A Highly Deterministic Hashing Scheme Using Bitmap Filter for High Speed Networking

Hai Sun
Washington State University
Email: hsun1@eecs.wsu.edu

Yan Sun
Huawei US Research Center
Email: yan.yansun@huawei.com

Victor C. Valgenti
Petabi, Inc.
Email: vvalgenti@petabi.com

Min Sik Kim
Petabi, Inc.
Email: msk@petabi.com

***Abstract*—In modern network devices the query performance for a hashing method degrades sharply due to non-determinism incurred by hash collision. Although previous collision resolution mechanisms have made remarkable progress, there is still much room to improve deterministic performance by resolving hash collisions more effectively. Further, the use of probabilistic, on-chip, summaries such as Bloom filters in these solutions causes a large number of off-chip memory accesses when under heavy network traffic. Even worse, these solutions use separate hash computation for filter queries and hash queries, doubling the computational overhead for hash access.**

We propose two novel collision resolution mechanisms, Double Out hashing and Bidirectional Hop hashing and establish a multiple-segment hash construction to facilitate deterministic query. Collision is restricted to only one segment and the length of a probe sequence in each segment is minimized to 1. In addition an important category of false positive is reduced to 0 due to exact bitmap filters. The use of a unique set of hash functions for filters and hash tables avoids unnecessary computation.

I. Introduction

Hash tables are widely used in various network applications for high-speed packet processing due to $O(1)$ primitive operations of query, insert and delete [1], [2]. However collisions occur frequently when the table occupancy, or load, increases. Newly inserted elements that collide with existing elements are inserted into other buckets, thereby leading to an increase in the length of a probe sequence or collision chain, which is followed during the query [3]. Elements at the tail of a long probe sequence require considerable more probing time than elements close to the head. Owing to non-determinism caused by long probe sequences the query performance in real-time networking applications is vulnerable to adversarial traffic. In modern hardware devices, multiple threads are coordinated to accelerate hash operations and synchronization is compulsory to keep processing order. While synchronization ensures that requests are tackled in order they arrive, it also degrades overall performance to that of the slowest thread determines. As the number of non-deterministic threads increases, the slowest thread tends to turn much slower and the average query performance degrades tremendously.

Among a variety of hash collision resolution approaches, multiple-segment hashing balances the bucket load by reducing the maximum number of keys in a bucket among all buckets. To avoid probing multiple buckets, on-chip summaries such as Bloom filters link keys in multiple buckets. Several multiple-segment hashing systems proposed in a string of papers [1], [3]–[5] remarkably improve deterministic hash operations and achieve good average performance. For instance Peacock hashing [3] limits the length of any collision chain in the table segments to a small constant. Peacock hashing and approaches [1], [4] adopt probabilistic, on-chip filters to reduce off-chip memory accesses and enable deterministic hashing performance. Nevertheless we uncover critical drawbacks in these systems. Firstly collisions are still allowed in each segment; next the length of a probe sequence is not minimized; probabilistic Bloom filters only create approximate summary and cannot eliminate an important category of false positive which invites costly off-chip memory accesses in a great number of high-speed networking applications; finally Bloom filters usually employ a distinct set of hash functions from those for table segments. The hash computation overhead for filter query is worthless to hash table access but never trivial when under heavy network traffic.

Bloom filters used in these multiple-segment hashing systems do not discriminate between two categories of false positive. The first category occurs when a request must match some element in one segment but the match result is wrongly reported in the other segment(s). In other words only when the element is exactly matched in the unique, correct segment, the first category of false positive never happens. The second category occurs when a request shall not match any element but the matching result is wrongly reported in one or more segments. The first category is of great significance to high-speed network applications which do not need to handle the second category. For example, IP address lookup forwards packets using routing tables. In a routing table a rule with lowest priority usually matches any packet which does not match others. For another instance, rule sets used in packet classification algorithms usually contain a rule which matches any packet. A Bloom filter can only reduce but never eliminate the first category. In a high-speed network device which processes millions of packets every second, tens of thousands of packets are mismatched even with a 1% false positive in a multiple-segment hashing system using Bloom filters, leading to costly off-chip memory access. The situation becomes worse as the volume of network traffic increases.

In this paper we establish a multiple-segment hashing system using Double Out hashing, or ***DO*** hashing, and Bidirectional Hop hashing, or ***BH*** hashing. Our hashing construction, called the hierarchy, consists of n segments, or hash tables,

ordered by insert sequence. Each hash table acts as a collision buffer for the tables with higher order and uses a distinct hash function to reduce collision probability. Collision is only allowed in the lowest-order table adopting ***BH*** hashing. Other tables use ***DO*** hashing to avoid collision with the aid of on-chip bitmap filters. Our scheme outperforms previous multiple-segment hashing schemes by:

- Promoting deterministic query performance to facilitate network processing by restricting collision to only one hash table and minimizing the probe sequence to only 1 in each hash table;
- Eliminating a significant category of false positive using innovative on-chip bitmap filters;
- Avoiding unnecessary hash function computation for on-chip filter queries.

