Journal of Network and Computer Applications 36 (2013) 657–666

Contents lists available at SciVerse ScienceDirect

Journal of Network and Computer Applications

journal homepage: www.elsevier.com/locate/jnca

Fast and deterministic hash table lookup using discriminative bloom filters

Kun Huang [a,*], Gaogang Xie [a], Rui Li [b], Shuai Xiong [b]

[a] Institute of Computing Technology, Chinese Academy of Sciences, Beijing, China
[b] School of Information Science and Engineering, Hunan University, Changsha, China

ARTICLE INFO

Article history:
Received 9 December 2011
Received in revised form
20 November 2012
Accepted 21 December 2012
Available online 8 January 2013

Keywords:
Network processing
Hash table
Bloom filter
Cuckoo hashing
Collision-free hashing

ABSTRACT

Hash tables are widely used in network applications, as they can achieve O(1) query, insert, and delete operations at moderate loads. However, at high loads, collisions are prevalent in the table, which increases the access time and induces non-deterministic performance. Slow rates and non-determinism can considerably hurt the performance and scalability of hash tables in the multi-threaded parallel systems such as ASIC/FPGA and multi-core. So it is critical to keep the hash operations faster and more deterministic.

This paper presents a novel fast collision-free hashing scheme using Discriminative Bloom Filters (DBFs) to achieve fast and deterministic hash table lookup. DBF is a compact summary stored in on-chip memory. It is composed of an array of parallel Bloom filters organized by the discriminator. Each element lookup performs parallel membership checks on the on-chip DBF to produce a possible discriminator value. Then, the element plus the discriminator value is hashed to a possible bucket in an off-chip hash table for validating the match. This DBF-based scheme requires one off-chip memory access per lookup as well as less off-chip memory usage. Experiments show that our scheme achieves up to 8.5-fold reduction in the number of off-chip memory accesses per lookup than previous schemes.

1. Introduction

Hash tables are a versatile data structure for fast lookups that associates a set of keys to a set of values. Hash tables can achieve constant $O(1)$ average memory accesses of query, insert, and delete operations at moderate loads. Due to the excellent average-case performance, hash tables have found widespread applications in networking, such as load balancing (Azar et al., 1994; Vocking, 1999), per-flow state management (Estan and Varghese, 2002; Estan et al., 2004), IP route lookup (Border and Mitzenmacher, 2001; Dharmapurikar et al., 2003), packet classification (Srinivasan et al., 1999; Baboescu and Varghese, 2001), deep packet inspection (Dharmapurikar et al., 2004), etc. Most of these applications are typically deployed in critical data paths of high-speed routers/switches. Hence, hash tables must provide a better performance in both average and worst cases for high-speed network applications.

1.1. Motivation

At high loads, collisions are prevalent in the hash table, which increases the access time and induces non-deterministic performance. As the table load increases, collided elements may lead to an increase in the length of the probe sequence. Long probe sequences can increase the cost of the primitive operations and make the lookups non-deterministic, which in turn degrades average performance. The well-known collision resolution policies have been proposed to maintain good average-case performance. Nevertheless, at high loads and frequent collisions, the worst-case performance degrades shapely and becomes highly non-deterministic.

Especially, non-determinism can considerably hurt the performance and scalability of hash tables in the multi-threaded parallel systems such as ASIC/FPGA and multi-core. In such systems, multiple threads are exploited to hide the memory access latency for handling high-speed packets in parallel. Each thread performs the hash table lookup using the same algorithm, but has the different lookup time due to the non-determinism. The slowest thread becomes a bottleneck and determines the overall throughput of these systems. Hence, it is critical to keep the hash operations faster and more deterministic.

Due to large memory requirements, hash tables are often not stored in small high-speed memory (e.g. on-chip SRAMs), but in slow off-chip DRAMs. In order to achieve high speeds and determinism, it is viable to minimize the memory and bandwidth requirements of hash tables. The key to fast and deterministic hash table lookup is as follows. First, a compact data structure is imperative to fit in limited on-chip SRAMs, improving the worst-case performance of off-chip hash tables. Second, a query

* Corresponding author. Tel.: +86 010 6260 0710.
E-mail addresses: huangkun09@ict.ac.cn, kylehuang@163.com (K. Huang), xie@ict.ac.cn (G. Xie), hunan.rui@gmail.com (R. Li), shuaixiong1987@gmail.com (S. Xiong).

1084-8045/$ - see front matter © 2013 Elsevier Ltd. All rights reserved.
http://dx.doi.org/10.1016/j.jnca.2012.12.031

operation requires a single off-chip memory access to hash tables with no collisions. Finally, it is desirable to improve the space utilization of off-chip hash tables, leading to low memory overhead.

1.2. Prior art

Multiple-choice hashing (Azar et al., 1994; Vocking, 1999; Border and Mitzenmacher, 2001) is a simple and efficient technique, which places each element in one of $d \geq 2$ possible buckets of the hash table. Such scheme can ensure a more even distribution of elements among all the buckets than traditional schemes using a single hash function, which helps to reduce the average-case and worst-case costs of hash tables.

Recently, several multiple-choice hashing schemes (Song et al., 2005; Kirsch and Mitzenmacher, 2008a,b; Kumar and Crowley, 2005; Kumar et al., 2008; Kirsch and Mitzenmacher, 2008a,b; Yu and Mahapatra, 2008) have been proposed to improve the worst-case performance. As memory access is very expensive and scarce, these schemes leverage a small summary in on-chip memory to significantly reduce off-chip memory accesses to an underlying multiple-choice hash table. Bloom filters (Bloom, 1970) are used to represent the summary as they are simple space-efficient data structures for fast membership query. On-chip Bloom filters can filter out most of unnecessary off-chip accesses, achieving better lookup performance. However, these schemes have the limitations of non-determinism and non-randomness. First, they cannot completely eliminate the collisions, which incur unpredictable lookup rate and complicated queuing, without guaranteeing deterministic worst-case performance. Second, they weaken the randomness as an element is hashed in a small sub-table not in the whole hash table. Due to these drawbacks, theses schemes suffer from high collisions, low space utilization, and load imbalance.

Collision-free hashing is recognized as a promising way to combat the non-determinism and non-randomness. This scheme hashes an element to a unique bucket in the hash table without any collision. The literatures (Kumar et al., 2007; Ficara et al., 2009) have proposed a collision-free hashing scheme that is a variant of multi-choice hashing. This scheme allows an element to contain a few additional c bits called discriminator, and maps the element plus its discriminator by a single hash function to a possible bucket. It is one-to-one perfect hashing. As a discriminator has 2^c possible values, each element has up to 2^c possible bucket choices for constructing a collision-free hash table. Due to the collision-free nature, this scheme has the advantages of high space utilization and deterministic lookups.

Unfortunately, the collision-free hashing scheme has the issue to be addressed which is how to quickly identify the discriminator value for a queried element. This scheme needs at least 2^c memory accesses per lookup to check for each query, incurring low throughput and large bandwidth requirements. There is a natural and simple solution, where a discriminator table stored in on-chip memory is indexed directly with the off-chip elements. But this solution causes memory inefficiency as there are a large number of elements that are hard to fit in limited amounts of on-chip memory. Hence, it is critical to design a fast and memory-efficient discriminator table in on-chip memory for improving high speeds and determinism while retaining collision-free lookups.

