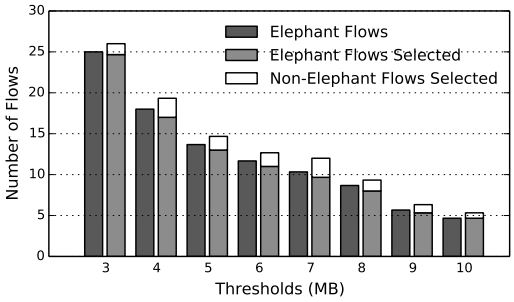


Figure 4: Accuracy of elephant detection using web search workload.

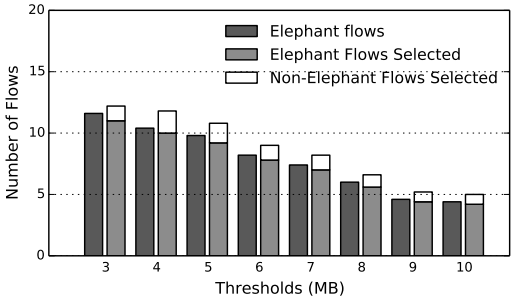


Figure 5: Accuracy of elephant detection using Facebook workload.

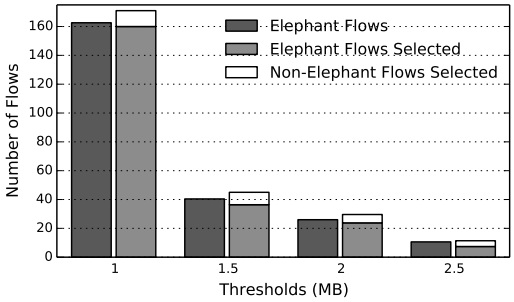


Figure 6: Accuracy of elephant detection using Azure WAN workload.

proportion of elephant traffic performs well for different definitions of elephant flows. For web search workload, the average accuracy of different definitions of elephant flows is 95.0%; the minimum is 91.9%. For Facebook workload, the average accuracy is 94.9%; the minimum is 93.3%. For Azure WAN workload, the average accuracy is 87.5%; the minimum is 69.8%. Tables 1, 2 and 3 show the false positive and false negative rates, which further demonstrates the effectiveness of our method.

Besides, Tables 1, 2 and 3 show that different workloads have different false positive and false negative rates. The Azure WAN workload has higher false positive and false negative rates. This is because different workloads have different flow size distributions, which is shown in Figure 1(a). Comparing the flow size distributions with the results of Tables 1, 2 and 3, we can see the trend is that the larger the proportion of mice flows is, the larger the false positive and false negative rates are. Taking the Azure WAN workload as an example, we can see more than 60% flows are smaller than 10KB. If we choose 2.5MB as the elephant threshold, the

number of elephant flows is very small. Besides, most non-elephant flows may be smaller than 10KB. This means that our elephant detection approach chooses more non-elephant flows to meet the proportion threshold. This results in larger false positive and false negative rates.

Definition (MB)	3	4	5	6	7	8	9	10
False Positive Rate	1.2%	1.9%	1.2%	1.2%	1.4%	1.1%	0.7%	0.6%
False Negative Rate	1.6%	5.6%	4.4%	5.2%	7.7%	7.0%	8.1%	0

Table 1: False positive and false negative rates for the web search workload.

Definition (MB)	3	4	5	6	7	8	9	10
False Positive Rate	0.6%	0.5%	0.5%	0.3%	0.3%	0.3%	0.2%	0.2%
False Negative Rate	5.2%	3.8%	6.1%	4.9%	5.4%	6.7%	4.3%	4.5%

Table 2: False positive and false negative rates for the Facebook workload.

Definition (MB)	1	1.5	2	2.5
False Positive Rate	6.4%	19.1%	19.6%	35.0%
False Negative Rate	1.2%	9.9%	8.4%	30.1%

Table 3: False positive and false negative rates for the Azure WAN workload.

We also compare our approach with the size-based approach. Tables 4 and 5 show the false negative rate of size-based detection for the Facebook workload and Azure WAN workload. Result of the web search workload is similar with the Facebook workload and omitted here. We observe that the size-based approach has a higher false negative rate compared to our approach. This is expected because a size-based approach easily misses elephant flows that do not meet the threshold at the time of migration.

