A Memory-Efficient Hashing by Multi-Predicate Bloom Filters for Packet Classification

Heeyeol Yu and Rabi Mahapatra
Computer Science Department
Texas A&M University
College Station, TX 77843
Email: {hyyu,rabi}@cs.tamu.edu

Abstract*— Hash tables (HTs) are poorly designed for multiple off-chip memory accesses during packet classification and critically affect throughput in high-speed routers. Therefore, an HT with fast on-chip memory and high-capacity off-chip memory for predictable lookup-throughput is desirable. Both a legacy HT (LHT) and a recently proposed fast HT (FHT) [1] have the disadvantage of memory overhead due to pointers and duplicate items in linked lists. Also, memory usage for an FHT did not consider the bits in counters for fair comparision with an LHT. In this paper, we propose a novel hash architecture called a *Multi-predicate Bloom-filtered HT* (MBHT) using *parallel* Bloom filters and generating off-chip memory addresses in the base-2^x number system, $x \in \{1, 2, \cdots\}$, which removes the overhead of pointers. Using a larger base of number system, an MBHT reduces on-chip memory size by a factor of $\log_2 b_2 / \log_2 b_1$ where b_1 and b_2 are bases of number system ($b_2 > b_1$). Compared to an FHT, the MBHT is approximately $x(\log_2 n + 4)/(2\log_2 n)$ times more efficient for on-chip memory, where n is the number of keys. This results in a significant reduction in the number of off-chip memory accesses. A simulation with a dataset of packets from NLANR [2] shows the on-chip memory reductions by 1.7 and 2 times over an LHT and an FHT are made. Besides, an MBHT of base-16 needs less off-chip memory accesses by 2117 in total URL queries of NLANR, compared to an FHT.

I. INTRODUCTION

The demand for high-speed and large-scale routers continues to increase, especially in legacy networking. As a result, fast packet classifications have become critical data path functions for many emerging networking applications [3, 4]. Networking devices to support firewall, access control list, and quality of service in several network domains use these functions. For instance, in layer 4 using TCP or UDP port [5, 6], provision of efficient URL switching is important in large scale content distribution networks due to the following reasons: URLs are much longer than IP addresses, URLs are not stable, and new URLs are continuously being created [7–9]. To meet seamless packet classification in wire-speed for high-speed routers, it is desirable to use high-bandwidth and small on-chip memory while the database of rules for packet classification resides in the slower and higher capacity off-chip memory [1, 10]. For predictable classification-throughput in the worst case, the speed of a packet classification algorithm is generally measured by the number of off-chip memory accesses. An interesting approach is the use of Ternary Content Addressable Memory (TCAM) to achieve deterministic, high-speed LPM for packet classification and IP lookup [11–13].

TCAM use, however, incurs high cost and power consumption. Approaches using a Bloom filter (BF), which address the cost and power issues, have been widely documented in the networking literature [1, 10, 14, 15], especially for packet classification. Dharmapurikar *et al.* [10] introduced the first algorithm to employ BFs working *in parallel* for LPM essential to IP routing lookup. A BF is a generalized form of a hash table (HT) using numerous hash functions and is essentially a fast binary predicate for membership testing on a set with efficient memory by returning *'Yes'* if a queried item exists in the set or *'No'* otherwise.

Traditionally for a fast search, an HT is used for performing fast associative lookups, which requires $O(1)$ average memory access per lookup under reasonable assumptions. The major concern in hashing is to reduce collision among keys, because even if the hash function is well designed, it is impossible to hash universal elements without any collisions. For example, the famous "birthday paradox" asserts that if 23 or more people are present in a room, chances are good that two of them will have the same month and day of birth. In other words, if we select a random function, which maps 23 keys into a table of 365 buckets, the probability that no two keys map into the same location is only 0.4927 [16]. Literature has proposed various collision resolutions, such as open addressing and chaining methods. Chaining method, especially like those described by [1], needs pointers to resolve hash collisions among items.

A typical application of an HT is packet processing in a network [1, 8]. Song *et al.* [1] proposed a fast HT (FHT) with the help of a BF to reduce the number of off-chip memory accesses compared to a legacy hash table (LHT) This benefit comes from sharing k linked lists, each indexed by one of the k hash functions so that only the shortest linked list is used in the search, i.e. *qeury* operation. Sharing k linked lists, however, not only has the generic memory overhead of a pointer in a linked list, it also has the following disadvantages. First, due to the merging of k linked lists, chance that duplicate items are saved in off-chip memory is ample enough that an FHT needs three times more off-

