

A Collision-Mitigation Hashing Scheme Utilizing Empty Slots of Cuckoo Hash Table

ChoongHee Cho*, JungBok Lee**, and Jeong-dong Ryoo*

* Information and Communication Network Technology, Korea University of Science and Technology (UST), Daejeon 34113, Korea

**Kakao Corp, Sungnam 13494, Korea

ww3w3@etri.re.kr, andrew.l@kakaocorp.com, ryoo@etri.re.kr

Abstract— In SDN and NFV technologies, performance of a virtual switch is important to provide network functionalities swiftly. Since the lookup operation of the virtual switch has been considered a major bottleneck of performance, we need to devise an efficient way to reduce the lookup time. To improve the lookup speed, previous research has suggested a compact lookup table in a fast memory, but the issue of collision in a hash table has not been addressed well enough. This paper proposes a new scheme that mitigates collisions by utilizing empty slots of the hash table. We propose to insert a new element that may collide to an empty slot instead of linking it with a pointer. According to an evaluation on our experiments, about 20% of elements that could have experienced collisions in the conventional scheme are inserted into empty slots in each bucket. Moreover, the access time of collided elements has halved by avoiding unnecessary memory accesses.

Keywords— Routing table lookup, Virtual switch, OpenFlow, Cuckoo hash table, Collision-mitigation.

I. INTRODUCTION

In recent years, Software-Defined Networking (SDN) and Network Functions Virtualization (NFV) technologies have been developed rapidly not only within academia but also within commercial network environment. By using these technologies, we can make a more flexible and reconfigurable network with a switch supporting OpenFlow protocol [1]. A switch such as Open vSwitch (OVS) [2] is a key component to meet requirement of network performance.

In particular, the lookup operation is closely related to the performance of the switch. To improve its performance, we should carefully consider various factors such as the placing of elements in memory, the way to access memory, the access speed of memory, etc. For fast lookup processing, a hash table and its algorithm are usually implemented in most switches. Using a hash function, we can directly access the location of elements. As is well known, however, a hash table wastes memory in terms of space utilization. Moreover, hash collisions are inevitable. Since OVS version 2.4.0 has been released, cuckoo hashing [3] and separate chaining [4] schemes are adopted to resolve those issues. The cuckoo hashing is a hash scheme which uses two hash tables and two hash functions. By using cuckoo hashing, memory slots can be

utilized up to 95% [5] and slots are fast accessible in constant time. If certain elements are placed at the same location (i.e., bucket) according to the hash function, we can distinguish the elements linked with a pointer since each element has an original key-value set. This hash-related data structure with a linked-list is known as separate chaining scheme. The more elements we store in limited memory space, the more collisions will occur. Due to the additional memory accesses to follow the address in each pointer, collisions degrade the lookup performance.

To improve lookup speed, previous research (e.g. [6], [7]) makes a compact lookup table to load the table into the fast memory. In this paper, we propose a new scheme that mitigates collisions (up to 20%) and improves the lookup speed by utilizing existing empty slots of cuckoo hash table. Our idea is that the empty slots which exist inevitably in the hash table are used for collision elements. Although empty slots are reserved for the elements to be inserted, one of the collision elements can be stored temporarily at the empty slot until the other element is inserted into the empty slot. For this new approach, we define the new *Collision Hash Function* and *Collision Flag Bits* to distinguish the elements that have collided in the same space.

This paper is organized as follows. Section II describes the existing scheme of cuckoo hashing in OVS as a background. Section III introduces our approach, and its experimental results are represented in Section IV. Finally, Section V concludes this paper.

II. EXISTING SCHEME

OVS is an open source virtual switch that is widely used for SDN and NFV. Since OVS adopted cuckoo hashing that guarantees fast search [3] as a hash scheme, the lookup performance and the table occupancy are improved. In this section, we describe the data structure and main operations of the cuckoo hashing in existing OVS.

A. Data Structure of Cuckoo Hash-table in Open vSwitch

Fig. 1 shows the data structure (64-bit system) of the cuckoo hash table in OVS. The whole table space is divided into a number of buckets. Each bucket has either five or seven slots, depending on whether the system is 32-bit or 64-bit.