1.3. Our approach

In this paper, we propose two approaches to constructing an efficient discriminator table for achieving fast and deterministic hash table lookup. We first develop a direct collision-free hashing scheme. This scheme directly uses a single Bloom filter to construct a discriminator table, where all elements in an off-chip hash table are inserted. It can eliminate most of unnecessary off-chip memory accesses and enhance collision-free lookup performance. But it cannot identify one of 2^c discriminator values for a queried element that belongs to the hash table, resulting in 2^c memory accesses.

To address this drawback, we develop a fast collision-free hashing scheme using Discriminative Bloom Filters (DBFs). DBF is essentially composed of an array of parallel Bloom filters organized by the discriminator. It is stored in on-chip memory, which can not only filter out irrelevant off-chip memory accesses but also identify a possible discriminator value for a queried element. To validate the match, we use the discriminator value plus the element identifier as input to a hash function to calculate and check a possible bucket. Using on-chip DBF, this scheme performs a single memory access per lookup, instead of 2^c memory accesses per lookup. We also explore two network applications of the DBFs, including IP route lookup and deep packet inspection. We show analytically and experimentally that the DBF-based scheme outperforms previous schemes in terms of the number of memory accesses per lookup.

1.4. Key contributions

This paper makes the main contributions as follows:

- We propose a novel DBF to accelerate collision-free hash table lookup. DBF consists of an array of parallel Bloom filters organized by the discriminator. The on-chip DBF identifies a possible discriminator value for each query, leading to a single off-chip memory access per lookup.
- We explore two network applications of the DBFs, including IP route lookup and deep packet inspection, for enhancing high-speed packet processing.
- Experiments show that compared to previous schemes, the DBF-based scheme achieves significant reduction in the number of off-chip memory accesses per lookup by up to 8.5 times.

The rest of this paper is organized as follows. Section 2 overviews the background on Bloom filters and collision-free hash table. Section 3 presents the related work. We describe the direct and fast collision-free hashing schemes, and explore two network applications of the DBFs in Section 4. Section 5 reports experimental results. Finally, Section 6 concludes.

2. Background

2.1. Bloom filters oveview

Bloom filters are simple space-efficient randomized data structures for representing a set to support fast approximate membership queries. A Bloom filter represents a set $S = \{x_1, x_2, \cdots, x_n\}$ of n elements by using a bit vector of m bits, initially all set to 0. A Bloom filter uses k independent random hash functions $h_1, \cdots, h_k$ with the range $\{1, \cdots, m\}$, each of which hashes an element to a random number uniform over the range. For each element $x \in S$, the bits $h_i(x)$ are set to 1 for $1 \leq i \leq k$. To query whether an element y is in the set S, we check whether all $h_i(y)$ are set to 1. If not, then clearly y is not a member of S. If all $h_i(y)$ are set to 1, we assume that y is in S with a certain probability. Hence a Bloom filter may yield a false positive.

The false positive probability of a Bloom filter is given as

$$f = \left(1 - \left(1 - \frac{1}{m}\right)^{nk}\right)^k \approx \left(1 - e^{-kn/m}\right)^k \quad (1)$$

where n is the number of elements in the set S, m is the size of the bit vector, and k is the number of hash functions. For a given ratio of m/n, the false positive probability can be reduced by increasing the number k of hash functions. When $k=(m/n)\ln 2$, the false positive probability is minimized as $f \approx (1/2)^k$ (Broder and Mitzenmacher, 2004). For most applications, the space savings of the Bloom filter often outweigh the false positive error.

A standard Bloom filter allows for easy insertion, but not deletion. Deleting elements from the Bloom filter cannot be done simply by changing ones back to zeros since each bit may correspond to multiple elements. A Counting Bloom Filter (CBF) (Fan et al., 2000) has been proposed to allow for insertions and deletions of elements. CBF uses an array of m counters instead of bits. Counters are incremented on an insertion and decremented on a deletion. The counters are used to track the number of elements currently hashed to the corresponding locations. To avoid counter overflow, four bits per counter have been shown to suffice for most applications. Due to using counters of four bits, CBF blows up the memory requirements by a factor of four over the standard Bloom filter, even though most counters are zero.

2.2. Collision-free hash table overview

A Collision-free Hash Table (CHT) has been proposed in the literatures (Kumar et al., 2007; Ficara et al., 2009). CHT is a variant of multiple-choice hash table. It is organized as follows. Assume that there are n distinct elements $x_1, \cdots, x_n$ and M buckets ($M \geq n$) in the hash table. Each element x_i has a c-bit discriminator that has total 2^c possible values. A single hash function g is used to hash x_i plus its discriminator values to 2^c possible buckets, in each of which x_i is placed without any collision, achieving one-to-one perfect hashing. When an element y is queried, the same hash function g is used to hash y plus each of 2^c discriminator values to 2^c possible buckets, performing 2^c checks in parallel for finding the exact match. Hence, CHT requires 2^c memory accesses per lookup in worst-case.

Fig. 1 illustrates an example of CHT. There are five elements in the element table and eight buckets in CHT, and each element has a 1-bit discriminator. Each entry in the element table contains a node identifier, node key, and all possible hash values. A node key plus a discriminator value is used as input to a single hash function g for producing a random number over the range $\{0, \cdots, 7\}$. As shown in Fig. 1, elements 0, 1, and 2 use the discriminator value 0 to select buckets 3, 4, and 0, respectively, while elements 3 and 4 use the discriminator value 1 to select buckets 6 and 2, respectively. Each hash table entry contains a pair of {*Node ID*, *Node Key*}. For a query of an element CHT in Fig. 1 needs to check two possible table buckets in parallel for the match.

Element Table

Node ID	Node Key	Hash
0	abc	g(abc, 0)=3 g(abc, 1)=0
1	bcd	g(bcd, 0)=4 g(bcd, 1)=2
2	cde	g(cde, 0)=0 g(cde, 1)=4
3	def	g(def, 0)=3 g(def, 1)=6
4	efg	g(efg, 0)=7 g(efg, 1)=2

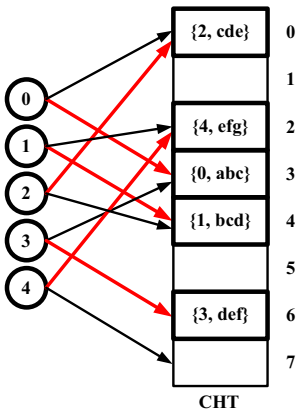

Fig. 1. An example of CHT.

The CHT construction is essentially reduced to a bipartite graph matching problem (Kumar et al., 2007; Ficara et al., 2009). A bipartite graph contains elements as the left nodes, buckets as the right nodes, and bucket choices as edges connecting the left to the right. In this paper, we employ the Cuckoo hashing scheme (Pagh and Rodler, 2004; Friez et al., 2009) to construct a CHT. This scheme adopts a random-walk approach to find a perfect matching in the bipartite graph. We assume that each element is inserted in the table one by one and hashed to 2^c possible buckets. If one of these buckets is empty, an element is placed in the bucket. If not, one of the elements in these buckets is displaced and moved to another of its 2^c possible buckets. The element in turn displaces other elements out of its possible buckets in a similar manner. Otherwise, the evicted element is moved to an overflow list. As this scheme requires a sequence of $O(\log n)$ moves until no further evictions are needed, the CHT construction has the time complexity of $O(\log n)$, scaling well to a large set of elements.