978-1-4244-2026-1/08/$25.00 © 2008 IEEE

chip memory than an LHT [1]. Although searching for an item is expedited by choosing the shortest linked list, the *insert* and *delete* operations take approximately k off-chip memory accesses. These operations are not suitable for a dynamically changing set because any change in the set by one item needs $2k$ times of off-chip memory access. Also, to get better performance over an LHT, an FHT also needs many buckets in on-chip memory to reduce collision, resulting in large number of wasted buckets. The number of buckets, which work as starting pointers in on-chip memory, affects the collision rate as a hash function is based on the number of the total buckets, i.e. the on-chip memory size in bits. The load factor, defined as the number of elements over the number of buckets, is a useful metric to measure the efficiency of memory usage, as well as the collision rate and the length of the linked list. As the load factor is decreased, the rate of collision is reduced. Yet, if the load factor is set very small value for predictable throughput as an FHT sets it 0.07, most of the buckets are not used during three operations.

Beyond previous approaches like TCAM, with its high cost and power consumption, or an FHT, with duplicate copies of items and shared linked lists requiring more memory accesses, we propose a novel hash architecture that uses a set of BFs *in parallel* for packet processing applications. Also, BFs used in our hash mechanism are designed to support a multi-predicate rather than a simple membership tester, i.e. binary-predicate, of a legacy BF. There are two benefits for using a *Multi-predicate Bloom-filtered HT* (MBHT) as regards to on-chip and off-chip memory. In on-chip memory, a multi-predicate BF reduces the memory size in base-2^x number system by x times compared to that of base-2^1 number system with a binary predicate BF, where x is a positive integer 1. The *insert* and *delete* operations are done on each BF in constant time *in parallel* in contrast to an FHT that takes $O(k)$ in an optimal case. For off-chip memory, by abolishing the linked list mechanism used to resolve collision in buckets, our MBHT saves memory by removing pointers in a linked list. Furthermore, an MBHT does not save the duplicated items, resulting in reduced off-chip memory.

This research makes the following contributions:

- For fast search, an MBHT scheme is proposed using a contiguous memory space in off-chip memory without a linked list.
- New algorithms on *insert*, *query*, and *delete* operations are proposed for the MBHT that reduce the on-chip memory.
- It is shown that the MBHT performs $\frac{x(\log_2 n+4)}{2\cdot\log_2 n}$ and $\frac{x}{2}$ times better in terms of space, compared to other contemporary techniques of an LHT and an FHT, respectively, where n is the number of items and x is an postive integer in the base-2^x number system.
- It is shown that the saved on-chip memory through a multi-predicate BF can be utilized to make average access per search to off-chip memory smaller for the *query* operation.

The paper is organized as follows. Sec. II explains the basic concepts, and Sec. III shows the design of an MBHT and the memory efficiency of a multi-predicate BF. Sec. IV presents analyses of memory efficiency and access time of an LHT, an FHT, and an MBHT for comparison. Related works and conclusions are presented in Sec. V and Sec. VI, respectively.

II. Baic Theory of Bloom filter

To understand the fundamental relationship among the number of buckets, m; the number of items, n; and the number of hash functions, k, we present the mathematics about a BF and a false positive, or f-positive. We then introduce the mechanism of `insert`, `query`, and `delete` operations.

A legacy BF for representing set $S=\{e_0, e_1, ..., e_{n\text{-}1}\}$ of n elements is described by an array of m bits with each initially set to 0. A BF uses a set H of k independent hash functions $h_0, h_1, ..., h_{k\text{-}1}$ with range $[0{:}m\text{-}1]$. For mathematical convenience, we make a natural assumption that these hash functions map each item in the universe to a random number uniform over the range. For each element $e_{j'} \in S$, the bits indexed by $h_{k'}(e_{j'})$ are set to 1 for $0 \leq k' \leq k\text{-}1$, $0 \leq j' \leq n\text{-}1$. To verify that item e' is in S, we check whether k bits in a BF indicated by $h_{k'}(e')$ are 1. If not, then clearly e' is not a member of S. Even if chosen bits indexed by $h_{k'}(y)$ have a value 1, there may be a probability called f-positive that item y is falsely believed to belong to set S due to the random gathering of k bits of value 1 set by independent items.