4.3 Downtime, Buffer Usage, and FCT

We now evaluate the performance benefit of U-HAUL compared to OpenNF along three dimensions: migration downtime, buffer usage for migration at the controller, and flow completion time (FCT) reduction to mice flows. Figures 7, 8 and 9 show the downtime and buffer usage results. Downtime here is defined as time elapsed between the beginning of the `moveElephant` call and the finish time of the last `delPerflow` call. Buffer usage refers to the overall memory used to buffer packets that arrive during the state transfer. We observe that U-HAUL provides significant performance benefits. For example for web search workload, U-HAUL reduces the downtime by at least 81.8% and up to 90.7% compared to OpenNF. It saves at least 59.3% buffer usage, and the saving can be up to 81.9%. The observation demonstrates U-HAUL can manage state migration very efficiently.

We also look at the FCT reduction to the mice flows provided by U-HAUL. FCT reduction is defined as the ratio between the mice flow's FCT in OpenNF and its FCT in U-HAUL. In OpenNF, the FCT includes the migration downtime as OpenNF moves all flows, whereas in U-HAUL the mice flows' FCT is not affected by the migration at all. We calculate FCT reduction for all mice flows excluded in U-HAUL's state migration. Tables 6 and 7 show the results. The mice flow FCT in OpenNF is at least 4.6 times that of U-HAUL for web search load, 18.3 times for Facebook workload, and 43.0 times for Azure WAN workload. This means U-HAUL reduces extra latency to mice flows by at least 78.3% for web search workload, 94.5% for Facebook cache workload, and 97.6% for Azure WAN workload.

Definition (MB)	3	4	5	6	7	8	9	10
False Negative Rate	12.5%	13.0%	18.2%	10.0%	10.0%	13.3%	14.3%	15.4%

Table 4: False negative rate of the size-based elephant detection for the Facebook workload.

Definition (MB)	1	1.5	2	2.5
False Negative Rate	12.0%	20.1%	13.2%	55.6%

Table 5: False negative rate of the size-based elephant detection for the Azure WAN workload.

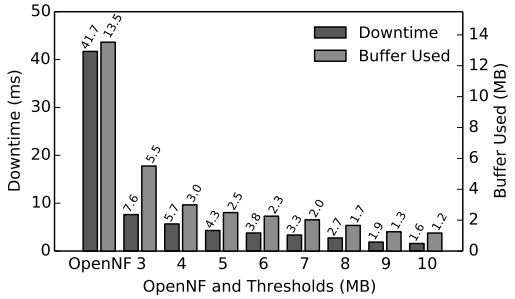


Figure 7: Downtime and buffer usage comparison in web search workload.

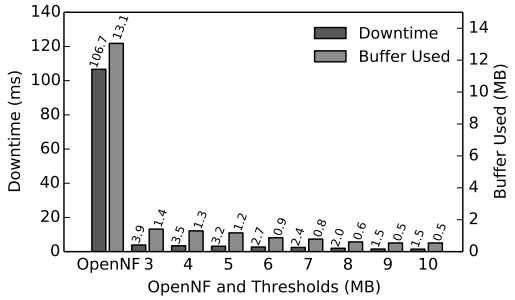


Figure 8: Downtime and buffer usage comparison in Facebook workload.

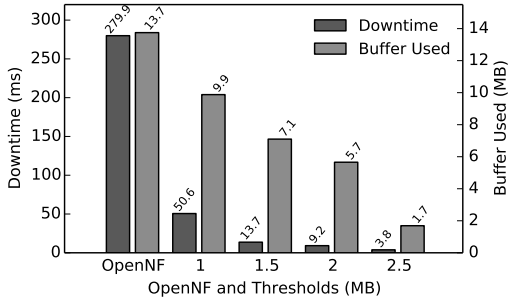


Figure 9: Downtime and buffer usage comparison in Azure WAN workload.

Definition (MB)	3	4	5	6	7	8	9	10
Web search	6.7	4.6	4.6	4.6	4.6	4.6	4.8	4.8
Facebook	20.9	20.9	20.9	18.3	18.3	18.3	18.3	18.3

Table 6: The mice flow FCT reduction of U-HAUL compared to OpenNF in Web search and Facebook workload.