3. Related work

There is a considerable amount of work on fast hashing schemes. The earlier studies (Azar et al., 1994; Vocking, 1999) suggest that the collisions can be reduced significantly with two choices. Using multiple hash functions has been considered to obtain good hash tables. The d-left hash table (Border and Mitzenmacher, 2001) is a well-known multiple-choice hash table. The hash table is divided into d sub-tables, and each element is hashed to d possible buckets, and the left-most least load bucket is selected to place the element. Recently, several multiple-choice hashing schemes have been proposed for high-speed network applications.

Fast Hash Table (FHT) (Song et al., 2005) is proposed to reduce d parallel lookups for an element to one single lookup. Each element is stored in an off-chip shortest linked list, and an on-chip CBF is used as a summary to indicate one linked list used for the search. Alternative scheme (Kirsch and Mitzenmacher, 2008a,b) is proposed to improve the FHT performance. This scheme uses an off-chip multilevel hash table and an array of parallel Bloom filters to implement the same functionality of the FHT, leading to less space for both the summary and underlying hash table.

Segmented hashing (Kumar and Crowley, 2005) is a simple variant of d-ary scheme. The hash table is divided into d equal-sized sub-tables, and each element is stored in a single bucket from one sub-table. This scheme uses an on-chip Bloom filter for each sub-table to avoid d off-chip memory accesses. Peacock hashing (Kumar et al., 2008) has recently been proposed to reduce on-chip memory space. This scheme employs a large main sub-table and a set of small sub-tables, and uses an on-chip Bloom filter for each small sub-table to achieve deterministic hash table lookup.

Cuckoo hashing (Pagh and Rodler, 2004) is a simple multiple-choice scheme that allows element moves. For insertion, each element is hashed to d possible buckets, and then kicks other elements away until every element is moved to its appropriate bucket. The insertion time has $O(\log n)$ moves for $d=2$, and this scheme can achieve about less than one half occupancy. For $d=3$, this scheme can achieve about 90% occupancy, but the upper bounds on insertion time have proven more difficult for $d \geq 3$ (Friez et al., 2009). The improved scheme (Kirsch and Mitzenmacher, 2008a,b) has been proposed to allow at most an element to be moved during an insertion, improving the substantial space utilization of a multiple-choice hash table.

These schemes above are orthogonal to ours. Different from these schemes using a single Bloom filter for a hash table or a small sub-table, our DBF-based scheme employs an array of Bloom filters for a hash table, reducing unnecessary off-chip memory accesses. We can incorporate the DBFs into these schemes to further improve the hash table lookup performance.

4. Discriminative bloom filters

For the purpose of clarity, we describe two collision-free hashing schemes incrementally in this section. First, we present a direct approach using a single Bloom filter. Second, we present a fast approach using Discriminative Bloom Filters (DBFs) that is composed of an array of parallel Bloom filters. Finally, we explore two network applications of the DBFs for high-speed packet processing.

4.1. Direct apporach using a single bloom filter

We now present a direct collision-free hashing scheme that directly uses a single Bloom filter to implement a fast hash table called Direct Collision-free Hash Table (DCHT). DCHT is composed of a front-end on-chip Bloom filter and an underlying off-chip CHT. The Bloom filter is used to construct a compact discriminator table, eliminating most of unnecessary off-chip memory accesses to the underlying CHT. Hence, this scheme can achieve faster collision-free hash table lookup.

Like previous solutions (Song et al., 2005; Kirsch and Mitzenmacher, 2008a,b; Kumar and Crowley, 2005; Kumar et al., 2008), this scheme uses an on-chip Bloom filter as a summary, where all elements stored in an off-chip CHT are inserted. For an irrelevant element that is not in CHT, the Bloom filter may drop its lookup, significantly reducing off-chip memory accesses to CHT. But for a true- or false-positive element through the Bloom filter, DCHT still requires 2^c off-chip memory accesses to check for the element, due to the factor that the Bloom filter cannot identify a unique discriminator value for the element.

Fig. 2 illustrates an example of DCHT, where a single Bloom filter is stored in on-chip memory, and a CHT is stored in off-chip memory. When an element x is queried, we perform parallel membership checks on the on-chip Bloom filter by computing $h_1(x)$, $h_2(x)$, and $h_3(x)$ to see whether x is in the filter. If x is in the filter, we use two possible discriminator values plus x to compute $g(x, 0) = 0$ and $g(x, 1) = 4$, and then search buckets 0 and 4 of CHT to find the exact match. If x is not in the filter, the search terminates, which can avoid unnecessary off-chip memory accesses. Due to the false positives of a Bloom filter, DCHT requires many additional off-chip memory accesses to validate the match, limiting the hash table lookup throughput.

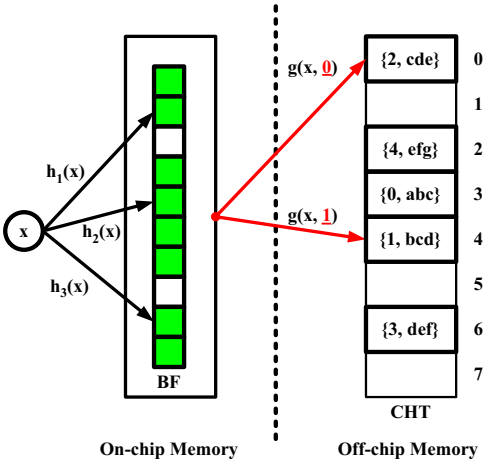

Fig. 2. An example of DCHT.

Alternative solution is to employ the recently proposed Bloomier Filter (Chazelle et al., 2004), in which encodes the discriminator values associated to elements. This solution can provide a possible discriminator value for a queried element, resulting in one off-chip memory access per lookup. However, this solution has the issues of large memory requirements and dynamically changed elements. First, each bucket in the Bloomier Filter needs at least c bits to store a c-bit discriminator instead of a bit of the standard Bloom filter, which incurs prohibitively larger on-chip memory space. Second, the Bloomier Filter can only support a static set of elements, not dynamically changed elements. To address these limitations, we propose DBFs to achieve a single memory access per lookup while retaining memory efficiency in the next subsection.

4.2. Fast apporach using DBF

To reduce the number of off-chip memory accesses, we propose a fast collision-free hashing scheme. This scheme uses a DBF and a CHT to implement a fast and deterministic hash table called Fast Collision-free Hash Table (FCHT). FCHT is composed of an on-chip DBF and an off-chip CHT. DBF is used as a summary to construct an efficient discriminator table, which can not only eliminate most of unnecessary off-chip memory accesses, but also identify a possible discriminator value for a queried element. To validate the match, we hash the discriminator value plus the element by calculating a single hash function to a possible bucket of the off-chip CHT for the check.