II. Related Work

A number of related work discussed various perspectives to improve hash performance for high-speed network applications. The importance of determinism for hardware performance is studied in [3], [6]. Demetriades et al. [6] resolves collision and guarantees deterministic IP lookup by dynamically migrating IP prefixes. Researchers [1], [7] attempt to ensure that a single bucket reference is used for each table lookup with high probability. Simple hash functions [8] are suggested for essentially the same performance as a truly random hash function which causes heavy computation overhead. The Hash Index Table [9] proposes a novel on-chip filter. Huang et al. [5] reduce on-chip memory consumption by storing hashed bucket addresses into an intermediate index table but incur space overhead and delays due to indirect access. Our scheme is influenced by these work and will show determinism can be improved by better resolving collisions in each hash table. Our scheme uses simple hash functions and on-chip bitmap filters for exact summary and quick query.

Some researchers [2], [6], [10], [11] argue multiple-segment hashing is among the best solutions to resolve collision. Our primary starting points are Fast Hash Table (FHT) [1] which summarizes the locations of items in a hash table that uses multiple hash functions, and Peacock hashing [3] that greatly reduces on-chip memory consumption compared to FHT while keeping determinism by limiting the length of a probe sequence in each table to a small constant. Both provide multiple table locations before inserting any new element. To avoid multiple off-chip memory accesses they use probabilistic Bloom filters and hence enable deterministic hashing performance. We will show that our scheme further improves determinism and eliminates drawbacks of the use of Bloom filters.

III. ***DO*** hashing and ***BH*** hashing

A. ***DO*** *hashing*

We invent ***DO*** hashing for highly deterministic query. A Double Out hash table, or a ***DO*** table, T_i $(1 \leq i \leq n-1)$ is an array of buckets. Given an element E its index idx in T_i is computed using $h_i(E)$, T_i's associated hash function. A bucket

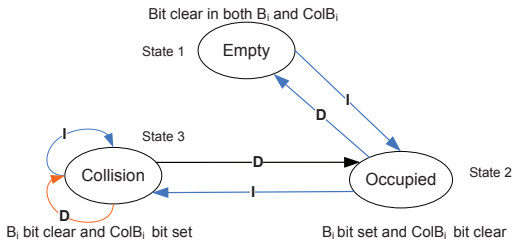

Fig. 1. Three Bucket States

contains at most one element. T_i is associated with three auxiliary structures: a main bitmap filter B_i, a collision table $ColT_i$ and a collision bitmap filter $ColB_i$. B_i maintains an exact summary of all T_i buckets and each B_i bit is associated with a specific T_i bucket. It is set if the bucket contains an element, or clear otherwise. $ColT_i$ keeps references to all the collided elements in T_i. A reference $(j, index)$ to E indicates E's residing table T_j and corresponding index. A $ColT_i$ bucket is associated with a specific T_i bucket and keeps a list of references to collided elements in the T_i bucket. $ColB_i$ maintains an exact summary of all $ColT_i$ buckets and each $ColB_i$ bit is associated with a specific $ColT_i$ bucket. It is set if the list in the bucket is not empty, or clear otherwise.

A ***DO*** table T_i and its auxiliary structures have the same dimension. Furthermore a T_i bucket's index is the same position to access its associated B_i bit, $ColB_i$ bit, and $ColT_i$ bucket. As the only hash function for T_i and its auxiliary structures, h_i is used for index computation.

A bucket state reveals how primitive operations are performed in a ***DO*** table. Fig. 1 demonstrates the state transfer among State 1 (Empty), State 2 (Occupied), and State 3 (Collision). A circle symbolizes a state and a directed arrow expresses a single primitive operation, e.g. D for a delete and I for an insert attempt. A bucket state is marked by a B_i bit and a $ColB_i$ bit collectively, e.g. State 2 by B_i bit set and $ColB_i$ bit clear.

State 1 stands for an empty bucket. When an element is inserted, State 1 is transferred to State 2. State 2 implies that an element resides in the bucket. After deleting the element State 2 is transferred to State 1. When another element is indexed to a bucket in State 2, the collision occurs and State 2 is transferred to State 3. The existing element is deleted and both elements try the next lower-order hash table. The details about table order will be discussed in section IV. A State 3 bucket remains empty and rejects the insert attempt of any new element which has to try the next table. Thus no state transfer occurs in a State 3 bucket for any insert attempt. A delete in a State 3 bucket causes two outcomes. The first outcome does not incur state transfer but the second one does. We will illustrate them later. A ***DO*** table disallows collision and restricts the length of a probe sequence to only 1.

The consequence of inserting an element E to T_j relies on current bucket state.

- State 1: no collision and inserted.