Definition (MB)	1	1.5	2	2.5
Azure WAN	43.0	47.6	47.0	45.0

Table 7: The mice flow FCT reduction of U-HAUL compared to OpenNF in Azure WAN workload.

4.4 Overhead

We now evaluate the overhead of U-HAUL. We first consider the overhead of using the FIT in EDM. Table 8 shows the number of entries in the FIT at each 100ms-eviction epoch for one run of our experiment using the Facebook workload and the Azure WAN workload. Results of web search workload, Facebook workload, and the Azure WAN workload for other runs are qualitatively similar. It indicates that the overhead of FIT is bounded in a reasonable level.

Time (x100ms)	1	2	3	4	5	6	7	8	9	10
Number of Entries (Facebook)	403	746	774	753	731	679	729	746	743	788
Number of Entries (Azure WAN)	624	1454	1576	1503	1529	1478	1579	1497	1511	1517

Table 8: The number of FIT entries in one run of the experiment using the Facebook workload and the Azure WAN workload.

Next, we consider the number of "missed" flows that persist after migration finishes, and therefore should have been included in the migration. A missed flow may be an elephant not detected by the EDM, which results in a false negative. More importantly it can also be a flow with size very close to the elephant threshold that is not detected by the EDM, which may be more likely in practice. Ideally the number of missed flows should be zero, meaning all of the flows left at the original VNF should finish when the migration of big flows completes. Yet this is difficult to achieve simply because predicting the FCT of a flow especially at the beginning of its transmission is challenging.

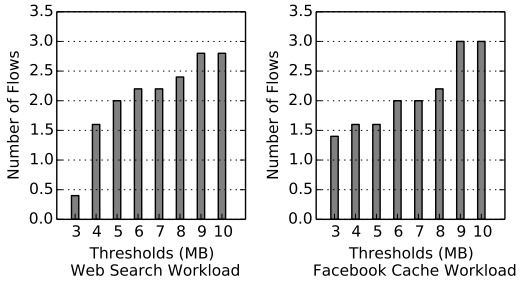


Figure 10: The number of missed flows in the Web search workload and Facebook workload. The results are the average of 5 runs.

Definition (MB)	1	1.5	2	2.5
Number of missed flows	0	0	0.4	2.4

Table 9: The number of missed flows in the Azure WAN workload. The results are the average of 5 runs.

Now, we evaluate the number of missed flows by comparing U-HAUL downtime with flows start times and FCTs from the trace files. Figure 10 and Table 9 show the results. Only a few missed flows exist in our experiments out of a total of 142.3 active flows in the web search workload, 364.0 in the Facebook workload, and 955.4 in the Azure WAN workload as mentioned in §4.2. Another phenomenon is that a larger elephant flow definition causes more missed flows in general. This is intuitive: As we move to consider bigger elephant flows, fewer states need to be moved and the migration downtime is also shorter, making it more likely to have missed flows.

4.5 Die-out Time After Migration

Definition (MB)	3	4	5	6	7	8	9	10
Web search (ms)	0.095	7.611	8.980	9.501	9.926	10.540	11.403	11.696
Facebook (ms)	9.281	9.623	9.818	15.820	16.245	16.746	17.155	17.251

Table 10: The die-out time of missed flows in the Web search workload and Facebook workload.

Definition (MB)	1	1.5	2	2.5
Azure WAN (ms)	0	0	0.350	1.445

Table 11: The die-out time of missed flows in the Azure WAN workload.

Definition (MB)	3	4	5	6	7	8	9	10
Web search (ms)	7.694	13.311	13.280	13.301	13.226	13.240	13.303	13.296
Facebook (ms)	13.181	13.123	13.018	18.520	18.645	18.746	18.655	18.751

Table 12: The total time of U-HAUL in the Web search workload and Facebook workload.

Definition (MB)	1	1.5	2	2.5
Azure WAN (ms)	50.603	13.685	9.542	5.266

Table 13: The total time of U-HAUL in the Azure WAN workload.

We now evaluate the *die-out time* of missed flows, defined as time elapsed between the completion of states transfer and the termination time of the last missed flow. Die-out time reflects how long the original instance has to wait till all missed flows die out. Tables 10 and 11 show the results. The minimum die-out time for web search workload and Facebook workload appears when elephant flow threshold is 3MB, and for Azure WAN workload 1MB. We can see the die-out time increases as elephant flow threshold increases, a trend similar to the number of missed flows.