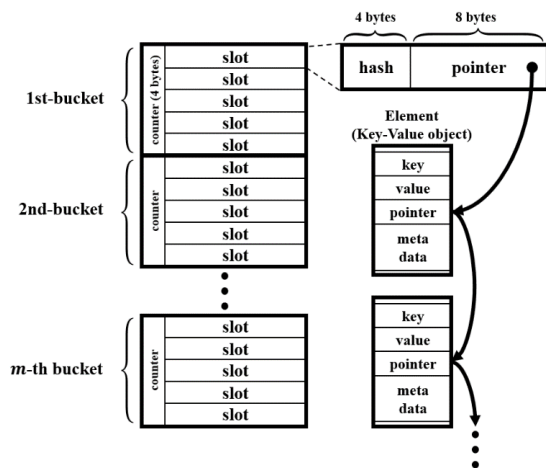


Figure 1. Data structure of cuckoo hash table in OVS on 64-bit system

Each slot of cuckoo hash table consists of a hash (4 bytes) and a pointer (4 bytes on 32-bit system, 8 bytes on 64-bit system). The hash field contains a hashed value of original key, and the pointer has a memory address where the element with the original key-value is stored. There is a counter field in each bucket, which is used to maintain consistency from concurrent data accesses. The number of slots per bucket is chosen to make buckets exactly one cache line in size. For example, one bucket size of a 64-bit system is 4 bytes (counter) + 4 bytes (hash) $\times$ 5 + 8 bytes (pointer) $\times$ 5 = 64 bytes, which is the cache line size commonly used to handle network traffic. By using this structure, each core of CPU can quickly access slots in one bucket since the slot data has been loaded on the cache line. The hash table size grows or shrinks according to the load factor (max: 85%, min: 20%). The number of buckets of the cuckoo hash table in OVS starts with one, and grows with a power of two when the load factor becomes more than 85%. In other words, 15% of the total slots are always empty. In contrast, the table shrinks by half when the number of elements is down below 20%. These maximum and minimum load factors are necessary to prevent the increase of insertion time and the waste of memory, respectively.

B. Operations of Cuckoo Hash-table in Open vSwitch

In this section, we describe two main operations (insertion, lookup) of cuckoo hash table. Because of space limitation, other operations are not included.

1) Insertion: The cuckoo hash table receives a hashed key (4 bytes) as an argument. The hashed key is hashed again by MurmurHash [8], which returns addresses of two buckets. If there is no slot with the same hash value as the given hashed key in the two buckets, the table searches an empty slot in the two buckets. Conversely, if the slot is found (i.e. a collision occurs), the address in the pointer field of the slot is replaced with the address in the pointer field of the new element, and then the existing element is linked to the new element with a linked-list. Unlike the previous case, if there is an empty slot in one of two buckets, the hashed key value and the memory address of the new element are stored in the hash field and the pointer field of the empty slot, respectively. When there is no

identical hash value, an empty slot is searched. If there is no empty slot, the hashed key is stored in one of the slots after pushing away one of the existing elements in the two buckets to make an empty slot. The pushed one can also push another element in another related bucket if there is no empty slot for it. This process is repeated until the table finds an empty slot or the number of iterations reaches the threshold (default is 500). If the threshold is reached, the cuckoo hash table rearranges total elements by rehashing all hash keys with a basis value randomly generated.

2) Lookup: Given a hashed key, the two buckets associated with the element that we want to find are determined through the hash function. If there is a slot which has the same hash value with the given hashed key, then the cuckoo hash table returns the address that the pointer of the slot has. With this address, we can access another pointer in the same space (*Struct* of C language) as the original key-value set. By accessing the original key in the relative location from the pointer address, we can check whether the key is the same as the given key for lookup. We take the value when those two are equal. Otherwise, we access the address of the next pointer and repeat the same operation until the pointer has NULL value. The number of memory accesses to get the value related to the key increases as the element that we want to locate is toward the end of the linked list. Thus, collisions affect the lookup performance because of the increase of memory accesses.