DBF comprises an array of parallel Bloom filters organized by the discriminator instead of a single Bloom filter. According to the discriminator values, we partition all elements into an array of 2^c groups, each with the same discriminator value, where c is the bit size of the discriminator. Each group of elements is inserted into a standard Bloom filter, so there are totally at most 2^c Bloom filters running in parallel. In fact, we construct one Bloom filter for each discriminator value, which contains all the elements that have the same discriminator value to select the appropriate buckets to place them. For example, when $c = 2$, all elements stored in CHT are partitioned into four groups, and DBF contains at most four parallel Bloom filters in on-chip memory.

It is easy to check whether an element belongs to the hash table using DBF. We first perform parallel membership checks on 2^c Bloom filters $BF_0, \cdots, BF_{2^c-1}$ of a DBF. Given a query of an element x, we calculate k hash functions $h_1, \cdots, h_k$, and check k hashed bits in each Bloom filter BF_i for $0 \le i \le 2^c-1$. If all k bits are 1, x is in the hash table; otherwise, it is not in the hash table. For x, we need to check which BF_i contains x, and produce a discriminator value i. Then, we use i plus x as input to a hash function $g(x,i)$, and map to a possible bucket of CHT, checking to see whether x exactly matches a pair of {*Node ID*, *Node Key*} in the bucket. Hence FCHT requires one memory access per lookup for most of elements stored in CHT, and filters out irrelevant elements. Algorithm 1 shows the pseudo-code of query operation in FCHT.

Algorithm 1. Query Operation in FCHT

1: **Query in FCHT** (Element x)

2: FCHT is composed of 2^c Bloom filters BF_0, …, BF_{2^c-1}, and a underlying *CHT* with a hash function g.

3: Each BF_i has the same k hash functions h_1, …, h_k, and a vector of m_i bits.

4: **for** (i=0; i <=2^c-1; i++) **do**

```
5:          result=true;
6:          for (j=1; j<=k; j++) do
7:              if (BF_i[h_j(x) % m_i]==0) do
8:                  result=false;
9:                  break;
10:             end do
11:         end do
12:         if (result==true) do
13:             if (CMP(x, CHT[g(x, i)])==true) do
14:                 return true;
15:             else
16:                 return false;
17:             end do
18:         else
19:             return false;
20:         end do
21: end do
```

Fig. 3 illustrates an example of FCHT. The on-chip DBF comprises two parallel Bloom filters, and the off-chip CHT contains five pairs of {*Node ID*, *Node Key*}. In the discriminator table, each entry contains a discriminator value and a subset of pairs of {*Node ID*, *Node Key*}. Each subset of pairs with the same discriminator value i is stored in Bloom filter BF_i of DBF. As shown in Fig. 3, pairs{0, *abc*}, {1, *bcd*}, and {2, *cde*} with discriminator value 0 are stored in Bloom filter BF_0, while pairs {3, *def*} and {4, *efg*} with discriminator value 1 are stored in Bloom filter BF_1. For a query of element x, we perform parallel membership checks on BF_0 and BF_1. If only BF_0 matches x, we produce discriminator value 0 for x. Then, we hash x plus discriminator value 0 by calculating $g(x, 0)$ to a bucket 0 of the off-chip CHT. Finally, we check the bucket to see whether x truly matches element 2 with key *cde*.

We observe that there are different numbers of elements associated with each discriminator value in the off-chip CHT.

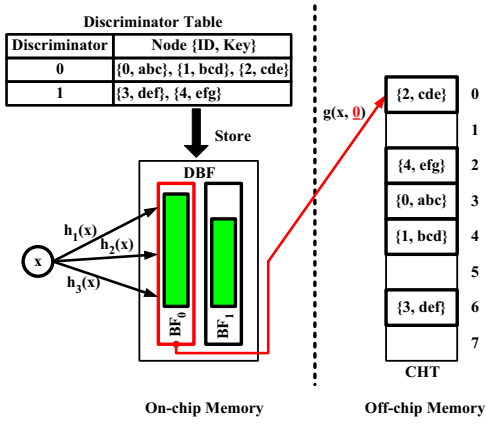

Fig. 3. An example of FCHT.

Hence, we use different size for each Bloom filter according to the number of elements stored in it, in order to minimize the overall false positive probability with a fixed size of on-chip memory. As all the elements are distributed among all the Bloom filters, the on-chip DBF has the same memory usage with a single Bloom filter of DCHT. Due to the false positives, DBF may produce multiple possible discriminator values for a queried element with acceptable probability, leading to a few additional off-chip memory accesses for validating the match.

Moreover, we must bind the lookup time of FCHT, which consists of hash computation time and off-chip memory access time. To reduce the hash computation time, we apply the same group of k hash functions to all the Bloom filters of DBF. We show that the off-chip memory access time is primarily determined by the overall false positive probability of all the Bloom filters. Hence, we will analyze the false positive probability of DBF in the next subsection.

4.3. Handling incremental updates

When elements are changed dynamically, FCHT must support fast incremental updates. However, with an array of standard Bloom filters, we cannot delete elements from the DBFs. A Counting Bloom Filter (CBF) is an extension of the standard Bloom filter that allows adding and deleting elements. CBF stores a counter rather than a bit in each location of the array. To add an element to CBF, we increment the counters at the positions calculated by the hash functions; to delete an element, we decrement the counters.

To handle incremental updates of FCHT, we use an array of parallel CBFs other than standard Bloom filters to compose an on-chip DBF. But, the use of CBF requires larger memory space. There have been several techniques (Bonomi et al., 2006a, 2006b; Hua et al., 2008; Ficara et al., 2008) proposed for reducing the space required, generally at the cost of additional computation and shuffling of memory, while still keeping constant worst-case time bounds on various primitive operations. Such efforts (Hua et al., 2008; Ficara et al., 2008) have exploited the idea of hierarchical structure to compress a great deal of wasted space corresponding to zero counters. To retain speeds and simplicity, we can apply this idea to the on-chip DBF.

Additionally, the underlying CHT must also allow for inserting and deleting elements. When an element is deleted, we just remove the related pair from a hashed bucket, and update the corresponding CBF. But the element insertion is a bit more complicated. To attain one-to-one perfect hashing, we employ the Cuckoo hashing scheme (Pagh and Rodler, 2004; Friez et al., 2009) that uses the same random-walk insertion approach as the CHT construction. This approach places at most one element at each location of the hash table by allowing elements to be moved after their initial placement. Therefore, we obtain a sequence of at most$O(\log n)$ moved elements for updating the corresponding CBFs of DBF.

Fig. 4 illustrates incremental updates of FCHT. When element 5 with the key fgh is inserted in FCHT, we first hash the element's key plus each discriminator value by calculating $h(fgh, 0)$ and $h(fgh, 1)$, and map to buckets 6 and 4 for placing the element. To insert element 5 into the off-chip CHT, we exploit the random-walk insertion approach to allow elements 1 and 4 to be evicted along the movement path $5 \rightarrow 1 \rightarrow 4$. Hence, we move elements 5, 1, and 4 to t buckets 4, 2, and 7 in CHT, and set their discriminator values to 1, 1, and 0. After that, we update the corresponding CBFs of DBF. Due to the changed discriminator values, we have to move element 1 from CBF_0 to CBF_1, and 4 from CBF_1 to CBF_0. As it has discriminator value 1, element 5 is inserted into CBF_1.