We also observe that there is an interesting tradeoff between die-out time and downtime shown in §4.3. With a larger elephant flow definition, the downtime decreases but the die-out time increases, because we move fewer elephant flows for state migration, and more flows are left at the original instance until they die out. This implies that it is not possible to minimize both metrics at the same time. As the extreme examples, one can minimize downtime by not migrating any flows, which results in the maximum die-out time; one can also minimize die-out time by migrating many elephant flows, which then yields long downtime. Neither are desirable for operators. Thus we consider the *total time* of U-HAUL, defined as downtime plus the die-out time of missed flows, to assess this tradeoff. Total time can be understood as the time of the entire migration process. It is different from downtime in §4.3, which is the actual time during which active flows are buffered and affected.

Tables 12 and 13 show the total time results. We observe that there is a sweet spot for the tradeoff depending on the flow size distribution: For web search workload, 3MB is the best elephant flow threshold that gives the shortest total time; for Facebook cache workload the sweet spot is 5MB, and for Azure WAN workload 1.5MB. Note that for Azure WAN workload, when the elephant flow thresholds are 2.5MB, 2MB, 1.5MB, and 1MB, respectively, the associated proportions of traffic are 20%, 32%, 66%, and 78%. It is more reasonable to choose 1.5MB for load balancing. Under this setting, U-HAUL migrates a significant proportion of traffic

from the original instance for load balancing to reduce the die-out time with short downtime as well.

5 Related Work

Many solutions exist for NFV state management. Split/Merge [21], Pico Replication [20], OpenNF [9, 10], and DiST [15] are systems that provide some control over both internal NF state and network state. Split/Merge and Pico Replication provide shared libraries that NFs use to create, access, and modify internal state through pre-defined APIs. OpenNF provides a northbound API for applications to specify which state to move, and which guarantees to enforce; it also implements a southbound API for the controller to perform the export or import of NF state. DiST differs from OpenNF by buffering the packets during migration at the destination VNF instead of at the controller. StatelessNF [14] re-architects network functions so that their internal state is maintained in a shared separate storage tier, which have to face the challenge that states update frequently. FTMB [24] focuses on reconstructing lost state when a software middlebox fails. Finally, Olteanu and Raiciu [16] attempt to migrate per-flow state between VM replicas without application modifications.

These solutions all provide state migration which moves state of all flows. The key difference between U-HAUL and existing work is that we distinguish elephant flows from mice, only migrate their states for load balancing scenario, and develop an efficient elephant detection method.

6 Conclusion

We introduced U-HAUL, an efficient state migration system in NFV. U-HAUL only migrates per-flow state for elephant flows and maintains state for mice flows in the original NF instance until they expire. We implemented U-HAUL based on OpenNF and evaluated it on an Emulab testbed. Our preliminary results show that U-HAUL significantly reduces the migration downtime and the performance penalty.

For future work, we intend to improve U-HAUL using more accurate elephant detection with less overhead. We also plan to optimize its implementation, and conduct more comprehensive evaluation with more NFs.

7 Acknowledgements

We thank the anonymous reviewers and our shepherd Florin Dinu for their valuable comments on this paper. This work was supported by the Research Grants Council, University Grants Committee of Hong Kong (awards ECS-21201714, GRF-11202315, and CRF-C7036-15G), the ICT R&D program of MSIP/IITP of Korea [B0101-16-1368], and National Research Foundation of Korea funded by MSIP (NRF-2013R1A1A1076024).

8 References

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

[2] Emulab. http://www.emulab.net/.

[3] Passive Real-time Asset Detection System. http://prads.projects.linpro.no.

[4] M. Al-Fares, S. Radhakrishnan, B. Raghavan, N. Huang, and A. Vahdat. Hedera: Dynamic Flow Scheduling for Data Center Networks. In *Proc. USENIX NSDI*, 2010.

[5] M. Alizadeh, A. Greenberg, D. A. Maltz, J. Padhye, P. Patel, B. Prabhakar, S. Sengupta, and M. Sridharan. Data Center TCP (DCTCP). In *Proc. ACM SIGCOMM*, 2010.