- State 2: encounter collision. Remove the existing element and try both elements in T_{j+1}.
- State 3: still collision and try E in T_{j+1}.

Algorithm 1 states the insert procedure. Re-balancing will be discussed in the delete procedure. All collided elements which fail to be inserted to the ***DO*** table T_{n-1} will try T_n. The insert procedure reveals that an element must reside in only one hash table. Once E is inserted into T_i, its reference is added into each $ColT_j$ $(1 \leq j \leq i-1)$ and the $ColT_j$ bucket's index is computed using $h_j(E)$.

Algorithm 1 Insert Algorithm for ***DO*** Tables

```
FuncDoubleOutInsert(E,i) {i: table number of T_i}
int idx = h_i(E); {calculate index}
Bucket b = T_i.getBucket(idx);
if b in State 1 then
  T_i.insert(idx, E);
  for all j from 1 to i − 1 do
    int newIdx = h_j(E); {recalculate index}
    ColT_j.insert(newIdx,i,idx); {insert reference}
else
  if b in State 2 then
    E_c=T_i.remove(idx); {E_c: existing element}
    FuncDoubleOutInsert(E_c, i + 1); {try E_c in T_{i+1}}
    FuncDoubleOutInsert(E, i + 1); {try E in T_{i+1}}
  else
    FuncDoubleOutInsert(E, i + 1); {b in State 3}
```

Fig. 2 exhibits a simple insert and delete example using two ***DO*** tables consisting of 8 and 6 buckets respectively. In Fig. 2(a) T_1 contains E_1 in b_3 (index as subscript). T_2 is empty and not shown. Next element E_2 collides with E_1 in b_3. Fig. 2(b) demonstrates the collision outcome. $ColT_2$ and $ColB_2$ are not shown. Suppose E_1 and E_2 are inserted into different buckets b_5 and b_0 in T_2. So list L_3 of $ColT_1$ contains their references, $(2,5)$ and $(2,0)$ respectively. B_1's corresponding bit is clear and $ColB_1$'s bit in the same position is set while B_2's corresponding bits are set.

Elements in T_1 can be deleted without involving any collision table. However if an element E is deleted from T_j $(1 < j \leq n)$, its references are removed from corresponding lists of collision tables from $ColT_1$ to $ColT_{j-1}$ and each list resides in the bucket with index of $h_j(E)$. As we discussed in Fig. 1, two outcomes happen after a delete in a State 3 bucket. After removing E's reference if any list contains at least two references, this is the first outcome and no state transfer occurs in any bucket. However if any list contains only one reference, the second outcome happens as shown in Fig. 2(c) and (d). Return to the example and now delete E_1. Fig. 2(c) displays its reference in L_3 of $ColT_1$ is also removed. Since L_3 of $ColT_1$ contains only one reference of E_2, no collision exists in b_3 of T_1 any more and under the circumstance E_2 should be moved back. Fig. 2(d) shows the result of moving E_2 from b_0 of T_2 to b_3 of T_1. Its reference is removed from $ColT_1$. T_2 is empty and not shown. B_1's associated bit with b_3 of T_1 is set and $ColB_1$'s associated bit with L_3 of $ColT_1$ is cleared. The

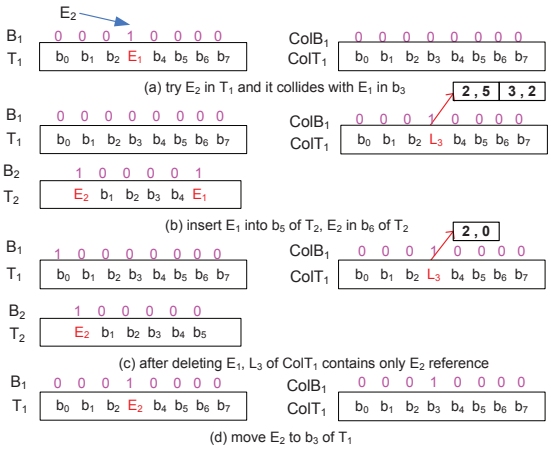

Fig. 2. Insert and Delete in a ***DO*** Table

delete in one bucket, e.g. b_5 of T_2, causes the state transfer at another bucket, e.g. b_3 of T_1, from State 3 to State 2 in Fig. 1.

Re-balancing is defined as a process to achieve load balance among hash tables by moving elements from low-order tables to high-order tables. Through re-balancing an element is moved to the first hash table where it has no collision, e.g. moving E_2 from T_2 to T_1 in Fig. 2(d). Re-balancing also takes effect when one-reference list is generated by discarding one element due to an insert in a State 2 bucket. Re-balancing is of extraordinary significance to maintain global load balance in the hash tables. Without it State 3 buckets in high-order tables cannot preserve new elements, leading to ever-increasing load in low-order tables and eventually high discard rate in T_n.