The above probability f of f-positive can be formulated in a straightforward way, given our assumption that hash functions are perfectly random. Among m bits, the chance of a bit being value 0 by one h_k is $1/m$. After all n elements of S are hashed k times into the BF, i.e. totaling $k\cdot n$ times, the probability that a specific bit is still 0 is asymptotically $p = (1\text{-}1/m)^{kn} \approx e^{-kn/m}$. Then, the probability of an f-positive by randomly choosing k bits among m bits is

$$f \geq \left\{1 - \left(1 - \frac{1}{m}\right)^{kn}\right\}^{k} \approx (1-p)^k \geq (1/2)^{m \ln 2/n} \tag{1}$$

because k bits with probability of becoming 0, or p, could independently become more than 0 when a membership test is requested. This probability is bounded and the optimal k, the number of hash functions, that minimizes f is easily found $k = \ln 2(m/n)$ according to the results of Broder and Mitzenmacher [17]. After some algebraic manipulation, Broder and Mitzenmacher [17] claimed that the requirement of $f \leq \epsilon$ suggests

$$m \geq n\frac{\log_2(1/\epsilon)}{\ln 2} \approx 1.44n\log_2(1/\epsilon). \tag{2}$$

Two important lemmas can be derived from Eq. (2), described as follows

Lemma 1 (Linear Property) *Linear property between m and n exists in Eq. (2) because given f requires that variable n is linearly proportionate to variable m. Therefore, if n is* *reduced by half or decreased by constant* α*, the desired* m *for a given* f *is reduced by half or decreased by the constant of* $\alpha\cdot 1.44\log_2(1/\epsilon)$*, respectively.*

LEMMA 2 (REVERSE EXPONENTIAL PROPERTY) *The change of* m *has an exponential effect on* f *for a given* n *from Eq. (2). That is, if* m *is increased by constant* α *or multiplied* x *times,* f *is exponentially divided on base-2 by the power of constant* α/c *or by the power of* xm/c *times where* $x>1$*, constant* $c=1.44n$*.*

These *Linear* and *Reverse Exponential Properties* are used in introducing a y-*BF* so that an MBHT has the benefit of memory saving in on-chip memory by the *Linear Property*, and, thereinafter the saved memory is designed to decrease f by the *Reverse Exponential Property*.

III. MULTI-PREDICATE BLOOM-FILTERED HASH TABLE

Fig. 1 shows the macro view of our architecture with two MBHTs, *l-MBHT* and *r-MBHT*, and a queue of free addresses residing in on-chip memory while there is a rule table of $n=2^2$ entries in off-chip memory. One of MBHTs is involved in `insert` operation depending on l/r-register. This register is to be switched l or r whenever n `inserts` are made on one MBHT so that once a window of the queue is used up the peer MBHT is cleaned up for future `insert`. Through this rotation, without counting BFs of 4-bit counters dual MBHTs can provide seamlessly `delete` operations for incremental updates of rules. In contrast, both of MBHTs are involved in `query` and `delete` operations because it is not known where a wanted item is located.

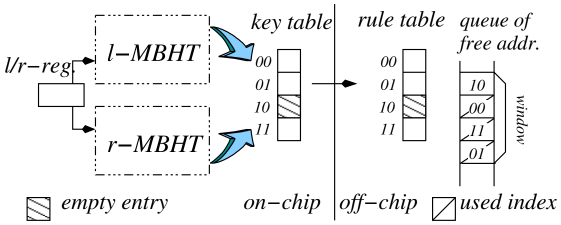


Fig. 1. Macro view of an MBHT in on/off-chip memory of base-2. $n=2^2$. Indexes 10, 11, and 00 in order are used in `insert`.

A. `Insert` operation in an MBHT

Beyond an FHT [1] and a scheme of multiple binary predicates [10], we propose a new hashing architecture capable of indexing off-chip memory space in the base-2^x number system and confirming that the indexed entry of a table in off-chip memory is for a wanted item, that is *exact matching*. Unlike a BF for shared linked lists in an FHT, we design an MBHT with a set of multi-predicate BFs. Although a BF is returning value one, we use an augmented y-*BF* capable of y-predicate membership testing returning value y when a membership test is met.

Denote y-*BF* a y-predicate BF composed of a legacy BF and indicator y in x bits for the base-2^x number system. Furthermore, assume there are n elements to hash in off-chip memory where they are saved in contiguous and flat memory space, unlike a saving scheme of shared linked lists [1]. The n elements are saved in an arbitrary order at address A^b of off-chip memory, where b is the base-b number system in the form of 2^x, $x\in\{1,2,\cdots\}$. Given n and the base-b number system, there are $r=\log_b n$ digits, and address A^b is composed of r digits of x bits, i.e. $A_0A_1\cdots A_{r\text{-}1}$, $A_i\in\{0,\cdots,2^x\text{-}1\}$. Now, given the address space is based on a number system of base-b, we are about to partition the address space with a set of y-*BF*s, $y\in\{0,\cdots,2^x\text{-}1\}$, so that each A_i in base-2^x is to be covered by A_i-BF^i, $0\leq i\leq r\text{-}1$. After the digits are partitioned column-wise by a set of y-*BF*s, one y_i-*BF* for A_i is involved in an `insert` operation to cover its own digit, the relevant y-*BF* from each column is to return value y in the `query` operation explained in Section III-B.

