Guided Multiple Hashing: Achieving Near Perfect Balance for Fast Routing Lookup

Xi Tao Yan Qiao Jih-Kwon Peir Shigang Chen
Department of Computer Information Science Engineering
University of Florida

Zhuo Huang
Facebook Inc.
Menlo Park, CA

Shih-Lien Lu
Intel Research Labs
Intel Corp.

***Abstract*—The routing and packet forwarding function is at the core of the IP network-layer protocols. The throughput of a router is constrained by the speed at which the routing table lookup can be performed. Hash-based lookup has been a research focus in this area due to its $O(1)$ average lookup time, as compared to other approachs such as trie-based lookup which tends to make more memory accesses. With a series of prior multi-hashing developments, including d-$random$, 2-$left$, and d-$left$, we discover that a new *guided multi-hashing approach* holds the promise of further pushing the envelope of this line of research to make significant performance improvement beyond what today's best technology can achieve. Our guided multi-hashing approach achieves near perfect load balance among hash buckets, while limiting the number of buckets to be probed for each key (address) lookup, where each bucket holds one or a few routing entries. Unlike the localized optimization by the prior approaches, we utilize the full information of multi-hash mapping from keys to hash buckets for global key-to-bucket assignment. We have dual objectives of lowering the bucket size while increasing empty buckets, which helps to reduce the number of buckets brought from off-chip memory to the network processor for each lookup. We introduce mechanisms to make sure that most lookups only require one bucket to be fetched. Our simulation results show that with the same number of hash functions, the guided multiple-hashing schemes are more balanced than *d-left* and others, while the average number of buckets to be accessed for each lookup is reduced by 20–50%.**

I. Introduction

Hashing provides an efficient mechanism to organize, look up, and update a large information table based on a key value, which has been used in a wide variety of network applications [1], [2], [3]. In particular, hash-based lookup has been an important research direction on routing and packet forwarding, among the core functions of the IP network-layer protocols. While there are other alternative approaches for routing table lookup such as trie-based solutions, this paper's focus is on hash-based solutions, which have the advantages of simplicity and $O(1)$ average lookup time, whereas trie-based lookup tends to make much more memory accesses.

Single-hashing however suffers from the collision problem, where multiple keys are hashed to the same bucket and cause uneven distribution of keys among the buckets. It takes variable delays in looking up keys located in different buckets. For hash-based network routing tables [1], [2], [3], [6], it is critical to perform fast lookup for the next hop routing information. In today's backbone routers, routing tables are often too big to fit into on-chip memory of a network processor. As a result, off-chip routing table access becomes the bottleneck for meeting the increasing throughput requirement on high-speed Internet [7], [8]. The unbalanced hash buckets further worsen the off-chip access. Today's memory technology is more efficient to fetch a contiguous block (such as a cache block) at once than individual data elements separately from off-chip memory. A heavy-loaded hash bucket may require two or more memory accesses to fetch all its keys. However, in order to accommodate the most-loaded bucket for a constant lookup delay, fetching a large memory block which can hold the highest number of keys in a bucket increases the critical memory bandwidth requirement, wastes the memory space, and lowers the network throughput [2], [6], [9], [10].

Methods were proposed to handle the hash collision problem for balancing the bucket load by reducing the maximum number of keys in a bucket among all buckets. One approach is to use *multiple* hashing such as *d-random* [11] which hashes each key to d buckets using d independent hash functions and stores the key into the least-loaded bucket. The *2-left* scheme [1], [9] is a special case of *d-random* where the buckets are partitioned into left and right regions. When inserting a key, a random hash function is applied in each region and the key is allocated to the least-loaded bucket (to the *left* in case of a tie). The multiple-hashing approach balances the buckets and reduces the fetched bucket size for each key look up. However, without the knowledge of which bucket that a key is located, *d-random* (*d-left*) requires probing all d buckets. As the bottleneck leans on the off-chip memory access, accessing multiple buckets slows down the hash table access and degrades the network performance [8], [10].

To remedy probing d buckets, extended Bloom Filter[3] uses counters and extra pointers to link keys in multiple hashed buckets to avoid lookups of multiple buckets. However, it requires key replications and must handle complex key updates. The recently proposed Deterministic Hashing [6] applies multiple hash functions to an on-chip intermediate index table where the hashed bucket addresses are saved. By properly setting up the bucket addresses in the index table, the hashed buckets can be balanced. This approach incurs space overhead and delays due to indirect access through the index table. In [10], an improved approach uses an intermediate table to record the hash function IDs, instead of the bucket addresses to alleviate the space overhead. In addition, it uses a single hash function to the index table to ease the update complexity. However, with limited index table and hashing functions, the achievable balance is also limited. In another effort to avoid collision, the perfect hash function sets more rigid goal to achieve one to one mapping between keys and buckets. It accomplishes the goal using complex hash functions encoded on-chip with significant space and additional delays [12], [13]. It also requires changes in the encoded hash function upon a hash table update.

In this paper, we propose a new multiple-hashing method, named *guided multiple hashing*, denoted as *d-ghash*, for balancing the hash buckets while limiting the number of bucket access for key lookup. Unlike *d-random* [11], which places each key into the least-loaded bucket progressively for localized balance among the hashed buckets, *d-ghash* achieves global balance among all buckets based on the complete placement information after all keys are hashed into buckets d times using d independent hash functions. In other words, the decision of individual key placement is deferred until the knowledge of the complete placement is available.

There are two criteria for placing the keys into buckets. First, the maximum number of keys in buckets should be as small as possible. We define the *optimal bucket load* as the lower bound on the maximum number of keys in a bucket, denoted as Ω_p. Clearly, $\Omega_p = \lceil n/m \rceil$, where n is number of keys and m is the number of buckets. *d-ghash* ensures that the maximum number of keys in a bucket is close to Ω_p. In order to achieve this, it assigns each key to one of the hashed buckets and removes duplicates from other buckets after the complete placement information is known. Reassignment may be needed for buckets exceeding the optimal load. Second, in order to reduce bucket access, *d-ghash* creates as many empty buckets as possible, called *companion empty* buckets, or *c-empty* buckets. A small array called the *empty* array is built to indicate if a bucket is a *c-empty* bucket. As it is unnecessary to access an empty bucket to find a key, the number of the bucket accesses for a lookup is reduced.