B. *BH hashing*

We design ***BH*** hashing for T_n, or the ***BH*** table, which allows one collision for each bucket. The first collided element in a bucket may be inserted in a backup bucket through bidirectional searching in a neighborhood area. The backup bucket is called next-hop and current bucket is called prev-hop. Their relation is expressed using a bitmap. An element is discarded if it is indexed to a bucket with next-hop or the bidirectional searching returns a non-empty bucket. The discard rate is the percentage of elements dropped in T_n. If T_n contains more next-hops the discard rate increases. If an element E in a bucket is deleted, the element in the bucket's next-hop needs to be moved to the bucket. This substantially improves T_n load and diminishes the discard rate. In addition E's references in all collision tables are removed.

A binary bitmap is associated with a bucket to encode collision information. We choose the value of k, a tunable parameter, to vary the bidirectional searching distance. Larger k values increase the opportunity for an element to find its next-hop in the bidirectional searching and hence reduce the discard rate in general. Fig. 3(a) depicts a bitmap format with $2k$+3 bits. From left the first bit indicates whether the bucket is empty. The second bit indicates whether the bucket has a next-hop. If it is set, the k-bit next-hop block (index from

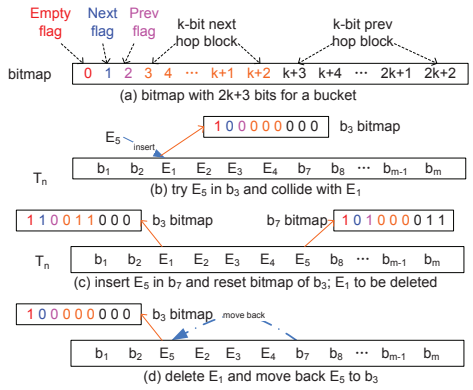

Fig. 3. Bitmap Format, Insert and Delete in the ***BH*** Table

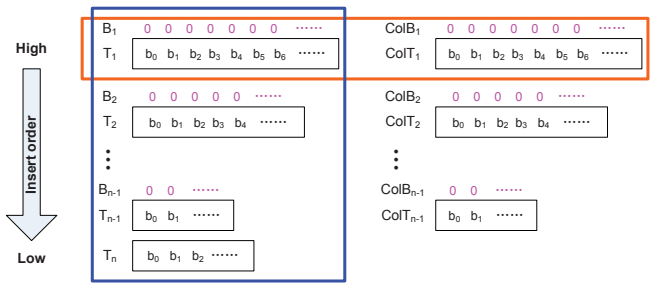

Fig. 4. Hash Table Hierarchy

3 to k+2) encodes the relative distance to the next-hop in binary. The third bit indicates whether the bucket has prev-hop. If it is set, the k-bit prev-hop block (index from k+3 to $2k$+2) encodes the relative distance to prev-hop. In practice the second bit and k-bit next-hop block (or the third bit and k-bit prev-hop block) act as the next pointer (or the previous pointer) in a double-linked list.

The neighborhood of a bucket b_p covers 2^k buckets, forwardly from b_{p+1} to b_{p+2^k-1} and backwardly from b_{p-1} to b_{p-2^k-1}. The bidirectional searching starts forwardly and then backwardly. As soon as an empty bucket is found, the bidirectional searching terminates and returns the bucket. If no empty bucket is found, discard the element.

Fig. 3 displays an example of insert and delete in the ***BH*** table. Assume k=3. T_n contains four elements. b_3 (index as subscript) contains E_1 and has no next-hop. The first bit of b_3 bitmap is set. b_4, b_5 and b_6 contain E_2, E_3 and E_4 respectively and their bitmaps are not shown. In Fig. 3(b) E_5 is indexed to b_3 and collides with E_1. The bidirectional searching begins from b_4 and returns b_7 to which E_2 is inserted since buckets from b_4 to b_6 are not empty. Fig. 3(c) describes the connection between b_3 and b_7. The second bit in b_3 bitmap is set for next-hop and k-bit next-hop block encodes the distance 4 (011 in binary). Likewise the first and third bits in b_7 bitmap are set for non-empty and prev-hop indication, and k-bit prev-hop block encodes the same distance. In Fig. 3(d) E_1 is deleted. E_5 in b_3's next hop b_7 is moved back to b_3. So b_7 is empty and its bitmap is not shown. Eventually only the first bit in b_3 bitmap is set to represent E_5.

By choosing a reasonable k value, a cache line can contain an element and its next-hop. In consequence only one memory access is sufficient to query an element. This eradicates the drawback of cuckoo hashing [12] or linear hashing, i.e. the need to access sequences of unrelated locations on different cache lines.

IV. Hierarchy Design

In our scheme ***DO*** hashing and ***BH*** hashing work collaboratively. All the hash tables except T_n employ ***DO*** hashing for the benefit of fast query using on-chip filters. Collision is disallowed in each ***DO*** table, and the length of a probe sequence is limited to only 1 accordingly. The ***BH*** table adopts ***BH*** hashing, stores the collided elements in all the ***DO*** tables, and allows one collision for each bucket. Only one memory access is needed in the ***BH*** table to look up two associated elements in a cache line.