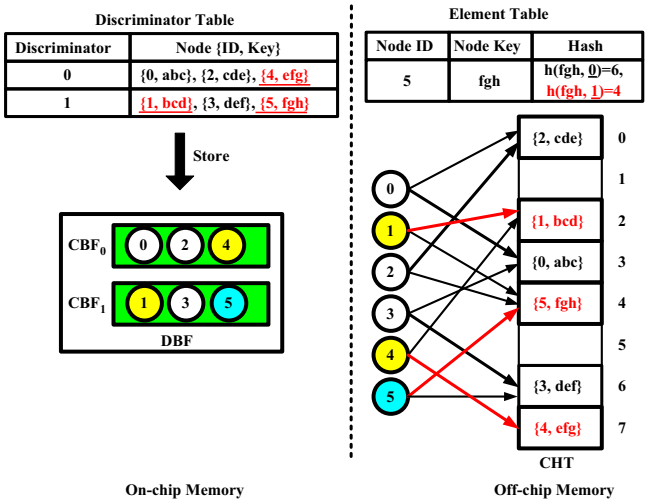

Fig. 4. Incremental updates of FCHT.

4.4. False positive probability analysis

One key problem of DBF is the false positive, where an element is claimed as a member in the set even though it is not in the set. When a queried element experiences false positives, DBF may produce multiple possible discriminator values for the element. Then, FCHT needs multiple additional memory accesses to the underlying CHT for finding the exact match. Therefore we analyze the false positives effect of DBF on the lookup performance of FCHT, and explore the key parameters to optimize the lookup performance.

We use the expected number of off-chip memory accesses to measure the hash table lookup performance. In FCHT, the lookup performance is dominated by the number of matching Bloom filters of an on-chip DBF. It is easy to find that the worst-case lookup performance is at most 2^c off-chip memory accesses, where c is the bit size of a discriminator. To improve the FCHT lookup performance, we must minimize the overall false positive probability of an on-chip DBF. Assume that each Bloom filter BF_i of DBF has the same k hash functions and the constant ratio m_i/n_i, where m_i is the number of bits in BF_i, and n_i is the number of elements in BF_i. Hence all Bloom filters of DBF have the same false positive probability f.

For a query of an element, we can derive the occurrence probability of the number of Bloom filters matching the element. Let F be a random variable for the number of matching Bloom filter, and j be constant in the range $\{0,\cdots,2^c\}$. When $F=j$, the probability that exactly j Bloom filters match the element is calculated as follows:

$$P(F=j)=\binom{2^c}{j}f^j(1-f)^{2^c-j} \tag{2}$$

For each value j, the expected number E of off-chip memory accesses of FCHT is calculated as follows:

$$E=\sum_{j=0}^{2^c} jP(F=j)=\sum_{j=0}^{2^c} j\binom{2^c}{j}f^j(1-f)^{2^c-j} \tag{3}$$

As the probability $P(F=j)$ follows the binomial distribution, the expected number E is calculated as follows:

$$E=2^cf \tag{4}$$

Therefore the worst-case number E_{worst} of off-chip memory accesses is calculated as follows:

$$E_{worst}=2^c \tag{5}$$

Eqs. (4) and (5) show that both the average-case and worst-case lookup performance primarily depends on the number 2^c of parallel Bloom filters of DBF. Hence, reducing bucket choices c is important to improve the worst-case lookup performance of FCHT.

We explore how the value ranges of c impact on the lookup performance of FCHT. Assume that there are n distinct elements and M buckets in the hash table. The analysis of the Cuckoo hashing scheme (Pagh and Rodler, 2004) has shown that we can have a constant small value of c if M is slightly greater than n. For example, if $M=1.1n$, then $c=2$ can ensure a perfect matching with high probability. Recent work (Kumar et al., 2007; Ficara et al., 2009) has also shown that when $M=n$, using $c=O(\log\log n)$ bits of a discriminator can guarantee that a perfect hash table exists and it can support fast updates. As there are at most $O(\log n)$ bucket choices for each element, FCHT using DBF can improve both the average-case and worst-case performance, achieving fast and deterministic hash table lookup.

4.5. Network applications of DBFs

We explore two network functions using DBF in high-speed routers, including IP route lookup and deep packet inspection (DPI). IP route lookup performs a longest prefix matching, where the destination IP of a packet is matched against a set of IP prefixes. DPI performs an exact pattern matching, where both packet headers and payloads are inspected against a set of known signatures. Besides in routers, these algorithms have been widely used in intrusion detection and prevention, network monitoring, and traffic accounting.

To achieve high speeds, the literatures (Dharmapurikar et al., 2003, 2004) have recently proposed Parallel Bloom Filters (PBFs) for IP route lookup and DPI. This solution consists of an on-chip PBF and an array of off-chip hash tables. PBF is composed of an array of standard Bloom filters organized by the rule length, e.g. prefix length for IP route lookup, and signature string length for DPI. According to the rule length, all rules in a database are partitioned into an array of subset, and each subset of rules with the same length is inserted into both a corresponding Bloom filter of PBF and an off-chip hash table. One hash table with a single hash function corresponds to one on-chip Bloom filter of PBF for validating the match. For a query, all Bloom filters of PBF operate on the corresponding fields (e.g. destination IP address or payload) of various lengths from an incoming packet. Each Bloom filter tests the membership, and simply indicates match or no match. Due to the false positives of Bloom filters, this PBF-based solution needs to check a corresponding off-chip hash table for validating each possible match.

To reduce off-chip memory accesses of the solution above, we propose a novel DBF-based architecture for high-speed IP route lookup and DPI. In this architecture, we use a DBF to replace each standard Bloom filter of a PBF, so that there is an array of parallel DBFs in on-chip memory. Accordingly, we use a CHT to replace each general hash table corresponding to an on-chip Bloom filter, so that there is an array of off-chip CHTs. Fig. 5 illustrates a packet processing architecture using DBFs. This architecture is composed of an array of on-chip parallel DBFs and an array of off-chip CHTs. Note that each DBF is composed of an array of parallel standard Bloom filters, each containing a group of rules with the same discriminator value. Hence, all the DBFs have the same on-chip memory space as PBF.

This DBF-based solution works as follows. When handling an incoming packet, we extract the fields of different lengths in the range $\{1,\cdots,W\}$ (see Fig. 5) from the packet, and then feed them to each corresponding DBF in on-chip memory. Each DBF checks the membership in parallel, and then identifies a possible

discriminator value for each field. Like FCHT, this solution hashes each field plus its discriminator value to a corresponding off-chip CHT, and then searches the exact match to output a matching rule for packet processing. In essence, our architecture is composed of an array of parallel FCHTs, each containing a subset of rules with the same length. We show that this solution has the same on-chip memory usage as the PBF-based solution above, while requiring less off-chip memory accesses as well as smaller off-chip memory usage.

5. Experimental results

We have conducted simulation experiments to evaluate the performance of DBF. In the experiments we mainly compare FCHT with CHT and DCHT in terms of the number of off-chip memory accesses per lookup and update overhead. There are two categories of experiments for performance evaluation.