III. PROPOSED SCHEME

When we insert a new element into an existing cuckoo hash table, the collisions are resolved as shown in Fig. 2(a). Even though there is an empty slot in the same bucket, the existing cuckoo hash table does not insert the new element into the empty slot when there exists a slot with the same hash value. What that means is that, if there is a way to differentiate between the existing element and the new element that has the same hash value as the existing element in the same bucket, we can exploit the empty slot for the new element. Based on this idea, we propose *Collision Hash Function (CHF)* and *Collision Flag Bits (CFB)* to apply to existing cuckoo hash table. CHF generates a new hash value different from the hashed key that is the initially received argument. We implement CHF by using a simple hash function in [9]. The last slot called *standby slot (SS)* in each bucket is used for collision avoidance when it is empty. Only when there is a collision as inserting a new element and SS is empty, the new hash value generated by CHF is inserted into SS in the same bucket as the hashed key. To mark the index of the slot where a collision occurs, we use CFB which is the first 3 bits of the existing counter field of each bucket. With 3 bits, we can represent the slot index in the bucket since the number of slots in each bucket is 5 or 7 ($< 2^3$).

Fig. 2(b) shows one example of our approach. Suppose h_a which is the hashed key of key_1 is already stored in the second bucket when we try to insert a new element having h_a which is the hashed key of key_2 not equal to key_1 . If the SS is empty and the CFB is not set (default is 0) in the bucket, we

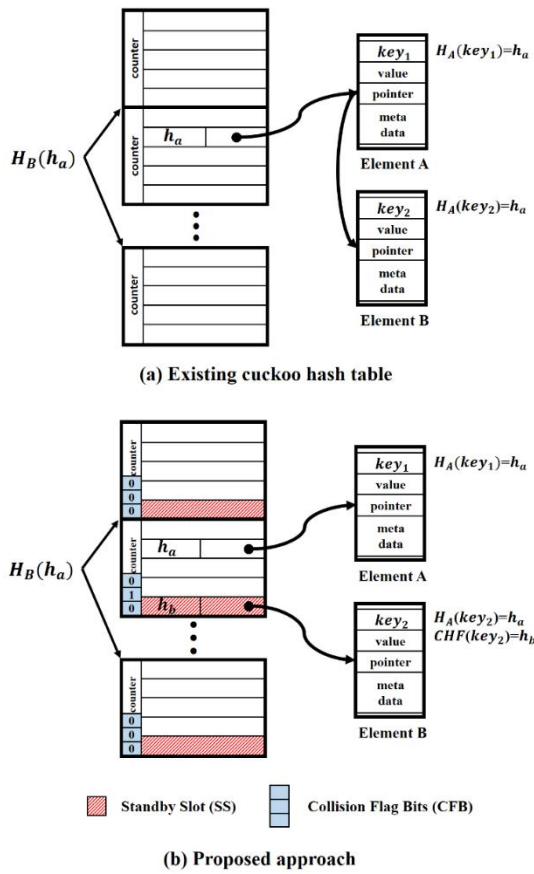


Figure 2. Examples of insertion operation of (a) existing cuckoo hash table and (b) proposed approach on 64-bit system

insert h_b into the hash field of the SS where h_b is the hashed key of key_2 generated by CHF. After that, the index of the slot collision occurs is stored in CFB, and the SS is linked to the new element via the pointer field of SS. During the lookup operation, first we check whether CFB is set or not if there is a slot which has the same hash value h_a for lookup. Only when the value of CFB is the same as the index of the slot having h_a and SS is not empty, we compare the hash value of SS and the hash value generated by CHF(key_2). If these hash values are the same, the hash value of SS is h_b . Hence, we return the address that the pointer of the SS has. Conversely, if these hash values are not the same, we return the address that the pointer of the slot having h_a has.

Through our approach, we can mitigate collisions and improve the table occupancy. However, it is not always possible to resolve collisions since SS is one of the slots used by the cuckoo hash table. As we mentioned, SS can be used for collision avoidance only when it is empty. Thus, when SS is being used for collision avoidance, SS should pass its space on to others in the following cases:

- There is no empty slot except the last slot (SS) when we insert a new element in a bucket.
- There is no empty slot except the last slot (SS) when we push away an element to other bucket.

These considerations mean that our scheme does not interfere with the operations of the existing cuckoo hash table. Note

that if the collision occurs again after the first collision in the same bucket when we insert a new element, our scheme behaves like the existing cuckoo hashing scheme with a linked-list for the new element. According to our experiments, we reduce about 20% collisions. Although CHF should be performed additionally for the elements that belong to 20%, our approach has a better performance than iterations of the linked-list.