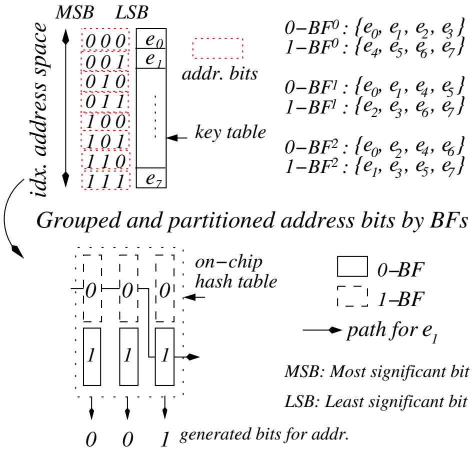
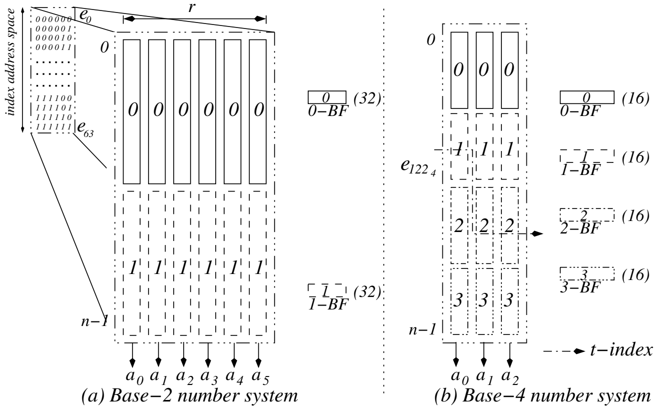

MSB
LSB
idx. address space
000
001
010
011
100
101
110
111
e_0
e_1
e_7
addr. bits
key table
$0\text{-}BF^0: \{e_0, e_1, e_2, e_3\}$
$1\text{-}BF^0: \{e_4, e_5, e_6, e_7\}$
$0\text{-}BF^1: \{e_0, e_1, e_4, e_5\}$
$1\text{-}BF^1: \{e_2, e_3, e_6, e_7\}$
$0\text{-}BF^2: \{e_0, e_2, e_4, e_6\}$
$1\text{-}BF^2: \{e_1, e_3, e_5, e_7\}$
Grouped and partitioned address bits by BFs
on−chip hash table
0 0 0
1 1 1
0−BF
1−BF
path for e_1
MSB: Most significant bit
LSB: Least significant bit
0 0 1 generated bits for addr.

Fig. 2. Partitioning of 8 elements in base-2 with 0-*BF*s and 1-*BF*s.

Fig. 2 shows an example of the base-2 number system with 2^3 elements and three pairs of 0-*BF*s and 1-*BF*s. The index addresses to a table in the upper figure are drawn based on the base-2 number system, where each column has the same number of 0/1 digits. Below, 8 elements are regrouped in every column according to their bits in the column, so that each of 0-BF^vs and 1-BF^vs, $v\in\{0,1,2\}$, has its own set for `insert` as shown in the right side. For instance, suppose $e_1=e_{001_2}$ is to be saved at address 001_2 of off-chip memory. For an MBHT, 0-BF^0, 0-BF^1, and 1-BF^2 from column 0, 1, and 2, respectively, are involved in saving e_1 as shown in the figure. Even if the address space in one column is partitioned by two BFs, they can be accessed with the same memory address by stacking BFs in the following ways: 1) two BFs in separate memory modules are accessed in the interleaving way and 2) two m-bit vectors are glued together so that one memory location has two bits for two BFs.

The base-2 number system used in Fig. 2 can be expanded into an arbitrary number system for the benefit of memory efficiency, as shown in Fig. 3. All address spaces in subfigures are partitioned column-wise and grouped by multi-predicate BFs. The address space for 2^6 elements in the base-2 number



(a) Base−2 number system (b) Base−4 number system

Fig. 3. Conversion of the base-2 number system to base-4 for 64 elements. $n = 2^6$. By (X), X means the number of the same digits in a BF.

system in Fig. 3 (a) is transformed to other address space of base-4 number systems in Fig. 3 (b), resulting in the fewer columns in each address space. However, this transformation does not affect addressing off-chip memory. For example, suppose item e_{011010_2} for base-2 is located at 011010_2. This can be at 122_4 of base-2^2 as shown in Fig. 3 (b).