To further reduce lookup time, a *target* array is established to guide the lookup of a key. Basically it records the hash function ID that each key uses to locate the assigned bucket. We call the *d-ghash* scheme with a *target* array the *enhanced d-ghash*, to separate from the *base d-ghash* that does not have the *target* array. Upon looking up a key, the hash function ID is retrieved from the *target* array based on the key value, and then the corresponding hash bucket is fetched from the off-chip memory. Only when the key is absent from this bucket, the remaining buckets are examined. Collision may happen in the *target* array. Using a sufficient *target* array, however, it has limited impact as we will discuss in Section IV.

In summary, the proposed *d-ghash* makes two important contributions. First, *d-ghash* achieves near perfect balance using multiple hash functions. By deferring individual key assignment, it is able to achieve global balance. Second, *d-ghash* further utilizes and arranges the available bucket space to reduce the number of buckets required to look up a key. It creates as many empty buckets as possible when removing duplicate keys during the assignment to avoid probing such empty buckets. It also records the hash function to the bucket where the key is located to guide the lookup. These two techniques together can reduce the bucket access significantly in comparison with other multiple-hashing approaches.

Performance evaluation shows significant advantages of using the guided multiple-hashing approach. Given different ratios of the number of keys, buckets, and hash functions, *d-ghash* out-balances existing *single-hash* and *d-left* hashing schemes by producing smaller bucket load among all buckets. For example, given 200,000 keys, *d-ghash* can achieve perfect balance with a maximum of one key per bucket using 4 hash functions and 275,000 buckets while other scheme cannot obtain the perfect balance even with 500,000 buckets. Meanwhile, *enhanced d-ghash* has the ability to reduce 20–50% of the bucket access for a key lookup in comparison with the *d-left* approaches. We also show that a similar conclusion can be made when applying *d-ghash* to several real network routing tables [7]. Results show that *enhanced 4-ghash* reduces 37–50% bucket access of *4-left* for a key lookup and *enhanced 2-ghash* reduces 20–23% bucket access of *2-left*. Simulations and experiments based on real routing table trace also demonstrate the robustness of our algorithm.

The remaining paper is organized as follows. We will first provide the background and motivation of using multiple hash functions to accomplish specific objectives in the hash table in Section II. Section III describes the detailed algorithm of the proposed guided multiple-hashing method. This is followed by the performance evaluation and comparison of the state-of-the-art hashing methods in Section IV. Section V provides a routing table experiment based on real Internet traces. Related works are give in Section VI and Section VII concludes the paper.

II. Background and Motivation

In this section, we describe the challenges of a hashing-based information table using a single hash function. We also bring up the motivation and applications of using a multiple-hashing approach for organizing and accessing a hash table. It is well-known that hash collision is an inherent problem when a single random hash function is used, which causes uneven distribution of keys among the hash buckets in a non-deterministic fashion. The multiple-hashing technique, on the other hand, uses d independent hash functions to place a key into one of d possible buckets. The criteria of selecting the target bucket for placement is flexible and can be controlled to accomplish a specific objective. One well-known objective of using multiple hash functions is load balancing, i.e. to balance the keys in the buckets [1], [5], [9], [11], [14], [15]. To achieve this objective with two hash functions, for example, each key is placed in the bucket with smaller number of keys. The power of balancing the buckets with two choices was reported in [9]. In comparison with single hashing, the maximum number of keys in a bucket can be reduced from $((1+o(1)) \ln n/ \ln \ln n)$ to $(\ln \ln n/ \ln 2 + O(1))$ when placing n keys into n buckets [11]. However, due to the two choices, the exact location of a key is unknown and both buckets need to be probed to look up a key.

Another known objective of multiple hashing sets an opposite criteria for reducing the *fill factor* of the hash buckets [9], [16]. The *fill factor* is measured by the ratio of non-empty buckets. Instead of placing a key in the bucket with smaller number of keys for load balancing, this approach places the key in the bucket with non-zero number of keys. The objective of this placement is to maximize the amount of empty buckets. One potential application is to apply the low *fill-factor* hashing method to a Bloom filter [9], [16]. A *Bloom filter* [17] is an efficient bit-map data structure $B[1], ..., B[m]$ with m bits for performing membership queries of a set of n keys, $S = \{x_1, x_2, ..., x_n\}$. $B[1], ..., B[m]$ are set to zero initially. Each key x is hashed to B k times using k independent hash functions $H_1, H_2, ..., H_k$ and set $B[H_i(x)] = 1$ where $1 \leq i \leq k$. When querying whether y is a member in the key set, we can check if all $B[H_i(y)] = 1$, $1 \leq i \leq k$. Any zero in $B[H_i(y)]$ guarantees y is not a member. A *false positive* occurs when y satisfies the membership query, but is actually not a member. With more zeros remained in the Bloom filter, the critical false positive rate can be reduced. To create more zeros in establishing the Bloom filter, however, multiple sets of hash functions are needed for different keys since all the hashed k bits for each key must be set during the setup of the Bloom filter. Therefore, the multiple hashing concept is actually applied for choosing a set of hash functions out of multiple groups to maximize the number of zeros in the Bloom filter after recording k '1's for every key.

To demonstrate the power of multiple hashing in accomplishing different objectives for the hash table, we compare the simulation results of four hashing schemes: single hashing (*single-hash*), 2-hash with load balancing (*2-left*), 4-hash with load balancing (*4-left*), and 2-hash with maximum zero buckets (*2-max-0*). We simulate 200,000 randomly generated keys to be hashed to 100,000 buckets. The distribution of keys in buckets is plotted in Figure 1. We can observe substantial differences in the key distribution among the four hashing schemes. The maximum number of keys in a bucket reach to ten for *single-hash* and *2-max-0*. Meanwhile, *2-max-0* produces 2.5 times empty buckets than *single-hash* does. *2-left* and *4-left* are more balanced with four and three as the maximum numbers of keys in a bucket, respectively. It's easy to see that increasing the number of hash functions from two to four helps improving the balance.

In this paper, we investigate a new objective for allocating keys in hash buckets using the multiple-hashing approach. The goal is to balance the keys in the buckets to reduce the bucket load so that the amount of data in a bucket to be fetched from off-chip memory is minimized. In addition, we want the average number of buckets to be probed for looking up a key to be as small as possible to alleviate the disadvantage of multiple hashing, which needs to probe multiple buckets.