IV. EVALUATION

In this section, we have conducted evaluations to verify whether our idea can be realized with an existing OVS. In particular, we measured the utilization rate of SS and the lookup time of our approach and compared them with the results from the existing cuckoo hash table. We used a machine with one 8-core CPU (Intel i7-5960X 3.00 GHz) and DDR4 16 GB RAM for a Linux (Ubuntu v14.04) environment where OVS is installed.

A. Utilization Rate of Standby Slot (SS)

To measure what percentage of elements are inserted into SS when collisions occur, we insert elements (85% of the number of slots) into the cuckoo hash table with the proposed scheme and the existing cuckoo scheme. Each element uses a destination IP address as a key. We conducted each experiment by varying the number of buckets and elements. We run experiments 100 times per case (total 5 cases). As shown in Table 1, we found that 15% to 20% of elements that experience collisions are inserted into SS in each bucket. In addition, the larger the table size, the higher the utilization rate of SS. These results indicate that our approach is more useful to mitigate collisions than existing cuckoo hash table.

TABLE 1. EXPERIMENTAL RESULTS

Number of buckets	Number of slots / 85% of #slots	Average of number of Collisions			
		Existing Cuckoo	With SS	With linked-list	Utilization Rate of SS
$2^{16}=65,536$	327,680 / 278,527	12.8	2.0	11.1	15%
$2^{17}=131,072$	655,360 / 557,055	29.9	5.3	25.8	17%
$2^{18}=262,144$	1,310,720 / 1,114,111	143.0	27.2	115.9	19%
$2^{19}=524,288$	2,621,440 / 2,228,223	568.5	113.8	455.2	20%
$2^{20}=1,048,576$	5,242,880 / 4,456,447	2,338.5	467.0	1,868.0	20%

B. Lookup Speed

When an element collision occurs, our approach performs CHF, and checking and modifying CFB, etc. We checked whether these additional operations slow down the lookup speed or not. In other words, the time required for all these operations should be less than the iterations of a linked-list. Thus, to compare the lookup speed of our approach and

existing cuckoo hash table, we measured the average time of each operation as shown in Fig. 3. We insert one million elements with unique keys into each table. After the insertion of elements, we measured the average lookup time of elements collision occurs. The left and right bars in the Fig. 3 indicate the average lookup times of element B in Fig. 2(a) and Fig. 2(b), respectively. As we can see, the lookup time of existing cuckoo hash table takes nearly two times longer than proposed scheme. The main reason is that the iterations of the linked-list is time-consuming work compared to other operations in existing cuckoo hash table case. Although our approach perform additional works related to *CHF* and *CFB*, these kind of works does not have much effect on the total lookup speed. In our architecture, the additional task which is required on all elements is only to check the *CFB* (it takes about 5 ns) to determine whether there is a conflict in the bucket. In addition, *CHF* is performed only for the elements involved in collision.

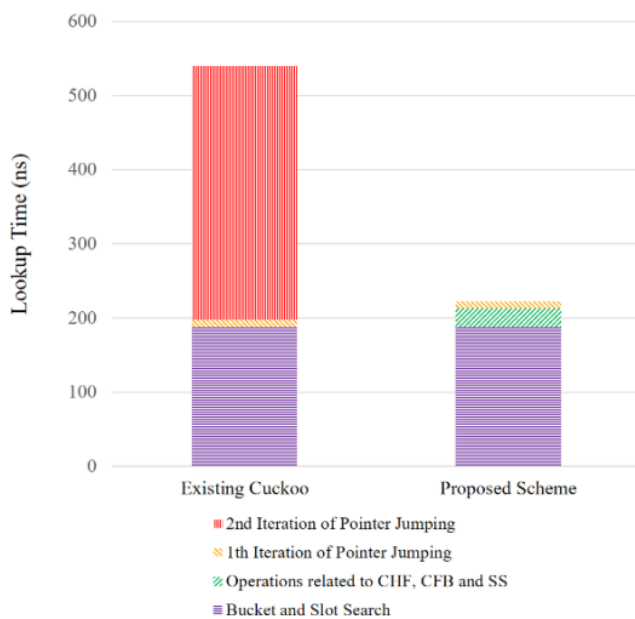


Figure 3. Lookup time of the element collision occurs of existing cuckoo hash table and proposed scheme

V. CONCLUSIONS

The collisions in a cuckoo hash table result in degrading performance due to multiple memory accesses. To mitigate collisions, we propose a collision-mitigation hashing scheme utilizing empty slots that are not being used. From our experiments, we confirm that the average lookup speed of 20% of elements that experience collision is improved about twice compared to existing cuckoo hash table. Moreover, the most important fact is that our approach does not interrupt basic operations of existing cuckoo hash table. These results and features are enough to be reflected in the next version of OVS. As a future work, we will try to improve the collision mitigation rate more by efficiently utilizing the other empty slots as well as the last empty slot in each bucket.