With this invariant, given the base-b_1 number system and requirement of f, *Linear Property* of Lemma 1 regarding variables m and n claims that even if the number of new BFs, b_2, is increased in a column in new base-b_2, the total memory size for the column remains the same. Although the number of elements to hash for each new BF in the column is reduced, the total number of items for the new column in base-b_2 is the same as that for base-b_1.

In general, considering two MBHTs the total memory usage in bits for the base-b number system as a function of f requirement is calculated as follows:

$$\begin{aligned} M_b &= 2\times C\times B = 2\times\left\{\log_b n\right\}\times\left\{b\left(1.44(n/b)\log_2(1/f)\right)\right\}\\ &= 2\times\left\{\frac{\log_2 n}{\log_2 b}\right\}\times\left\{1.44n\log_2(1/f)\right\}, \end{aligned} \quad (3)$$

where C is the number of columns, and B is the number of bits in a series of BFs in one column. Note that M_b does not consider the memory of an indicator in a *y-BF*, which is minuscule enough to ignore. From this equation, denominator term $\log_2 b$ makes $M_b(f)$ smaller as it increases provided that n and f are constants. This is manifested in Fig. 4 showing the total memory usage in bits considering only BFs in several number systems based on Eq. (3). Along with n axis of b=2, $M_b(f)$ increases greatly for a given f due to $\log_b n$ and n terms in Eq. (3). Similarly, the change rates in axes of f and n for a smaller b are much larger than those of a larger b. Furthermore, the gap of $M_b(f)$ among different bs is large enough that the saved memory can be used to reduce the f-positive of each BF. Therefore, rather than using base-2, using larger base-2^x number system is advantageous because of x times on-chip memory saving. However, choosing the appropriate base system depends on the current technology of memory hardware. For example, b=2^{20}, the largest base system, could be the best choice for n=2^{20} because that gives the highest memory efficiency. In contrast to theoretical benefit, in real hardware it is very hard to probe 2^{20} memory modules for all BFs at the same time in high speed to generate index address. Although this paper does not suggest a solution of choosing the approprioate base, hardware architect must consider the trade-off.

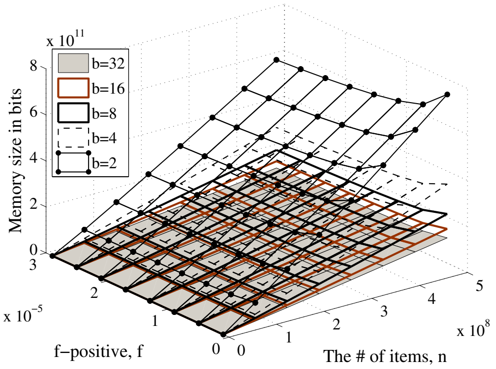


Fig. 4. Memory size $M_b(f)$ for $b = 2, 4, 8, 16,$ and 32 with f and n.

The detailed procedure of the `insert` operation is described in **Algorithm** `insert`. Address A fed to **Algorithm** `insert` is provided by a queue of free addresses. The first vertically-lined **for** loop in it is executed *in parallel* at each column. Also, the second **for** loop is done *in parallel*, as in the conventional BF. Therefore, the time complexity in on-chip memory is Θ(1) on the conditions that hash functions return indexes in constant time, and each column conducts hashing in parallel. Moreover, the number of off-chip memory accesses $Mem[A]$, where Mem is off-chip memory and A is a given address, is exactly 1 because item e is saved in the designated address A as shown in the last line. Therefore, the complexity of **Algorithm** `insert` for off-chip memory access is Θ(1). In contrast, an FHT was calculated to be a time complexity of $O(nk^2/m + k)$, which is not suitable for dynamic update in database for packet processing [18].

Algorithm 1: `insert`(x, e, A)

```
Input: $x$-MBHT $x \in \{l, r\}$, item $e$ and address
       $A = A_0A_1\cdots A_{r-1}$ in base-b
Result: Encoded MBHT about $e$
for column $i = 0$ to $r - 1$ do                /* On-chip Op. */
    for $t = 0$ to $k - 1$ do
        $g = h_t(e)$;
        $A_i$-$BF^i_d[g] = 1$;          /* gth bit in $A_i$-$BF^i_d[g]$ */
    end
end
$Mem[A] = e$ ;                  /* Off-chip Memory Access */
```

B. `Query` *operation in an MBHT*

After all elements are saved contiguously in off-chip memory and encoded in a set of BFs in on-chip memory, the remaining and ultimate goal of an HT is to search for an item by a fast `query` operation. There are two kinds of search patterns: a successful search in which an item is found; and an unsuccessful time-consuming search for an item that does not exist in an HT.