TABLE I
NOTATION AND DEFINITION

n	Total number of keys
m	Total number of buckets
$B[i]$	Set of keys in i-th bucket
$v(B[i])$	Number of keys of the i-th bucket
s	indices of the buckets in B sorted in ascending order of $v(B[i])$
H_i	i-th hash function
Ω_p	Optimal bucket load, $\lceil n/m \rceil$
Ω_a	Achievable bucket load
n_u	Total number of keys in under-loaded buckets (bucket load less than Ω_a)
b_u	Number of under-loaded buckets
θ	Memory usage ratio

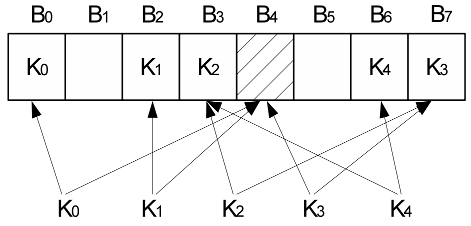

Fig. 2. A simple d-ghash table with 5 keys, 8 buckets and 2 hash functions. (The shaded bucket is a *c-empty* bucket. The final key assignment is as illustrated.

III. d-$ghash$ HASHING SCHEME

In this section, we describe the detailed algorithms of the guided multiple-hashing scheme that consists of a setup algorithm, a lookup algorithm, and an update algorithm. Assume we have m buckets $B_1, ..., B_m$ and d independent hash functions $H_1, ..., H_d$. Each key x is hashed and placed into *all* d buckets, $B_{H_i(x)}$, $1 \leq i \leq d$. The set of keys in bucket B_i is denoted by $B[i]$, and the number of keys in bucket B_i is $v(B[i])$, $1 \leq i \leq m$. The *bucket load* Ω_a is defined as the maximum number of keys in any bucket. We define the *memory usage ratio* as: $\theta = (\Omega_a \times m)/n$ to indicate the memory requirement of the hash table. Other terminologies are self-explanatory and are listed in Table I. For better illustration of d-ghash, we use a simple hashing table with 5 keys and 8 buckets. All keys are hashed to the buckets using two hashing functions, where buckets B_0 to B_7 have 1, 0, 1, 2, 3, 0, 1, 2 keys as indicated by the arrows in Figure 2.

A. The Setup Algorithm

Since the objective is to minimize the bucket load while approaching to a single bucket access per lookup, the setup algorithm needs to satisfy two criteria: (1) achieving near

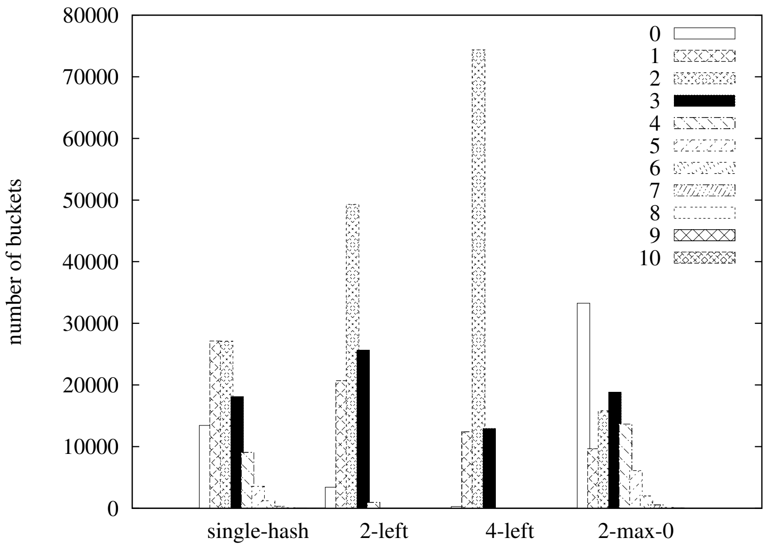

Fig. 1. Distribution of keys in buckets of four hashing algorithms.

perfect balance, and (2) maximizing the number of *c-empty* buckets. Recall that a *c-empty* bucket serves as a multiple-hashing target of one or more keys, but the key(s) is placed into other alternative buckets that make *c-empty* bucket access unnecessary.

Given n keys and m buckets, a perfect-balance hashing scheme achieves optimal bucket load $\Omega_p = \lceil n/m \rceil$. However, the perfect balance may not be achieved under our or other multi-hashing schemes because even with multiple hashing, some buckets may still probabilistically be *under-loaded*, i.e. zero or less than Ω_p keys are hashed to the bucket. And this translates to some other buckets being squeezed with more keys. Increasing the number of hash functions reduces under-loaded buckets and helps approaching to the perfect balancing. Our simulation shows that with 4 hash functions, the achievable balance is the same as or very close to Ω_p.

The first step in the setup algorithm is to estimate Ω_a, the achievable balance. The idea is to count the number of under-loaded buckets and the number of keys inside. If the remaining buckets can not hold the rest of the keys with Ω_a keys in each bucket, we increase Ω_a by one. Details are shown in Algorithm 1. We then use Ω_a as the benchmark bucket load for key assignment. We sort all buckets in B resulting a sorted index array, s, such that $v(B[s(i)]) \leq v(B[s(i+1)]), 1 \leq i \leq m-1$. In the simple example of Figure 2, $\Omega_a = \Omega_p = \lceil n/m \rceil = 1$.

The next step is key assignment, which consists of two procedures, creating *c-empty* buckets, and balancing key assignment. For creating *c-empty* buckets, the procedure removes duplicate keys starting from the most-loaded buckets to maximize their services as companion buckets to reduce the bucket access. A key can be safely removed from a bucket if it exists in other bucket(s). As shown in Algorithm 2, the procedure goes through all buckets whose initial load are greater than Ω_a and tries to remove keys from them. In the illustrated example

Algorithm 1: Achievable Ω_a

```
1  initialize $\Omega_a = \lceil n/m \rceil$;
2  $n_u = 0$; $b_u = 0$; $S = \emptyset$;
3  for $i = 1$ to $m$ do
4      // count buckets and keys in under-loaded buckets
5      if $v(B[i]) < \Omega_a$ then
6          $n_u = n_u +$ no. of distinct keys in $B[i]$ not in $S$;
7          $b_u = b_u + 1$;
8          add distinct keys in $B[i]$ to $S$;
9  $n_k = n - n_u$; $b_k = \lceil n_k/\Omega_a \rceil$;
10 if $b_u + b_k > m$ then
11     $\Omega_a = \Omega_a + 1$; go back to (2);
```