Fig. 4 depicts the hierarchy framework. The vertical block highlights n number of hash tables. The insert order is the table sequence to insert an element and T_1 is always the first choice to try an element E. If E collides in T_1, try T_2. Continue to try E in each lower-order table along the insert order chain until it is either inserted into a table without collision or discarded if it fails to be inserted in T_n. Thus each hash table except T_1 acts as a collision buffer for higher-order tables. The horizontal block encircles T_1 and its three auxiliary structures, also in every ***DO*** table. Our scheme abides by two fundamental principles.

1) Keep as few elements as possible in T_n.
2) Insert an element to the first hash table with no collision.

Principle 1 enables ***DO*** tables to contain the majority of elements for the sake of fast query using on-chip filters. According to Principle 2 an element is inserted to T_j ($j > 1$) only if it collides in all the higher-order tables, i.e. from T_1 to T_{j-1}. A delete of an element E from T_j ($j > 1$) leads to removal of E's all references in the collision tables from $ColT_1$ to $ColT_{j-1}$. Principle 2 results in element movement due to re-balancing under one-reference list circumstance.

In terms of the framework and principles we need to determine three essential aspects in the hierarchy. Firstly is there an optimal load for a ***DO*** table and what is relation between load factor and discard rate in the ***BH*** table? Secondly what is an ideal number of hash tables in the hierarchy and how to assign table dimension given an input set? Finally how to query an element using filters and tables? Our purpose is to achieve balance between memory consumption, discard rate and query efficiency. In the remaining paper c_i denote the dimension of a hash table. m and m_i represent the numbers of elements in all hash tables and T_i, or the i^{th} table, respectively.

A. Load Analysis

An element is discarded if indexed to a T_n bucket with next-hop or the bidirectional searching returns a non-empty

bucket. Therefore a larger T_n load gives rise to more next-hops and a larger discard rate. We discover that at 38% load the discard rate is about 1% and all the discarded elements are only attribute to next-hop collision. The bidirectional searching always returning empty buckets even with a fairly small k value, e.g. 2.

Formally S_w represents the number of buckets in a ***DO*** table T_i under State w ($1 \leq w \leq 3$). So $\sum_{w=1}^{3} S_k = c_i$ wherein c_i is T_i's dimension. Suppose an element has an equal probability to be indexed in any bucket. Equation [1] counts the number of State 2 buckets in T_i where r_i is the number of elements to be inserted.

$$S_2(r_i) = r_i \times \left(1 - \frac{1}{c_i}\right)^{r_i - 1} \tag{1}$$

ρ_i is defined as the ratio of input elements to the number of T_i buckets and calculated by $\frac{r_i}{c_i}$. Equation [1] derives Equation [2] where $f(\rho_i)$, a function of ρ_i and c_i, signifies T_i's load. The optimal value of $f(\rho_i)$ is proven in Lemma 1.

$$f(\rho_i) = \rho_i \times \left(1 - \frac{1}{c_i}\right)^{\rho_i \times c_i - 1} \tag{2}$$

Lemma 1. $\rho_B = e^{-1}$ at $\rho_i = 1$ when $c_i \to \infty$. e is base of natural logarithm.

Proof. To reach the peak value of $f(\rho_i)$, $\frac{\mathrm{d}f(\rho_i)}{\mathrm{d}\rho_i} = 0$. Equation [3] is derived from Equation [2].

$$(1 - \frac{1}{c_i})^{\rho_i \times c_i - 1} + c_i \times \rho_i \times (1 - \frac{1}{c_i})^{\rho_i \times c_i - 1} \times \ln(1 - \frac{1}{c_i}) = 0 \tag{3}$$

Equation [4] is thus obtained from Equation [3].

$$\rho_i = \frac{-1}{c_i \times \ln\left(1 - \frac{1}{c_i}\right)} \tag{4}$$

Now $\lim_{c_i \to \infty} \left(1 - \frac{1}{c_i}\right)^{c_i} = e^{-1}$ when $c_i \to \infty$ and $\rho_i = 1$. Eventually $f(\rho_i)$ reaches its peak value at ρ_B in Equation [5] where ρ_B is defined as the ρ_i value (100%) leading to optimal load. In conclusion when the dimension of a ***DO*** table equals the input size, its optimal load is $e^{-1}, \approx 36.79\%$.

$$f(1) = \lim_{c_i \to \infty} \left(1 - \frac{1}{c_i}\right)^{c_i - 1} = e^{-1} \tag{5}$$