Before we examine two kinds of searches with possible false access, let us introduce definitions of a true index and a false index. A true index, or *t*-index, is defined as a series of indicators resulting from true membership testing. They are interconnected and back-to-back of each other from column 0 to column *r-1*, where *r* is the number of columns in the base-*b* number system, making a sequence of full address bits. The sequence of bits is also matched with an arbitrary memory address associated with an element saved in off-chip memory. Also, the number of returned indicators should be *r*.

For instance, item *e* is to be saved at address 122_4 in base-4 as shown in Fig. 3 (b). In base-4, sequence 122_4 from 1-BF^0, 2-BF^1, and 2-BF^2 are involved as a *t*-index to save the item *e*. From the definition of a *t*-index, we can conclude the following corollary: Once item *e* is saved at address A with a series of $r = \log_b n$ BFs, i.e. y_0-$BF^0 \cdots y_{r\text{-}1}$-$BF^{r\text{-}1}$, in base-*b*, the involved BFs should return $y_0 y_1 \cdots y_{r\text{-}1}$ for the `query` operation of item *e* if membership testing is met as a legacy BF returns 1. Due to the independent and identically distributed (i.i.d) property of BFs, it is possible that irrelevant BFs could return their predicates in the `query` operation.

In addition, a false index, caused by the irrelevant predicates, is defined as the following: In the `query` operation of item *e*, in each column *i* of an MBHT, a group of *BF*s in column *i* not pertaining to a *t*-index for the `insert` operation can return their indicator values. The indicator values in the column *i* could lead to a false index, *f*-index, with other BF^js, $j \neq i$ involved in the `insert` operation. Therefore, an *f*-index is a combination of indicator values of BFs irrelevant and relevant to the `insert` operation of item *e* and by using **MUX** device in each column, BFs responding to membership test for item *e* return their indicator values. Also, the length of an *f*-index should be $r = \log_b n$. From the above two definitions, the numbers of *f*-indexes for a successful and an unsuccessful search are at most *n-1* and *n*, respectively. This means that a *t*-index for an item leads to one off-chip memory access to a wanted item like an LHT and an FHT, while the number of *f*-indexes corresponds to the number of off-chip memory accesses right before an access to the wanted item in a linked list. The next important step is to recognize a *t*-index and annul a series of false positives randomly scattered in an MBHT so that the possibility of a *f*-index can be reduced.

1) False indexing for a successful search in a MBHT: We have explained the definitions of *t*-index and *f*-index and how they can both occur in the `query` operations on an MBHT. Now, we derive and calculate the probability of the number of false accesses in a successful search. In a `query` for a successful search, at least one BF in each column needs to return its indicator value so that the sequence of $A_0 A_1 \cdots A_{r\text{-}1}$ forms the full address *A*, i.e. *t*-index. Furthermore, in case of an *f*-index, false addresses can be created through false positives in each BF of each column can be constituted.

Suppose X_i^s is a random variable of the number of false positives from BFs irrelevant to the *t*-index of a **s**uccessful `query` operation at column *i*. Due to the i.i.d *f*-positives of the BFs, the probability density function of X_i^s is a binomial distribution, $B(b\text{-}1, f)$. Also, assume that the random variable X^s for a **s**uccessful search denotes for the total number of *f*-indexes in a given `query` operation. Then, random variable X^s is defined as the product of random variable X_i^ss, i.e. $(\prod_{i=0}^{r-1}(X_i^s+1))$-1, because of the i.i.d property of each column and the probability of $X^s = x$ is the following

$$
\begin{aligned}
Pr\{X^s{=}x\} &= \sum_{(x_0+1)\cdots(x_{r-1}+1)=x+1} Pr\{X_0^s{=}x_0, \cdots, X_{r-1}^s{=}x_{r-1}\} \qquad (4)\\
&= \sum_{(x_0+1)\cdots(x_{r-1}+1)=x+1} Pr\{X_0^s{=}x_0\} \cdot Pr\{X_1^s{=}x_1\} \cdots Pr\{X_{r\text{-}1}^s{=}x_{r\text{-}1}\}.
\end{aligned}
$$

Also, the mean of X^s is calculated based on the i.i.d property of X_i^s as shown

$$
\begin{aligned}
E[X^s] &= \sum_{t=0}^{n-1} t \cdot Pr\{X^s = t\} = E[(\prod_{i=0}^{r-1}(X_i^s + 1)) - 1]\\
&= \prod_{i=0}^{r-1} E[X_i^s + 1] - 1 = [1 + (b-1)f]^r - 1. \qquad (5)
\end{aligned}
$$