Algorithm 2: Create *c-empty* buckets

```
1  set count($key$) = $d$ for all $keys$;
2  repeat based on the sorted list $s$
3      find next most-loaded bucket with largest $v(B[s(i)])$;
4      if $v(B[s(i)]) > \Omega_a$ then
5          for each key $x$ in bucket do
6              if count($x$)$> 1$ then
7                  delete key in the bucket;
8                  count($x$) = count($x$)$-1$;
```

in Figure 2, all 3 keys in B_4 are successfully removed and B_4 becomes empty. Next, we check B_3 and B_7, each of which has 2 keys. Note that both K_2 and K_4 in B_3 can be removed if B_3 is emptied first. As a result, K_2 and K_3 cannot be removed from B_7 and exceed Ω_a. All buckets with the bucket load exceeding Ω_a will be a target for reallocation as described next.

After emptying the buckets, the key assignment procedure assigns each key to a bucket starting from the least-loaded bucket as described in Algorithm 3. Once a key is assigned, its duplicates are removed from the remaining buckets. During

Algorithm 3: Balanced key assignment

```
// Assign keys:
repeat based on the sorted list $s$
    find next unassigned bucket with smallest $v(B[s(i)])$;
    if $v(B[s(i)]) \leq \Omega_a$ then
        assign all keys in $B[s(i)]$;
        remove keys in $B[s(i)]$ from other buckets;
        update $s$;
until all buckets $B[i]$ are either assigned or $v(B[i]) > \Omega_a$;

// Reassign overflow buckets:
repeat based on the sorted list s
    find next maximum load bucket $v(B[s(i)]) > \Omega_a$;
    for each key $x$ in $B[s(i)]$ do
        if exists $i$, such that $0 < v(B[H_i(x)]) < \Omega_a$
            then reassign $x$ to $B[H_i(x)]$;
        else choose the smallest of all $v(B[H_i(x)])$
            then reassign $x$ to that bucket;
if all buckets $B[i]$ satisfies $v(B[i]) \leq \Omega_a$, then stop;
if any bucket with $v(B[i]) > \Omega_a$
    go back to (11) and iterate $r$ times;
if exceed $r$ times then
    $\Omega_a = \Omega_a + 1$; redo key assignment;
```

the assignment, buckets with more than Ω_a keys are skipped in order to maintain the achievable balance. A re-assignment of the buckets with load greater than Ω_a is necessary after all the buckets are assigned. During re-assignment, the keys in the overflow buckets are attempted to be relocated to other buckets. In our experiment, we use Cuckoo Hashing [5] to relocate keys from an overflow bucket to an alternative bucket using multiple hashing functions. If all alternative buckets are full, an attempt is made to make room in the alternative buckets. For simplicity, however, such attempts stop after r tries, where r can be any heuristic number. A larger r brings better balance at the expense of longer setup time. In the illustrated example, K_2 in B_7 is relocated to B_3 to reduce the bucket load of B_7, hence, the optimal load is achieved.

In case that the perfect balance is not achievable, Ω_a is incremented by one and the key assignment procedure repeats. It is important to note that the priority of the key assignment is to achieve perfect balance. Therefore, the keys that are previously removed from an empty bucket can be reassigned back in order to accomplish the perfect balance such that the number of keys are less than or equal to Ω_a in all buckets. It is also important to know that in order to reduce the bucket load, we can decrease the ratio of n/m, i.e. to increase the number of buckets for a fixed number of keys. However, increasing the number of buckets inflates the memory space requirement as the memory usage ratio can be calculated by $\theta = (\Omega_a \times m)/n$ for a constant bucket size for efficient fetch of a bucket from off-chip memory.

Algorithm 4: Key lookup

```
the lookup key $x$ is hashed using $\{H_i\}$;
use $\{H_i(x)\}$ as indices to get $d$ bits from empty array;
let $e$ = number of 1s in $d$ bits from the empty array;
retrieve a hash function ID $t$ from the target array;
if $e = 1$ then fetch the only non-empty bucket;
else if $e > 1$ then
    lookup the bucket $B[H_t(x)]$;
    if $x$ is not found then
        lookup remaining nonempty buckets till found
```

B. The Lookup Algorithm

In order to speed up the lookup of keys, we introduce a data structure called the *empty* array, which is a bit array of size m indicating whether a bucket is empty or not. If a bit in the *empty* array is '0', it means that the corresponding bucket is empty; otherwise it is not empty. Upon looking up a key x, the bits of indices $H_1(x), ..., H_d(x)$ in the *empty* array are checked. If there is only one of the hashed buckets is non-empty, we simply fetch that bucket and thus complete a lookup. If there are two or more non-empty buckets, we access them one by one until we find the key. Algorithm 4 gives the details. In the worst case, all d bits are ones and d buckets are examined before we find the key. As discussed above, creating *c-empty* buckets helps reduce bucket accesses per lookup, thus alleviates the lookup cost.

To further enhance our algorithm, we introduce another data structure, the *target* array, to record the hash function ID once a key is hashed to two or more non-empty buckets. To separate from the algorithm described above, we call it *enhanced d-ghash* algorithm. The algorithm only using the *empty* array is called *base d-ghash* algorithm. The recorded ID indicates the bucket that the key is most-likely located. The *empty* array has m bits while the size of the *target* array varies depending on the number of keys. Suppose m = 200K, and we use 200K-entry *target* array, then the *empty* array takes 25KB and *target* array 25KB for *enhanced 2-ghash* and 50KB for *enhanced 4-ghash.* These two small arrays can be placed on chip for fast access. Multiple keys may collide in the *target* array. When a collision occurs, the priority of recording the target hashing function is given to the key which hashes to more non-empty buckets. Given a fixed number of keys, we can adjust the number of buckets (m) and hash functions (d) to achieve a specific goal of the bucket size and the number of buckets to be fetched for looking up a key. More discussions will be in Section IV.

C. The Update Algorithm

There are three common types of hash table updates: insertion, deletion, and modification. It is straightforward to delete or to modify a key in the hash table. For deletion, the key is probed first by fetching the bucket from off-chip memory. The key is then removed from the bucket before the bucket is written back to memory. If the key is the last one in the bucket, the corresponding bit in the *empty* array is set to zero. For modification of the associated record of a key, the key and its associated record are fetched. The new record replaces the old one before the bucket is written back to memory. Those two types of updates do not involve the modification of the *target* array.