B. Dimension Assignment

On basis of Equation [5] we propose a hierarchy to obtain optimal load in all ***DO*** tables. Given an input set with size m, c_1 is set to be m to meet $\rho_B = 1$ and $m_1 = c_1 \times \rho_B$. So T_1 has a theoretical optimal load. The number of collided elements in T_1 is $c_1 \times (1 - \rho_B)$, used as the input to T_2. Therefore c_2 is set to be $c_1 \times (1 - \rho_B)$ and $m_2 = c_2 \times \rho_B$. The remaining elements are used as the input to T_3. Repeat the dimension assignment strategy until in some point assign the remaining elements to c_n. Equation [6] summarizes the strategy.

$$c_i = m \times (1 - \rho_B)^{i-1}; m_i = m \times (1 - \rho_B)^{i-1} \times \rho_B \tag{6}$$

According to Principle 1 we use a tunable parameter β to determine the upper-bound percentage of elements assigned to T_n and the parameter directly determines the total number of hash tables. If $\frac{c_i - m_i}{m} < \beta$ we set c_n. Recall that discard rate is around 1% at 38% T_n load. Thus $c_n = \frac{c_{n-1} - m_{n-1}}{0.38}$ and the number of buckets in all the hash tables is approximately $m \times e$.

TABLE I states an 8-table hierarchy given m=100000, β=5% and ρ_B=36.79%. c_1 is exactly m. $m_1 = m \times \rho_B$ or 36790. Use $c_1 - m_1$ as input to T_2 and c_2=63210. Continue until $\frac{c_7 - m_7}{m} < \beta$ and $c_8 = \frac{c_7 - m_7}{0.38}$, or 10954. If β increases, the number of hash tables decreases. For example under 7% value of β, the number of hash tables is 7 and the last table T_7 has a dimension c_7 of 17331. In case that β=7%, the number is 6 with c_6=27421. Though the number of hash tables varies along with different β, the optimal load is maintained in each ***DO*** and the hashing performance is thus not affected. For the purpose of incremental update a smaller constant than ρ_B is more practical for real implementation.

C. Query

To query an element E, each index idx_i ($1 \leq i \leq n-1$) for a ***DO*** table T_i is calculated using $h_i(E)$. Firstly all the bits at idx_i position of main and collision filters are inspected in on-chip memory. Next a hash table access is taken in main memory only when the following two bit patterns are matched.

1) No positive response from main filters. Each B_i bit is clear and each $ColB_i$ bit is set.
2) At least one positive response from main filters. Suppose T_j is the highest-order ***DO*** table whose associated B_j contributes a positive response. So each B_l ($1 \leq l \leq j-1$) bit is clear and the B_j bit is set while each $ColB_l$ bit is set and the $ColB_j$ bit is clear.

If pattern 1 fires, access a corresponding T_n bucket and its next-hop if necessary. If pattern 2 fires, access T_j bucket in idx_j. In case of other bit patterns no hash table is accessed as E does not match any element.

Fig. 5 elaborates how to query four elements, E_0 to E_3. Suppose their indexes are 0, 1, 2 and 3 respectively in all the hash tables. Actually it is impossible for an element to have the same index in each hash table. We simply make the assumption for introducing the example conveniently. Firstly all the filters are inspected. For E_0, the bits at index 0 are all clear in the main filters and all set in the collision filters. Therefore pattern 1 matches. For E_1, pattern 1 does not match since the bit at index 1 of $ColB_j$ is not clear. For E_2, pattern 2 does not match since the bit at index 2 of $ColB_1$ is not clear. For E_3, each B_l ($1 \leq l \leq j-1$) bit is clear and the B_j bit is set while each $ColB_l$ bit is set and the $ColB_j$ bit is clear. Therefore pattern 2 matches. Consequently E_1 and E_2 match no element. Next T_n is accessed to query E_0 using $h_n(E_0)$ and T_j is accessed for E_3.

Throughout query procedure ***DO*** hashing guarantees no collision in the majority of hash tables and ***BH*** hashing ensures a single cache-line access. Query is thus performed in a highly deterministic manner because only one hash access is needed

TABLE I
8-TABLE HIERARCHY

c_i	c_1	c_2	c_3	c_4	c_5	c_6	c_7	c_8	$c = \sum_{i=1}^{8} c_i$
Table Dimension	100000	63210	39954	25254	15962	10088	6376	10605	271449