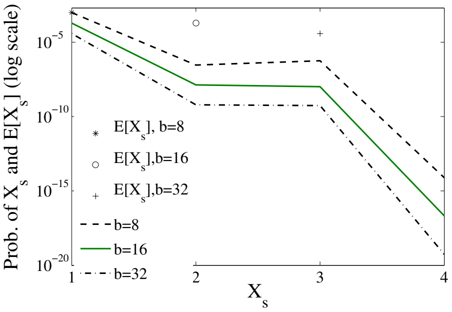


Fig. 5. Probability of X^s, false memory access, in a successful search. $n = 2^{16}$. Required $f{=}2^{-10}$ for an LHT & an FHT

Fig. 5 shows the probabilities for three base systems (2^3, 2^4, and 2^5) derived from Eqs. (4) and (5). For a fair comparison, each memory size of M_{2^3}, M_{2^4}, and M_{2^5} are set equally so that inequality $f_{2^3} > f_{2^4} > f_{2^5}$ is satisfied based on Lemma 2 where f_{2^3}, f_{2^4}, and f_{2^5} are *f*-positives of each BF in base-2^3, base-2^4, and base-2^5, respectively. The lines in Fig. 5 are not shown in monotonic decrease due to binomial coefficient in binomial distribution $B(b\text{-}1, f)$. However, the average value of X^s from Eq. (5) is decreased as the number system of base-*b* increases.

2) False indexing in an unsuccessful search in a MBHT: In addition to ensuring a low probability of more than one access to off-chip memory in a successful search, the design must also ensure the low probability of an unsuccessful search is. Unlike a successful search, an unsuccessful search has no valid index, which means that all BFs return their predicates as an f-positive. However, by definition of f-index, each column should have at least one BF return its predicate as an f-positive, otherwise a group of f-positives can not constitute an f-index. Therefore, we expect a much lower probability because of the product of each independent f-positive probability of BFs.

Let X_i^u denote a random variable of the number of false positives from BFs at column i. Then, the probability density function of X_i^u follows a binomial distribution $B(b, f)$ due to the i.i.d f-positives of the BFs. Also, suppose random variable X^u is the number of f-indexes in an **u**nsuccessful search on an MBHT. Then, random variable X^u can be formulated with random variable X_i^u into $\prod_{i=0}^{r\text{-}1} X_i^u$. In general, the probability of X^u becomes

$$\begin{aligned}Pr\{X^u = x\} &= \sum_{x_0 \cdots x_{r\text{-}1} = x} Pr\{X_0^u = x_0, \cdots, X_{r\text{-}1}^u = x_{r\text{-}1}\} \\ &= \sum_{x_0 \cdots x_{r\text{-}1} = x} Pr\{X_0^u{=}x_0\} \cdot Pr\{X_1^u{=}x_1\} \cdots Pr\{X_{r\text{-}1}^u{=}x_{r\text{-}1}\}.\end{aligned}$$

Finally, the mean of random variable X^u can be calculated with i.i.d property:

$$E[X^u] = \sum_{t=0}^{n} t \cdot Pr\{X^u = t\} = E[\prod_{i=0}^{r-1} X_i^u] = [bf]^r. \qquad (6)$$

Algorithm 2: `query(MBHT,e)`

```
Input: An MBHT and item $e$
Output: Set of $A^b = A_0 \cdots A_{r-1}$ including false indexes
for column $i = 0$ to $r - 1$ in an MBHT do
    for $t = 0$ to $b - 1$ do
        if $e \in t\text{-}BF^i$ then
            $S_{A_i} = S_{A_i} \cup \{t\}$;
        end
    end
end
$S_A = \emptyset$;                  /* Set of $i$-index and $f$-indexes */
$S_A$ = make_paths($S_{A_0}, \cdots, S_{A_{r-1}}$);
return $S_A$;                /* No off-chip memory access */
```

The `query` operation shown in the **Algorithm** `query` only considers on-chip operation and it needs to be called twice on *l-MBHT* and *r-MBHT*. Therefore, the average of random variables $\mathcal{X}^u$ and $\mathcal{X}^s$ for an unsuccessful and successful search, respectively, using two MBHTs are

$$E[\mathcal{X}^u] = 2 \cdot E[X^u] \qquad \text{and} \qquad E[\mathcal{X}^s] = E[X^u] + E[X^s],$$

because for an unsuccessful search both MBTHs do not have the wanted item and for a successful search one of MBHT does not have the wanted item. The time complexity of overall `query` is $\Theta(1)$ on the condition that function `make_paths` making false indexes from set S_{A_i}, $0 \le i \le r\text{-}1$ is performed in constant time as is done in hashing. Also, time complexities of accessing off-chip memory depend on Eq. (5) and (6) for a successful and an unsuccessful search.