The key insertion is slightly complicated. All hashed buckets are probed and the key is inserted into the least-loaded, non-empty bucket with the number of keys $< \Omega_a$. If all non-empty buckets are full, the key is inserted into an empty bucket if it is also hashed. The *empty* array is updated accordingly. In case that all hashed buckets are full, the Cuckoo Hashing is applied to make a room for the new key, i.e., "rehashing" a key in one of the hashed buckets to another alternative bucket. During key relocations, both the *empty* and the *target* arrays are updated accordingly.

There are two options in case a key cannot be inserted without breaking the property of $v(B[i]) \leq \Omega_a$, i.e., all its hashed/rehashed bucket loads are greater than or equal to Ω_a. First, set $\Omega_a = \Omega_a + 1$ and insert the key normally; Second, initiate an off-line process to re-setup the table. Normally, the possibility that a key cannot be inserted is small, and we should use the second option to prevent the bucket size from growing fast. However, if this operation happens very frequently, it implies that most of the buckets are "full", i.e. the average number of keys in buckets are approaching Ω_a. In this case we should use the first option. By increasing the maximum load by one, all buckets gain one extra space to store another key.

IV. Performance Evaluation

The performance evaluation is based on simulations for seven hashing schemes: *single-hash*, *2-left*, *4-left*, *base 2-ghash*, *enhanced 2-ghash*, *base 4-ghash*, and *enhanced 4-ghash*. Note that we do not include *d-random* in the evaluation, because it is outperformed by *d-left* both in terms of the bucket load and the number of bucket accesses per lookup. We simulate 200,000 randomly generated keys to be hashed into 100,000 to 500,000 buckets. To test the new multiple hashing idea, we adopt the random hash function in [18] which uses a few *shift*, *or*, and *addition* operations on the key to produce multiple hashing results. For relocation, we try to relocate keys in no more than ten buckets to other alternative buckets in the Setup Algorithm and no more than two in the Update Algorithm. We first compare the bucket load and the average number of bucket accesses per lookup by varying n/m. Then we normalize the number of keys per lookup based on the memory usage ratios to understand the memory overhead for different hashing schemes. In addition, we demonstrate the effectiveness of creating *c-empty* buckets to reduce the bucket access. We also give a sensitivity study on the number of bucket accesses per lookup with respect to the size of the *target* array. Lastly, we evaluate the robustness of *d-ghash* scheme by using two simple probabilistic models.

Figure 3 displays the bucket loads of the hashing schemes.

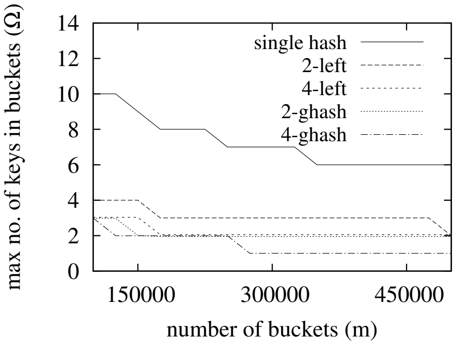

Fig. 3. Bucket loads for the five hashing schemes. The *enhanced d-ghash* scheme and *base d-ghash* scheme has the same bucket load.

Note that *enhanced d-ghash* and *base d-ghash* have the same bucket load. The only difference between the two is that *enhanced d-ghash* uses a *target* array to reduce the number of bucket accesses per lookup. The results show that *d-ghash* has the least bucket load, and hence achieves best balance among the buckets. This is followed by *d-left*. More hash functions improve the balance for both *d-ghash* and *d-left*. With 275,000 buckets, *4-ghash* accomplishes perfect balance with the bucket load of a single key. No other simulated scheme can achieve such balance with up to 500,000 buckets. *2-ghash* performs slightly better than *4-left* as the former needs 150,000 buckets to reduce the bucket load to two keys while the latter requires 175,000 buckets. This result demonstrates the power of *d-ghash* in balancing the keys over that of *d-left*. The *single-hash* scheme is the worst. The bucket load is six even with 500,000 buckets. Note that bucket load is an integer, but we slightly adjust the integer values to separate the curves of different schemes for easy read.

In Figure 4, we evaluate the lookup efficiency of the seven hashing schemes. *Single-hash* only accesses one bucket per lookup. The *d-left* scheme looks up a key from the left-most bucket. In case that the key is not found, the next bucket to the right is accessed until the key is located. Since the key is always placed in the left-most bucket to break a tie, the number of bucket accesses per lookup is quite low, $1.68 \sim 2.36$ for *4-left* and $1.27 \sim 1.44$ for *2-left*. The *base 4-ghash* and *base 2-ghash* reduce the number of bucket accesses per lookup to $1.25 \sim 2.18$ and $1.11 \sim 1.44$ respectively with a 5–34% and 0–14% reduction. With a *target* array of $1.5n$ entries, the *enhanced 4-ghash* and the *enhanced 2-ghash* can further reduce the number of bucket accesses per lookup to as low as $1.03 \sim 1.23$ and $1.01 \sim 1.11$ respectively with a 38–51% and 21–24% reduction.

It is interesting to see that the number of bucket accesses per lookup for *d-ghash* does not decrease continuously when the number of buckets increases. We can observe a sudden jump at $m = 125,000$ and $m = 275,000$ for *4-ghash*. This is due to the fact that the optimal bucket load drops from three to two when $m = 125,000$ and from two to one when $m = 275,000$. As the average number of keys per bucket is very close to the optimal bucket load, it is hard to create *c*-

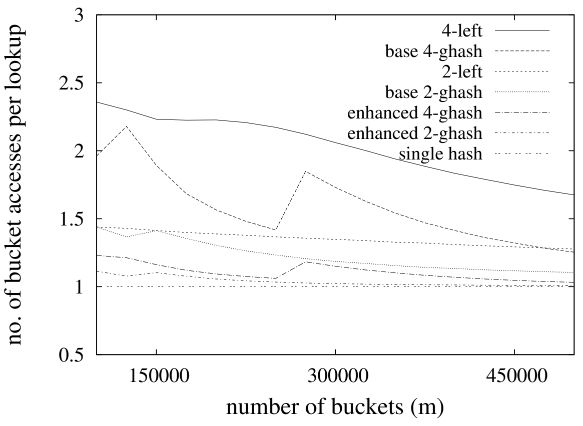

Fig. 4. Number of bucket accesses per lookup for d-ghash.