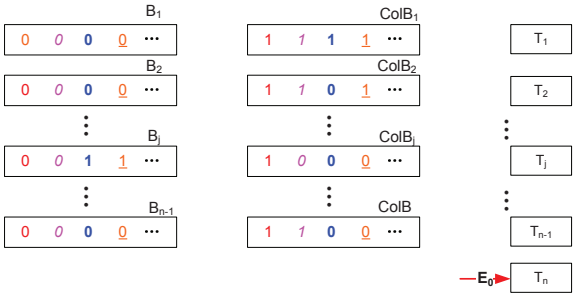

Fig. 5. Query Example

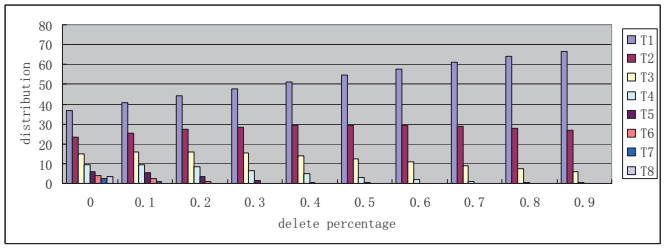

Fig. 6. Rebalancing Consequence

for any query and the length of a probe sequence in each table is minimized to 1. Additionally the first category of false positive is reduced to 0 and unnecessary hash function computation is avoided for filter queries. Actually queries in filters are independent and can be paralleled. Our scheme is highly qualified for hardware implementation accordingly.

V. SIMULATION

The simulation is performed in a commodity computer to evaluate several significant performance metrics including re-balancing influence, update efficiency and memory usage. The advantages and disadvantages of the use of on-chip filters in our scheme are compared with FHT and Peacock hashing. All the experiments in the simulation are conducted upon the proposed 8-table hierarchy using input sets from real IP routing tables [13]. Each input set has 100,000 elements randomly chosen from a routing table after eliminating duplication and each experiment is repeated 10000 times per input set. Through experiments we discover that the average ***DO*** table load is nearly optimal, consistent with theoretical analysis. In addition the first category of false positive is exactly 0. The bit pattern of each queried element exactly matches either pattern 1 or pattern 2.

We choose two groups of hash functions. The first group only uses CRC32 with different feed length for each hash table. The second group consists of eight various message digest functions such as SHA-256. The experimental results using the two groups show that there is no difference in two crucial performance metrics, the average lengths of the collision lists and the global distribution of elements in all the hash tables. Consequently the hash function choice is independent from the hash performance.

A. Re-balance Influence

We observe the re-balancing influence through the distribution of remaining elements after deleting a percentage of elements, randomly chosen from all the hash tables. The percentage varies from 0% to 90%. Distribution ratio is formulated as $\frac{m_i}{m}$ $(1 \leq i \leq n)$ where m_i is the number of elements in a table and m is the number in all the tables after re-balancing.

Fig. 6 accounts for the re-balancing influence. Vertical axis and horizontal axis represent the distribution ratio and the percentage of deleted elements. Each column points out a table's distribution ratio. As the delete percentage increases, we observe three phenomena. Firstly the distribution ratios of low-order tables becomes 0 as a result of element movement towards high-order tables. For example only a few elements remain in T_8 after 10% percentage deletes. In tables from T_4 to T_8 distribution ratios decrease monotonically. Secondly T_1's distribution ratio monotonically increases. Thirdly the distribution ratios of T_2 and T_3 increase and then decrease. For example at delete percentages less than 50% elements moved from lower-order tables are more than elements deleted out of T_2. Thus T_2's distribution ratio increases. Otherwise the ratio decreases as lower-order tables are almost empty and the outgoing elements overpasses the incoming elements in T_2. These phenomena explain how re-balancing facilitates global load balance by moving elements to high-order tables.

B. Update Efficiency

Update operations involve bitmap filters, hash tables and collision tables. An insert attempt in a ***DO*** table may cause collision and hence an existing element is deleted from the table. The chaining reaction due to one insert may be propagated to lower-order hash tables. In the worst scenario $O(2^{n-1})$ elements may be deleted owing to an insert and more deletes occur in collision tables. Nevertheless the average update performance is much better in the simulation. Averagely only one element is deleted from the hash tables owing to an insert.

Meanwhile averagely 5 insert and delete operations occur in collision tables per hash table insert. On the other hand a delete may involve element movement to high-order tables due to re-balancing. Though the worst delete case is difficult to analyze, average delete efficiency can be evaluated by the average number of moved elements per delete. Averagely 2 elements per delete are moved and 4 insert and delete operations occur in collision tables per hash table delete.

Our scheme outperforms FTH and Peacock hashing in on-chip filter update because only 2 bits, one in a main filter and the other in a collision filter, represent an element while 10 bits represent an element in Bloom filters used by FHT and Peacock hashing. Thus a limited number of off-chip and extremely few on-chip memory accesses per update operation justifies the capability of incremental update in our scheme.

C. Memory Evaluation and Filter Comparison

A reference in a collision table occupies a few bytes in real data structures. By comparison a hash table element usually consists of dozens, or even hundreds, of variant bytes and hence hash tables consume the majority of main memory. In the proposed hierarchy the total amount of buckets is approximately $m \times e$ where m is the input size and e is base of natural logarithm. The main memory consumption in our scheme is moderate concerning less than three folds of input size, reasonable for deployment in network processors which nowadays are equipped with plenty of main memory.