In the first experiments, we synthesize a storage set that is inserted in a hash table, and a testing set for query on the hash table. The testing set contains 10-fold elements of a storage set. Each element is a 4-byte string that is randomly generated from a given alphabet {'*a*'-'*z*','*A*'-'*Z*'}. The testing set contains true elements of 20% to 80% that are stored in the storage set.

In the second experiments, we obtain a storage set of equal-sized IP prefixes and Snort signatures from real-world networks. We synthesize a testing set of IP addresses and payload strings for query, which contains true elements of 40% and 80% that are stored in the storage set.

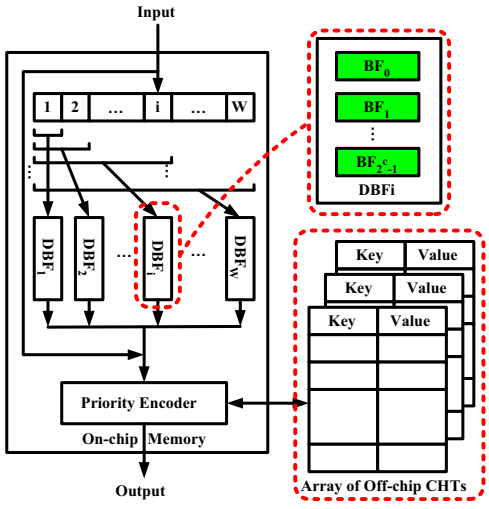

Fig. 5. Packet processing architecture using DBFs.

5.1. Experiments on synthetic sets

We conduct the experiments on synthetic sets to examine the performance of FCHT, CHT, and DCHT in different settings. In the experiments, there are $n=10K$ and $n=100K$ unique elements and $M=n$ buckets in an off-chip hash table; each element has a discriminator of $c=3$ or $c=4$ bits, indicating that there are 8 or 16 bucket choices for an element. For fair evaluation, we must choose the appropriate value of the parameters (e.g., m, n, an$k$) of Bloom filters to optimize the performance of FCHT, CHT, and DCHT. Given that the ratio of m/n is constant in an on-chip Bloom filter, we select $k=(m/n)\ln 2$ hash functions to minimize the false positive probability, where n is the number of elements, and m is the number of buckets in the Bloom filter. Each Bloom filter of DBF has the same parameters of k and m/n. As m/n is the same, both DCHT and FCHT have the same on-chip memory usage. When $m/n=10$, we can easily calculate the on-chip memory size of $m=100K$ bits and $m=1M$ bits in case of $n=10K$ and $n=100K$.

Table 1 shows the comparisons of false positive probability between DCHT and FCHT. Due to the aggregate false positives from all Bloom filters of DBF, FCHT has larger false positive probability than DCHT. For instance, in case of $m/n=16$ and $c=4$, when the ratio of true elements to total queried elements increases from 20% to 80%, the false positive probability of DCHT and FCHT separately decreases from 1.59% down to 0.10% and from 3.00% down to 0.19%. Nevertheless, our later experiments show that FCHT requires less off-chip memory accesses per lookup than DCHT since DBF can identify a unique discriminator for a queried element.

Table 2 shows the number of off-chip memory accesses per lookup in case of $c=3$. DCHT and FCHT have constant number of off-chip memory accesses per lookup, while CHT has variable number of off-chip memory accesses. For instance, when $m/n=16$ and $k=11$, CHT has 6.8–40.5 off-chip memory access per lookup, and DCHT has 4.5 off-chip memory accesses while FCHT has only 1.0 off-chip memory accesses. Table 2 shows that FCHT requires about one off-chip memory access per lookup, less than both CHT and DCHT.

Fig. 6 depicts average off-chip memory accesses per lookup in case of $c=3$. We observe that FCHT requires much less average off-chip memory accesses per lookup than both CHT and DCHT. When $m/n=10$, FCHT reduces average off-chip memory accesses per lookup by 6.5–30.9 times and by 4.1 times compared to CHT and DCHT; when $m/n=16$, FCHT reduces average off-chip memory accesses by 6.7–39.8 times and by 4.5 times. Fig. 6 demonstrates that FCHT using DBF outperforms previous schemes in terms of the number of off-chip memory accesses per lookup.

Table 3 shows the number of off-chip memory accesses per lookup in case of $c=4$. We also observe that DCHT and FCHT have constant number of off-chip memory accesses per lookup, while CHT has variable number of off-chip memory accesses.

Table 1
False positive probability.

Ratio of True Elements (%)	**False Positive Probability** (%)			
	***m/n*=10, *k*=6 *c*=3**		***m/n*=16, *k*=11 *c*=4**	
	DCHT (%)	**FCHT** (%)	**DCHT** (%)	**FCHT** (%)
20	21.28	33.77	1.59	3.00
30	13.65	22.95	1.01	1.77
40	9.24	16.03	0.65	1.14
50	6.35	11.29	0.42	0.77
60	4.31	7.83	0.27	0.51
70	2.82	5.17	0.17	0.33
80	1.67	3.08	0.10	0.19

Table 2
Number of off-chip memory accesses per lookup in case of $c=3$.

$c=3$	$M=n=10$ K					$M=n=100$ K				
Ratio of True Elements (%)		$m/n=10$, $k=6$		$m/n=16$, $k=11$			$m/n=10$, $k=6$		$m/n=16$, $k=11$	
	CHT	**DCHT**	**FCHT**	**DCHT**	**FCHT**	**CHT**	**DCHT**	**FCHT**	**DCHT**	**FCHT**
20	40.5	4.8	1.3	4.5	1.0	40.5	4.8	1.3	4.5	1.0
30	25.5	4.7	1.2	4.5	1.0	25.5	4.7	1.2	4.5	1.0
40	18.0	4.6	1.1	4.5	1.0	18.0	4.6	1.1	4.5	1.0
50	13.5	4.6	1.1	4.5	1.0	13.5	4.6	1.1	4.5	1.0
60	10.5	4.6	1.1	4.5	1.0	10.5	4.6	1.1	4.5	1.0
70	8.4	4.6	1.1	4.5	1.0	8.4	4.6	1.1	4.5	1.0
80	6.8	4.5	1.1	4.5	1.0	6.8	4.5	1.1	4.5	1.0

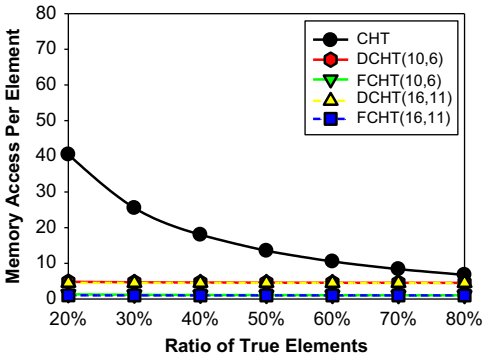

Fig. 6. Average off-chip memory accesses per lookup in case of $c=3$.

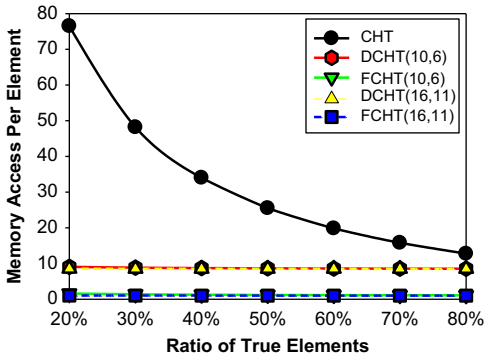

Fig. 7. Average off-chip memory accesses per lookup in case of $c=4$.