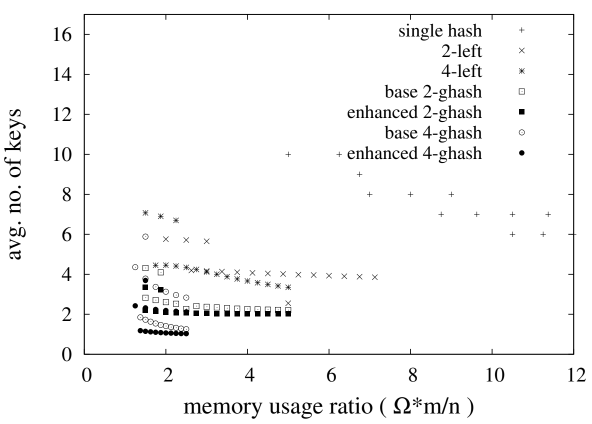

Fig. 5. Average number of keys per lookup based on memory usage ratio.

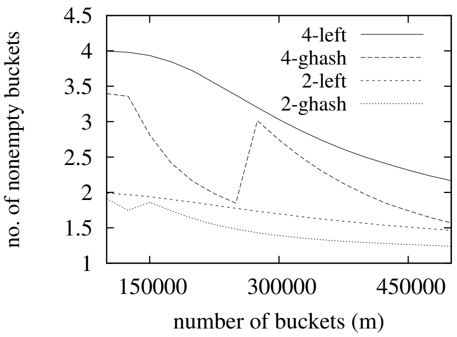

Fig. 6. The average number of non-empty buckets for looking up a key. This parameter is the same for *enhanced d-ghash* scheme and *base d-ghash* scheme.

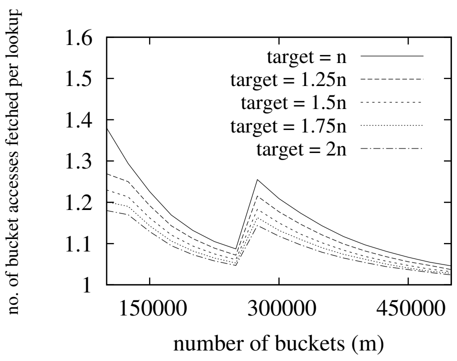

Fig. 7. Sensitivity of the number of bucket accesses per lookup for *enhanced 4-ghash* with respect to the *target* array size.

empty buckets. Therefore, there are sudden decreases in the amount of *c-empty* buckets at those two points. As a result, *4-ghash* experiences more bucket access for key lookup. The same reason goes for *2-ghash*.

In order to reduce the bucket load for a fixed number of keys, we can increase the number of buckets. However, increasing buckets inflates the memory space requirement. In Figure 5, we plot the average number of keys per lookup based on the memory usage ratio, where the average number of keys is the product of the bucket load and the average number of buckets per look up. The results clearly show the advantage of the *d-ghash* scheme. *Enhanced 4-ghash* accomplishes a single key per bucket with 275,000 buckets which are only 37% more than the number of keys. With slightly larger than one key per lookup, *enhanced 4-ghash* requires the least amount of memory to achieve close to one key access per lookup.

Besides the perfect balance, *d-ghash* creates *c-empty* buckets to maximize the number of keys hashing to empty buckets. Figure 6 shows the effectiveness of the *c-empty* buckets for reducing the bucket access. In this figure, y-axis indicates the average number of non-empty buckets that each key is hashed into. In comparison with *d-left*, *d-ghash* reduces non-empty buckets more significantly, resulting in smaller number of bucket access. For *d-left*, the number of non-empty buckets decreases as the number of buckets increases. This is due to the fact that *d-left* assigns each key to the least-loaded bucket. *d-ghash*, on the other hand, creates *c-empty* buckets by removing keys away from those buckets with more keys hashed into. As a result, there are fewer non-empty buckets for looking up each key. It is interesting to observe that the ability to create *c-empty* buckets depends heavily on the optimal bucket load and the ratio of keys and buckets. For example, when the number of buckets is 250,000, the optimal bucket load is 2 for 200,000 keys that leaves plenty of room to create many *c-empty* buckets. However, when the number of buckets increases to 275,000, the optimal bucket load drops to 1 that leaves little room for the *c-empty* buckets. Hence, the average number of non-empty buckets increases for each key to be hashed into.

Moreover, we show a sensitivity study of bucket access per lookup with respect to the size of the *target* array. We vary the size of the *target* array from n to $2n$ entries using *enhanced 4-ghash* and the result is shown in Figure 7. As expected, larger *target* array reduces the collision, resulting in a smaller number of bucket accesses. We pick $1.5n$ entries as the *target* array size in earlier simulations which has the best tradeoff in terms of space overhead and bucket access per lookup.

Finally, we evaluate the robustness of our scheme. We first set up a table using 200,001 keys, 200,000 buckets, and 300,000 *target* array entries. The achievable bucket load Ω_a is 2 in this setting. We simulate two update models:

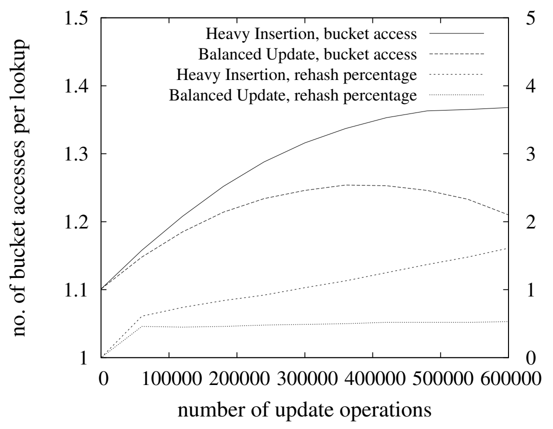

Fig. 8. Changes in the number of bucket accesses per lookup and rehash percentage for two update models using *enhanced 4-ghash*. Bucket accesses per lookup lines correspond to the left Y axis. Rehash percentage lines correspond to the right Y axis.