[6] A. R. Curtis, W. Kim, and P. Yalagandula. Mahout: Low-Overhead Datacenter Traffic Management using End-Host-Based Elephant Detection. In *Proc. IEEE INFOCOM*, 2011.

[7] A. R. Curtis, J. C. Mogul, J. Tourrilhes, P. Yalagandula, P. Sharma, and S. Banerjee. DevoFlow: Scaling Flow Management for High-Performance Networks. In *Proc. ACM SIGCOMM*, 2011.

[8] European Telecommunications Standards Institute. Network Functions Virtualisation: Introductory White Paper. http://portal.etsi.org/NFV/NFV_White_Paper.pdf.

[9] A. Gember-Jacobson and A. Akella. Improving the Safety, Scalability, and Efficiency of Network Function State Transfers. In *Proc. ACM HotMiddlebox*, 2015.

[10] A. Gember-Jacobson, R. Viswanathan, C. Prakash, R. Grandl, J. Khalid, S. Das, and A. Akella. OpenNF: Enabling Innovation in Network Function Control. In *Proc. ACM SIGCOMM*, 2014.

[11] S. Han, K. Jang, A. Panda, S. Palkar, D. Han, and S. Ratnasamy. SoftNIC: A Software NIC to Augment Hardware. Technical report, UC Berkeley, 2015.

[12] K. He, A. Fisher, L. Wang, A. Gember, A. Akella, and T. Ristenpart. Next Stop, the Cloud: Understanding Modern Web Service Deployment in EC2 and Azure. In *Proc. ACM IMC*, 2013.

[13] D. A. Joseph, A. Tavakoli, and I. Stoica. A Policy-aware Switching Layer for Data Centers. In *Proc. ACM SIGCOMM*, 2008.

[14] M. Kablan, B. Caldwell, R. Han, H. Jamjoom, and E. Keller. Stateless Network Functions. In *Proc. ACM HotMiddlebox*, 2015.

[15] B. Kothandaraman, M. Du, and P. Sköldström. Centrally Controlled Distributed VNF State Management. In *Proc. ACM HotMiddlebox*, 2015.

[16] V. A. Olteanu and C. Raiciu. Efficiently Migrating Stateful Middleboxes. In *Proc. ACM SIGCOMM*, 2012.

[17] S. PALKAR, C. Lan, S. Han, K. Jang, A. Panda, and S. Ratnasamy. E2: A Framework for NFV Applications. In *Proc. ACM SOSP*, 2015.

[18] R. Potharaju and N. Jain. Demystifying the Dark Side of the Middle: A Field Study of Middlebox Failures in Datacenters. In *Proc. ACM IMC*, 2013.

[19] Z. A. Qazi, C.-C. Tu, L. Chiang, R. Miao, V. Sekar, and M. Yu. SIMPLE-fying Middlebox Policy Enforcement Using SDN. In *Proc. ACM SIGCOMM*, 2013.

[20] S. Rajagopalan, D. Williams, and H. Jamjoom. Pico Replication: A High Availability Framework for Middleboxes. In *Proc. ACM SoCC*, 2013.

[21] S. Rajagopalan, D. Williams, H. Jamjoom, and A. Warfield. Split/Merge: System Support for Elastic Execution in Virtual Middleboxes. In *Proc. USENIX NSDI*, 2013.

[22] L. Rizzo. netmap: A Novel Framework for Fast Packet I/O. In *Proc. USENIX ATC*, 2012.

[23] A. Roy, H. Zeng, J. Bagga, G. Porter, and A. C. Snoeren. Inside the Social Network's (Datacenter) Network. In *Proc. ACM SIGCOMM*, 2015.

[24] J. Sherry, P. X. Gao, S. Basu, A. Panda, A. Krishnamurthy, C. Maciocco, M. Manesh, J. a. Martins, S. Ratnasamy, L. Rizzo, and S. Shenker. Rollback-Recovery for Middleboxes. In *Proc. ACM SIGCOMM*, 2015.

[25] J. Sherry, S. Hasan, C. Scott, A. Krishnamurthy, S. Ratnasamy, and V. Sekar. Making Middleboxes Someone Else's Problem: Network Processing as a Cloud Service. In *Proc. ACM SIGCOMM*, 2012.