TABLE II
COMPARISON RESULTS

Scheme	1^{st}FP	On-chip Mem Usage	Filter Hash Function
FHT	1%	$10 \times m$	separate
Peacock	1%	m	separate
Ours	0	$2e \times m$	identical

TABLE II compares our scheme with Peacock hashing [3] and FHT [1] in three on-chip filter metrics. 1^{st}FP represents the first category of false positive, 0 in our scheme. The other two schemes use Bloom filter with nearly 1% false positive rate. There is no false negative in all schemes. Concerning the on-chip memory usage, to restrict the false positive to 1% a 10-bit, on-chip memory is used for each element and hence FHT uses $10 \times m$ bits wherein m is the input size. Peacock hashing utilizes m bits because about 10% of elements are represented using filter bits. In our scheme all filters use a total of $2e \times m$ bits, e.g. less than 1MByte on-chip memory given an input size of one million. Our scheme employs a unique set of hash functions for all the filters and tables. In the other schemes the set of Bloom-filter hash functions varies from the set of hash functions for hash tables. Consequently our scheme eliminates the doubled computation overhead. Further, bitmap filter update is more efficient in our scheme as discussed before. In a high-speed network device, e.g. a core router, which processes billions of packets per second, millions of packets are mismatched even with a 1% false positive, leading to considerable off-chip memory access. As well the extra computational overhead for separate filter hash function can never be considered trivial. The situation becomes even worse as the volume of network traffic rapidly increases.

Though our scheme consumes more on-chip memory than Peacock hashing, the novel design of exact summary using bitmap filters eliminates the first category of false positive and avoids unnecessary hash computation for filter query. The trade-off is particularly valuable for numerous high-speed network applications to accelerate packet processing. On-chip memory consumption is easily satisfied by modern hardware devices such as network processors. Despite false positives in the second category, only one memory access is needed in our scheme and hence high deterministic performance is maintained.

VI. CONCLUSION

By means of novel collision resolution mechanisms high deterministic performance is achieved in our scheme. An important category of false positive is reduced to 0 and unnecessary computation for filter query is avoided. Through mathematical reasoning we propose a hierarchy with optimal load in each hash table and low discard rate. Simulation results adequately justify the efficiency of re-balancing and incremental update. The comparison to other schemes distinguishes effective use of the exact-summary, bitmap filter in our scheme. In conclusion our scheme is highly qualified for hardware implementation to boost network processing. Future work intends to focus on rehashing to accommodate more frequent updates and multiple next hops in a ***BH*** table bucket.

REFERENCES

[1] H. Song, S. Dharmapurikar, J. Turner, and J. Lockwood, "Fast hash table lookup using extended bloom filter: An aid to network processing," in *IEEE Sigcomm*, 2005, pp. 181 – 192.

[2] A. Broder and M. Mitzenmacher, "Using multiple hash functions to improve ip lookups," in *IEEE Infocom*, 2001, pp. 1454 – 1463.

[3] S. Kumar, J. Turner, and P. Crowley, "Peacock hashing: Deterministic and updatable hashing for high performance networking," in *IEEE Infocom*, 2008, pp. 556 – 564.

[4] S. Kumar and P. Crowley, "Segmented hash: An efficient hash table implementation for high performance networking subsystems," in *IEEE/ACM ANCS*, 2005, pp. 91 – 103.

[5] Z. Huang, D. Lin, S. Chen, J. Peir, and I. Alam, "Fast routing table lookup based on deterministic multi-hashing," in *IEEE ICNP*, 2010, pp. 31 – 40.

[6] S. Demetriades, M. Hanna, S. Cho, and R. Melhem, "An efficient hardware-based multi-hash scheme for high speed ip lookup," in *IEEE HOTI*, 2008, pp. 103 – 110.

[7] X. Tao, Y. Qiao, J. K. Peir, S. Chen, Z. Huang, and S. L. Lu, "Guided multiple hashing: Achieving near perfect balance for fast routing lookup," in *IEEE ICNP*, 2013, pp. 1 – 10.

[8] A. Kirsch, M. Mitzenmacher, and G. Varghese, "Chapter 9 hash-based techniques for high-speed packet processing," in *Algorithms for Next Generation Networks*, 2010, pp. 207 – 208, 212 – 213.

[9] Z. Huang, J. K. Peir, and S. Chen, "Approximately-perfect hashing: Improving network throughput through efficient off-chip routing table lookup," in *IEEE Infocom*, 2011, pp. 311 – 315.

[10] Y. Kanizo, D. Hay, and I. Keslassy, "Optimal fast hashing," in *IEEE Infocom*, 2009, pp. 2500 – 2508.

[11] A. Kirsch and M. Mitzenmacher, "Simple summaries for hashing with choices," *IEEE ToN*, vol. 16, pp. 218 – 231, 2008.

[12] R. Pagh and F. F. Rodler, "Cuckoo hashing," in *9th Annual European Symposium on Algorithms*, 2001, pp. 121 – 133.

[13] R. project, "www.routeviews.org," in *routing table and packet trace*, 2014.