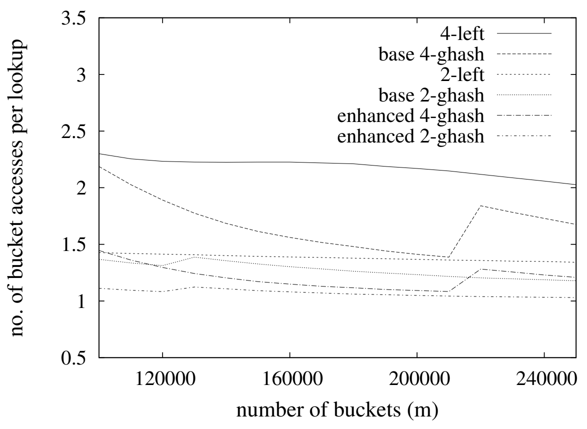

Fig. 9. Number of bucket accesses per lookup for experiments with five routing tables.

(1) *Balanced Update*: 33% insertion, 33% deletion, and 33% modification; and (2) *Heavy Insertion*: 40% insertion, 30% deletion, and 30% modification. We run for 600K updates and record the rehash percentage of all the update operations and the number of bucket accesses per lookup. The results are presented in Figure 8. The top two lines reflect the number of bucket accesses per lookup under *Heavy Insertion* model and *Balanced Update* model respectively. We notice increases for both lines. The number of bucket accesses per lookup increases continuously to 1.37 for *Heavy Insertion*, an increase of 25% than the original number. While for *Balanced Update*, the number first increases up to 1.25 and then drops to 1.21, with an increase of 10% in the end. The bottom two lines are rehash percentages of the whole update operations. These two lines give a clear view that if the insertion is heavy, we will come across more rehashes. For *Balanced Update*, the rehash percentage stays almost the same at 0.5%. There is a slight increase in *Heavy Insertion* rehash percentage. Since the rehash percentages for both models are less than 2% and the rehash operation involves keys in no more than two buckets, we believe *d-ghash* is able to handle these rehashes without incurring too much delay.

V. Routing Table Experiment

Finally, we apply our algorithm to a real routing table application. We use five routing tables downloaded from the Internet backbone routers: as286 (KPN Internet Backbone), as513 (CERN, European Organization for Nuclear Research), as1103 (SURFnet, the Netherlands), as4608 (Asia Pacific Network Information Center, Pty. Ltd.), and as4777 (Asia Pacific Network Information Center) [7], with 276K, 291K, 279K, 283K, 281K prefixes respectively after removing the redundant prefixes.

To handle the longest prefix matching problem, hash-based lookup adopts the controlled prefix expansion[19] along with other techniques [20], [21], [22]; it is observed that there are small numbers of prefixes for most lengths, and they can be dealt with separately, for example, using TCAM, while other prefixes are expanded to a limited number of fixed lengths. Lookup will then be performed for those lengths. In this experiment, we expand the majority of prefixes (with lengths in the range of [19, 24]) to two lengths: 22 bits and 24 bits. Assuming the small number of prefixes outside [19, 24] are handled by TCAM, we perform lookups against lengths 22 and 24. Because there are more prefixes of 24-bits long after expansion, we present the results for 24-bit prefix lookup.

There are 159,444 ,159,813, 159,395, 159,173 and 159,376 prefixes of 24-bits long from the five routing tables, respectively. We use these prefixes to setup five tables separately and vary the number of buckets from 100K to 250K with a *target* array of 150K entries.

We find the number of bucket accesses per lookup for *d-ghash* and *d-left* scheme. The results are obtained based on the average of the five tables. As shown in Figure 9, both *base 4-ghash* and *base 2-ghash* performs better than the respective *d-left* scheme. The maximum reduction rate for *base 4-ghash* than *4-left* is about 36% when m = 210,000 and *base 2-ghash* 12% when m = 250,000. The average number of bucket accesses per lookup for *enhanced 4-ghash* scheme is almost one bucket less than *4-left*, with up to 50% reduction. For *enhanced 2-ghash*, there is an average of 20% reduction over *2-left*. We also notice that there is a jump for *4-ghash* at m = 220,000 and another one for *2-ghash* at m = 130,000. This is due to the change in Ω_a, as mentioned before.

In the second experiment, we setup our hash tables with the routing table as286 downloaded at Juanuary 1st, 2010 from [7] and use the collected update trace of the whole month of January, 2010 to simulate the update process. To make experiments simple, we also choose prefixes with the length of 24. There are 159,444 24-bit prefixes in the table. The update trace contains 1,460,540 insertions and 1,458,675 deletions for those 24-bit prefixes. We vary the number of buckets from 110K to 150K. For all these settings, the achievable bucket load Ω_a is 2 for *enhanced 4-ghash*. We also use a fixed 150K-entry *target* array.

TABLE II
ROUTING TABLE UPDATES FOR *enhanced 4-ghash*

Number of buckets	110K	120K	130K	140K	150K
Rehash percentage	re-setup	0.23%	0.13%	0.08%	0.05%

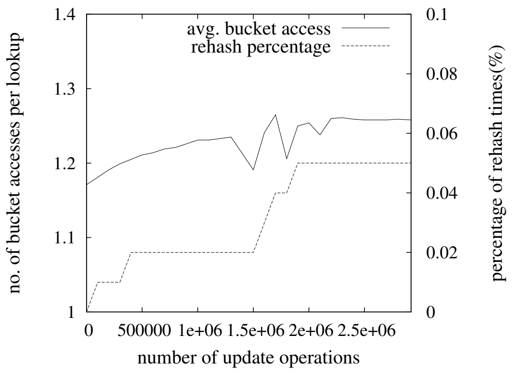

Fig. 10. Experiment with the update trace using *enhanced 4-ghash*. Bucket access per lookup line correspond to the left Y axis. Rehash percentage line correspond to the right Y axis.

As shown in Table II, if we use 110K buckets, we need a re-setup for the whole table. If we use 120K buckets, we don't need a re-setup, but have to rehash 0.23% of the whole update operations, which is about 0.5% of the 1.4 million insertions. And if we increase the number of buckets, we will rehash less. When using 150K buckets, we have close to 0.05% chance of rehash. We also show the lookup efficiency change in Figure 10 with m = 150K. The update trace used has nearly the same number of insertions and deletions, which is similar to a *Balanced Update* model used in Section IV. We can view that the rehash percentage grows continually to 0.05%. The number of bucket accesses per lookup increases and decreases through the update process, with a 7% increase in the end.