For instance, when $n=10K$ and $m/n=16$, CHT has 12.8–76.5 off-chip memory accesses per lookup, and DCHT has 8.5 off-chip memory accesses, while FCHT has 1.0 off-chip memory accesses. Table 3 also demonstrates that FCHT requires much less off-chip memory accesses per lookup than both CHT and DCHT.

Fig. 7 depicts average off-chip memory accesses per lookup in case of $c=4$. Fig. 7 also demonstrates that FCHT requires less average memory accesses per lookup than both CHT and DCHT. When $m/n=10$, FCHT reduces average off-chip memory accesses per lookup by 11.6–47.4 times and by 7 times compared to CHT and DCHT; when $m/n=16$, FCHT reduces average off-chip memory accesses by 12.7–73.9 times and by 8.5 times.

Fig. 8 shows the update overhead. Fig. 8(a) shows that CHT, DCHT, and FCHT have constant off-chip memory accesses per deletion. As a single on-chip Bloom filter cannot identify a discriminator value for a queried element, CHT and DCHT has the same deletion overhead. We also observe that CHT and DCHT also have the same insertion overhead. Because DBF can identify a unique discriminator value for an element, FCHT has one off-chip memory access per deletion, less than both CHT and DCHT. Fig. 8(b) shows the insertion overhead of FCHT, where newly inserted elements account for 0.1%, 1%, and 10% of elements in the hash table. When the occupancy increases from 0.1% to 10%, FCHT requires less off-chip memory accesses per insertion. The reason is that more inserted elements have more empty buckets, which offers more opportunities to find an argument path for the insertion (Pagh and Rodler, 2004; Friez et al., 2009).

5.2. Experiments on realistic sets

We conduct the experiments on realistic sets to evaluate two packet processing applications of DBF, including IP route lookup and DPI. To evaluate IP route lookup, we obtain four representative real-world IPv4 prefix tables of core routers. AS6447 and AS65000 are collected from (⟨http://bgp.potaroo.net⟩), while OIX and V3 are collected from (⟨http://www.routerreview.org⟩). As shown In Table 4, AS6447, AS65000, OIX, and V3 are large-scale IPv4 BGP tables that contain about 310 K, 217 K, 347 K, and 250 K prefixes, respectively. As the prefixes of length 24 primarily account for about 30–50%, we extract these prefixes from four prefix tables to examine the performance of IP route lookup.

To evaluate DPI, we also obtain a set of Snort attack signatures. Snort 2.7 has a set of total 7840 signatures (⟨http://www.snort.org⟩). For evaluation purposes, we partition the Snort set into a group of subsets according to the signature length, and sort the group in descending order. Similarly, we extract three first subsets of Snort signatures to examine the performance of DPI. As shown in Table 5, Snort379, Snort348, and Snort247 contain 379, 348, and 247 signatures of equal length, respectively.

We mainly compare PBF-based and DBF-based approaches to IP route lookup and DPI. For fair evaluations, we set the optimal parameters of on-chip Bloom filters and off-chip hash tables in the experiments. Given a constant ratio of $m/n=16$, $k=11$ is chosen to minimize the false positive probability of on-chip Bloom filters. Furthermore, we set the parameters of $M/n=1.1$ and $c=4$ for each off-chip CHT in the DBF-based approach to optimize the hash table lookup performance. In the experiments a testing set contains true elements of 40% and 80% stored in a storage set of IP prefixes and Snort signatures.

Fig. 9 depicts the number of off-chip memory accesses per lookup in case of IP route lookup. In this experiment we synthesize a set of testing prefixes each with the length 24. Fig. 9 shows that the PBF-based approach has 8.5 average off-chip memory accesses per lookup, while the DBF-based approach has 1.0 average off-chip memory accesses. Therefore for IP route lookup, the DBF-based approach reduces the number of off-chip memory accesses per lookup by 8.5 times compared to the PBF-based approach.

Fig. 10 depicts the number of off-chip memory accesses per lookup in case of DPI. In this experiment we synthesize a set of

Table 3
Number of off-chip memory accesses per lookup in case of $c=4$.

$c=4$	$M=n=10$ K					$M=n=100$ K				
Ratio of True Elements (%)		$m/n=10$, $k=6$		$m/n=16$, $k=11$			$m/n=10$, $k=6$		$m/n=16$, $k=11$	
	CHT	**DCHT**	**FCHT**	**DCHT**	**FCHT**	**CHT**	**DCHT**	**FCHT**	**DCHT**	**FCHT**
20	76.5	9.1	1.6	8.5	1.0	76.50	9.08	1.6	8.6	1.0
30	48.2	8.8	1.4	8.5	1.0	48.17	8.84	1.3	8.5	1.0
40	34.0	8.7	1.3	8.5	1.0	34.00	8.72	1.2	8.5	1.0
50	25.5	8.7	1.2	8.5	1.0	25.50	8.65	1.2	8.5	1.0
60	19.8	8.6	1.3	8.5	1.0	19.84	8.60	1.1	8.5	1.0
70	15.8	8.6	1.1	8.5	1.0	15.79	8.56	1.1	8.5	1.0
80	12.8	8.6	1.1	8.5	1.0	12.75	8.54	1.1	8.5	1.0

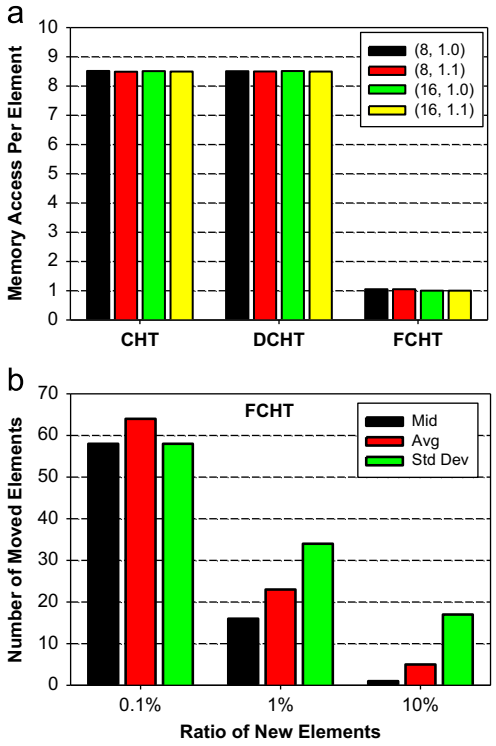

Fig. 8. Update overhead: (a) deletion and (b) insertion.

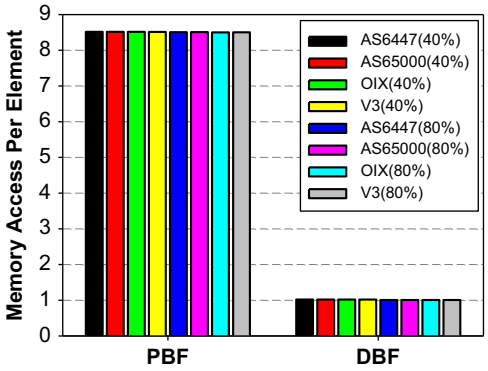

Fig. 9. Off-chip memory accesses per lookup in case of IP route lookup.