ACKNOWLEDGMENT

This work was supported by Institute for Information & communications Technology Promotion (IITP) grant funded by the Korea government (MSIP) (B0101-16-233, Smart Networking Core Technology Development).

REFERENCES

- [1] N. McKeown, T. Anderson, H. Balakrishnan, G. Parulkar, L. Peterson, J. Rexford, S. Shenker, and J. Turner, "OpenFlow: enabling innovation in campus networks," *ACM SIGCOMM Computer Communication Review*, vol. 38, No. 2, pp. 69-74, 2008.
- [2] Open vSwitch: an open virtual switch. [Online]. Available: <http://openvswitch.org/>.
- [3] R. Pagh and F. Rodler, "Cuckoo hashing," *Journal of Algorithms*, vol. 51, no. 2, pp. 122-144, 2004.
- [4] D. Knuth, *The Art of Computer Programming: Sorting and Searching*, vol. 3. Reading, MA, USA: Addison-Wesley, 1998, ch. 6.3.
- [5] B. Fan, D. G. Andersen, and M. Kaminsky, "MemC3: Compact and concurrent memcache with dumber caching and smarter hashing," in *Proc. of USENIX NSDI*, 2013.
- [6] H. Lee and A. Nakao, "Improving bloom filter forwarding architecture," *IEEE Communications Letters*, vol. 18, no. 10, pp. 1715-1718, 2014.
- [7] H. Lee and A. Nakao, "Improving Routing Table Lookup in Software Routers," *IEEE Communications Letters*, vol. 19, no. 6, pp. 957-960, 2015.
- [8] MurmurHash, Available: <https://sites.google.com/site/murmurhash/>
- [9] B. Stroustrup, *The C++ programming language*. Pearson Education India, 1995.



ChoongHee Cho was born in Seoul, Korea, in 1986. He received B.S. degree in Computer Science from the Sahmyook University, Seoul, Korea, in 2010, and he is an integrated M.S. and Ph.D. student in Information and Communication Network Technology of the Korea University of Science and Technology (UST), Daejeon, Korea, since 2014. In 2014, he joined the Optical Transport Network Research Section, ETRI, Korea, as an UST graduate student and his current research interests include SDN, OpenFlow protocol, NFV and Shared Mesh Protection.



JungBok Lee He received B.S degree in Computer Science and Engineering from the Konyang University, Nonsan, Korea, in 2010. He studied security, network, DPI, OpenFlow protocol and Open vSwitch. In 2015, he joined the portal-mail infrastructure team, Kakao Corp, Korea, and his current research interests include SDN and NFV-related web services for the cloud infrastructure.



Jeong-dong Ryoo is a principal researcher at ETRI, Daejeon, Rep. of Korea and a professor at the Department of Engineering, Korea University of Science and Technology, Daejeon, Rep. of Korea. He holds an MS and a PhD in EE from the Polytechnic Institute of New York University, New York, USA, and a BS in EE from Kyungpook National University, Daegu, Rep. of Korea. After completing his PhD in the area of telecommunication networks and optimization, he started working for Bell Labs, Lucent Technologies, NJ, USA, in 1999. While he was with Bell Labs, he was mainly involved with performance analysis, evaluation, and enhancement study for various wireless and wired network systems. Since he joined ETRI in 2004, his work has been focused on next generation network, carrier class Ethernet and MPLS-TP technology research, especially participating in OAM and protection standardization activities in ITU-T. He is the editor of various ITU-T Recommendations including G.8131 (MPLS-TP linear protection), G.8132 (MPLS-TP ring protection) and G.808.1 (Generic protection - Linear), and a vice-chairman of ITU-T Study Group 15. He co-authored *TCP/IP Essentials: A Lab-Based Approach* (Cambridge University Press, 2004). He is a member of the Eta Kappa Nu association.