VI. Related Work

Using multiple hash functions to place keys in buckets is known to better balance hash table buckets than using a single hash function. There are many studies using multiple hash functions to improve load balancing [1], [5], [9], [11], [14], [15]. The recent proposed Deterministic Hashing [6] uses multiple hash functions of each key to an on-chip intermediate index table where the hashed bucket addresses are saved. This approach incurs space overhead in building the index table and delays due to indirect access through the index table. In addition, it also faces complexity in handling key updates. In [10], an improved approach uses the same intermediate index table to balance the buckets. In contrast to these approaches which go through an intermediate table for balancing the buckets, *d-ghash* allocates keys directly into multiple hashed buckets and then remove duplicate keys to achieve a perfect balance. A *minimal* perfect hash function [13], [23] accomplishes a one-to-one collision-free mapping when the number of buckets matches the number of keys. The delay of searching for the perfect hash function as well as the difficulty in handling the complicated hash table update involved in the perfect hash function itself make it difficult for practical use. The goal of the proposed *d-ghash* is not to create or use a minimal perfect hash function; instead, it accomplishes near-perfect balance among the hash buckets using a small number of simple hash functions.

VII. Conclusion

A new guided multiple-hashing method, *d-ghash* is introduced in this paper. Unlike previous approaches which select the least-loaded bucket to place a key progressively, *d-ghash* achieves global balance by allocating keys into buckets after all keys are placed into buckets d times using d independent hash functions. *d-ghash* calculates the achievable perfect balance and removes duplicate keys to achieve this goal. Meanwhile, *d-ghash* reduces the number of bucket accesses for looking up a key by creating as many empty buckets as possible without disturbing the balance. Furthermore, *d-ghash* uses a table to encode the hash function ID for the bucket where a key is located to guide the lookup and to avoid extra bucket access. Simulation results show that *d-ghash* achieves better balance than existing approaches and reduces the number of bucket accesses significantly.

VIII. Acknowledgment

This work is supported in part by the National Science Foundation under grant CNS 1115548.

References

[1] A. Broder and M. Mitzenmacher, "Using Multiple Hash Functions to Improve IP Lookups," *Proc. IEEE INFOCOM*, 2001.

[2] S. Demetriades, S. C. M. Hanna, and R. Melhem, "An Efficient Hardware-Based Multi-hash Scheme for High Speed IP Lookup," *Proc. IEEE HOTI*, 2008.

[3] H. Song, S. Dharmapurikar, J. Turner, and J. Lockwood, "Fast Hash Table Lookup Using Extended Bloom Filter: An Aid to Network Processing," *Proc. ACM SIGCOMM*, 2005.

[4] Brodnik, A. Munro, and J. I., "Membership in Constant Time and Almost-Minimum Space," *SIAM Journal on computing*, vol. 28, no. 5, pp. 1627–1640, 1999.

[5] P. R. Rodler and F. F., "Cuckoo Hashing," *Journal of Algorithms*, vol. 51, pp. 122–144, 2004.

[6] Z. Huang, S. C. David Lin Jih-Kwon Peir, and S. M. I. Alam, "Fast Routing Table Lookup Based on Deterministic Multi-hashing," *IEEE ICNP 18th Intl. Conf.*, pp. 31–40, 2010.

[7] "Routing Information Service," *http://www.ripe.net/ris*, 2009.

[8] C. Hermsmeyer, H. Song, R. Schlenk, R. Gemelli, and S. Bunse, "Towards 100G packet processing: Challenges and technologies," *Bell Labs Technical Journal*, vol. 14, no. 2, 2009.

[9] S. Lumetta and M. Mitzenmacher, "Using the Power of Two Choices to Improve Bloom Filters," *Internet Mathematics*, vol. 4, no. 1, pp. 17–33, 2007.

[10] Z. Huang, J.-K. Peir, and S. Chen, "Approximately-Perfect Hashing: Improving Network Throughput through Efficient Off-chip Routing Table Lookup," *Proc. IEEE INFOCOM*, 2011.

[11] Y. Azar, A. Broder, A. Karlin, and E. Upfal, "Balanced Allocations," *Proc. 26th ACM Symn. on Theory of Computing*, pp. 593–602, 1994.

[12] R. Sprugnoli, "Perfect hashing functions: a single probe retrieving method for static sets," *Comm. ACM*, 1977.

[13] F. C. Botelho, R. Pagh, and N. Ziviani, "Simple and space-efficient minimal perfect hash functions," *WADS*, 2007.

[14] B. Vocking, "How Asymmtry Helps Load Balancing," *Proc. 40th IEEE Symn. on FCS*, pp. 131–141, 1999.

[15] A. Kirsch and M. Mitzenmacher, "On the Performance of Multiple Choice Hash Tables with Moves on Deletes and Inserts," *Communication, Control, and Computing, 46th Conf.*, 2008.

[16] F. Hao, M. Kodialam, and T. V. Lakshman, “Building high accuracy bloom filters using partitioned hashing,” *Proc. ACM SIGMETRICS*, vol. 35, pp. 277–288, 2007.
[17] B. Bloom, “Space / Time Trade-offs in Hash Coding with Allowable Errors,” *Comm. ACM*, vol. 13, no. 7, pp. 422–426, 1970.
[18] T. Wang, “http://burtleburtle.net/bob/hash/integer.html.”
[19] V.Srinivasan and G. Varghese, “Fast Address Lookups Using Controlled Prefix Expansion,” *ACM Transactions on Computer Systems*, 1999.
[20] S. Dharmapurikar, P. Krishnamurthy, and D. Taylor, “Longest Prefix Matching Using Bloom Filters,” *Proc. ACM SIGCOMM*, 2003.
[21] B. Chazelle, R. R. J. Kilian, and A.Tal, “The Bloomier filter: an efficient data structure for static support lookup tables,” *Proc. ACM 15th SIAM*, 2004.
[22] J. Hasan, S.Cadambi, V.Jakkula, and S. Chakradhar, “Chisel: A Storage-efficient, Collision-free Hash-based Network Processing Architecture,” *ISCA*, 2006.
[23] M. L. Fredman and J. Komlos, “On the Size of Separating Systems and Families of Perfect Hash Functions,” *SIAM. J. on Algebraic and Discrete Methods*, 1984.