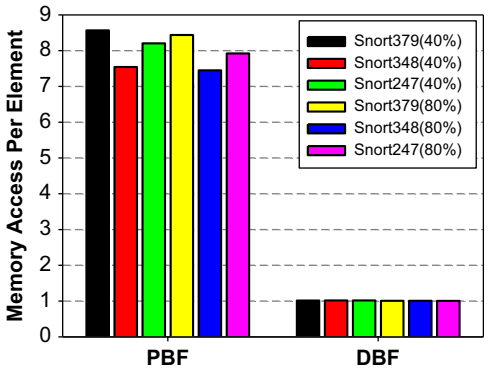

Fig. 10. Off-chip memory accesses per lookup in case of DPI.

Table 4
IP Prefixes.

Type	**Prefix Table**	**#Prefixes**
IPv4	AS6447	310,344
	AS65000	217,952
	OIX	347,408
	V3	250,376

Table 5
Snort Signatures.

Type	**Signature Set**	**#Signatures**
Snort	Snort379	379
	Snort348	348
	Snort247	247

testing strings against a storage set of Snort signatures. We observe that the PBF-based approach has 8.0 average off-chip memory accesses per lookup, while the DBF-based approach has 1.0 average off-chip memory accesses. Fig. 10 shows that for DPI, the DBF-based approach reduces the number of off-chip memory accesses per lookup by 8.0 times compared to the PBF-based approach.

6. Conclusions

We propose a fast collision-free hashing scheme called FCHT using DBF to achieve fast and deterministic hash table lookup. FCHT is composed of an on-chip DBF and an off-chip CHT. DBF is composed of an array of parallel Bloom filters organized by the discriminator. Each element in a CHT has a discriminator, indicating that it has multiple bucket choices to place the element. For a query of an element, all Bloom filters of an on-chip DBF are checked in parallel for the membership to produce a possible discriminator value for the element. Then the discriminator value

plus the element is hashed to a possible bucket in an off-chip CHT for validating the match. We also explore two network applications of DBF, including IP route lookup and DPI.

We show that FCHT achieves one off-chip memory access per lookup by using DBF. Experiments on synthetic sets show that compared to CHT and DCHT, FCHT reduces the number of off-chip memory accesses per lookup by up to 73.9 times and by up to 8.5 times. Experiments on realistic sets show that for packet processing the DBF-based approach requires less significant off-chip memory accesses per lookup than the PBF-based approach.

Acknowledgment

This work is supported in part by the National Basic Research Program of China (No. 2012CB315801), the National Science Foundation of China (No. 61100171, No. 61133015 and No. 61061130562), the China Postdoctoral Science Foundation (No. 201104155), and the National Science and Technology Major Project of China (No. 2012ZX03002016).

References

Azar Y, Broder A, Karlin A, Upfai E. Balanced allocations. In: ACM STOC; 1994. p. 593–602.

Border A, Mitzenmacher M. Using multiple hash functions to improve IP lookups. In: IEEE INFOCOM; 2001. p. 1454–63.

Baboescu F, Varghese G. Scalable packet classification. In: ACM SIGCOMM; 2001. p. 199–210.

Bloom B. Space/time tradeoffs in hash coding with allowable errors. Communications of the ACM 1970;13(7):422–6.

Broder A, Mitzenmacher M. Network applications of Bloom filters: a survey. Internet Mathematics 2004;1(4):485–509.

Bonomi F, Mitzenmacher M, Panigrapy R, Singh S, Varghese G. Beyond bloom filters: from approximate membership checks to approximate state machines. In: ACM SIGCOMM, 2006a. p. 315–26.

Bonomi F, Mitzenmacher M, Panigrapy R, Singh S, Varghese G. An improved construction for counting Bloom filters. In: ESA; 2006b. p. 684–95.

Chazelle B, Kilian J, Rubinfeld R, Tal A. The Bloomier filter: an efficient data structure for static support lookup tables. In: SODA; 2004. p. 30–9.

Dharmapurikar S, Krishnamurthy P, Taylor D. Longest prefix matching using bloom filters. In: ACM SIGCOMM; 2003. p. 201–12.

Dharmapurikar S, Krishnamurthy P, Sproull T, Lockwood J. Deep packet inspection using parallel bloom filter. IEEE Micro 2004;24(1):52–61.

Estan C, Varghese G, New directions in traffic measurement and accounting. In: ACM SIGCOMM; 2002. p. 323–36.

Estan C, Keys K, Moore D., Varghese G. Building a better netflow. In: ACM SIGCOMM; 2004. p. 245–56.

Ficara D, Giordano S, Kumar S, Lynch B. Divide and discriminate: algorithm for determistic and fast hash lookups. In: ANCS; 2009. p. 133–42.

Fan L, Cao P, Almeida J, Broder A. Summary cache: a scalable wide-area web cache sharing protocol. IEEE/ACM Transactions on Networking 2000;8(3):281–93.

Friez A, Melsted P, Mitzenmacher M. An analysis of random-walk cuckoo hashing. In: APPROX and RANDOM; 2009. p. 490–503.

Ficara D, Giordano S, Procissi G, Vitucci F. Multilayer compresses counting Bloom filters. In: IEEE INFOCOM; 2008. p. 311–5.

Hua N, Zhao H, Lin B, Xu J. Rank-indexed hashing: a compact construction of Bloom filters and variants. In: IEEE ICNP; 2008. p. 73–82.

Kirsch A, Mitzenmacher M. Simple summaries for hashing with choices. IEEE/ACM Transactions on Networking 2008a;16(1):218–31.

Kumar S, Crowley P. Segmented hash: an efficient hash table implementation for high performance networking subsystems. In: IEEE/ACM ANCS; 2005. p. 91–103.

Kumar S, Turner J, Crowley P. Peacock hashing: deterministic and updatable hashing for high performance networking. In: IEEE INFOCOM; 2008. p. 101–5.

Kirsch A, Mitzenmacher M. The power of one move: hashing schemes for hardware. In: IEEE INFOCOM; 2008b. p. 106–10.

Kumar S, Turner J, Crowley P, Mitzenmacher M. HEXA: compact data structures for faster packet processing. In: IEEE ICNP; 2007. p. 246–55.

Pagh R, Rodler F. Cuckoo hashing. Journal of Algorithms 2004;51(2):122–44.

Srinivasan V, Suri S, Varghese G. Packet classification using tuple space search. In: ACM SIGCOMM; 1999. p. 135–46.

Song H, Dharmapurikar S, Turner J, Lockwood J. Fast hash table lookup using extended Bloom filter: an aid to network processing. In: ACM SIGCOMM; 2005. p. 181–92.

Vocking B. How asymmetry helps load balancing? In: IEEE FOCS; 1999. p. 131–41.

Yu H, Mahapatra R. A memory-efficient hashing by multi-predicate Bloom filters for packet classification. In: IEEE INFOCOM; 2008. p. 1795–803.

Bgp table, 〈http://bgp.potaroo.net〉.

Route reviews project, 〈http://www.routerreview.org〉.

Snort, 〈http://www.snort.org〉.