C. `Delete` *operation in MBHTs*

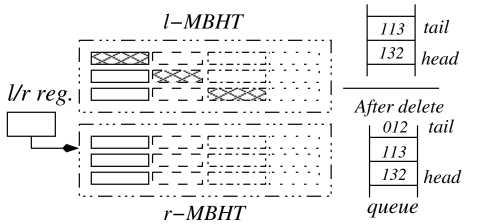


Fig. 6. An example of `delete` for item e located at 012_4 in base-4.

Unlike the two kinds of searches in `query` operations, we consider `delete` operation for a successful deletion. The `delete` operation needs two `query` operations on both *l-MBHT* and *r-MBHT*, where only one of MBHTs has a relevant item e. Fig. 6 shows an example with n=64 for `delete`. Initially, *l-MBHT* has been fully used for `insert`, *l/r-reg.* indicates the *r-MBHT* for future `insert`, and the stack has 113_4 and 132_4 for `insert`. Suppose item e was inserted in $0\text{-}BF^0$, $1\text{-}BF^1$ and $2\text{-}BF^2$ in checked boxes as shown in the figure. Once item e for `delete` operation is confirmed by accessing off-chip memory with the address 012_4, it is to be put on the stack for future `insert`.

Like the `query` operation, if there are an f-index and a t-index associated an item, two accesses are necessary. Therefore, when random variable $\mathcal{Z}$ is denoted as the number of accesses to off-chip memory with both MBHTs, the average memory access for a `delete` operation on the condition that the item exists, i.e. in a successful *delete*, is

$$\begin{aligned}E[\mathcal{Z}] &= \left(\sum_{v=1}^{n} v \cdot Pr\{X^u = v\}\right) + \left(1 + \sum_{v=1}^{n-1} v \cdot Pr\{X^s = v\}\right) \qquad (7) \\ &= [bf]^r + [1 + (b-1)f]^r,\end{aligned}$$

where the first term accounts for an unsuccessful search on one of MBHTs while the second term explains the a successful search on the other.

The detailed procedure of the `delete` operation is shown in **Algorithm** `delete`. The complexity in on-chip memory is $O(1)$ because the complexity of `query` used in the algorithm is $O(1)$. The complexity of memory access is $O(E[\mathcal{Z}])$ on average for a successful *delete*, and it is to be constant as $E[\mathcal{X}^s]$ is $O(1)$.

Algorithm 3: `delete`(*l-MBHT*,*r-MBHT*,*e*)

```
Input: Two MBHTs and item $e$
Result: Update associated BF in each column
$S_{l\text{-}MBHT}$=query($l$-$MBHT$,$e$) ;      /* Only on-chip Op. */
$S_{r\text{-}MBHT}$=query($r$-$MBHT$,$e$) ;      /* Only on-chip Op. */
for $A \in S_{l\text{-}MBHT} \cup S_{r\text{-}MBHT}$ do          /* $A = A_0 A_1 \cdots A_{r-1}$ */
    if $Mem[A] == e$ then          /* Off-chip Mem. Acc. */
        push($A$,queue);              /* push $A$ to queue */
    end
end
```

IV. Analysis and Simulation

In this section, we present analyses of memory efficiency and the average access of search (AAS) to off-chip memory for three schemes; an LHT, an FHT, and an MBHT. Also, we analyze a phenomenon of duplicated keys in an FHT. Finally, we perform one simulation for determining the on-chip memory usage and AAS with three traces. Among a class of universal hash functions, a scheme of [19] we choose for our simulation is suitable for hardware implementation.

A. On/Off-Chip memory usage

In this section, we present and compare the memory usage about on- and off-chip for each scheme. Given f-positive f=2^{-w} and the number of elements n, the memory usages in bits of an LHT and an FHT are the product of the number of layers, L_v, and the number of bits in one layer, B_v, $v \in \{L, F\}$ as follows:

$$M_L = R_L \times B_L = \Big\{\log_2 n\Big\} \times \Big\{1.44 n \log_2(1/f)\Big\} \text{ and}$$

$$M_F = R_F \times B_F = \Big\{\log_2 n + 4\Big\} \times \Big\{1.44 n \log_2(1/f)\Big\}, \qquad (8)$$

respectively, where 4 accounts for the number of bits in a counter. Thus, based on Eqs. (8) and (3, memory efficiency ratio $R_{M,F}$ of M_M to M_F in base-2^x becomes

$$R_{M,F} = \frac{M_F}{2\{\log_b n \cdot 1.44 n(\log_2(1/f)+\alpha)\}} = \frac{x(\log_2 n+4)w}{2 \cdot \log_2 n(w+\alpha)} \approx \frac{x(\log_2 n+4)}{2\log_2 n}, \qquad (9)$$

where α=$\log_2(b$-$1)$ due to coefficients in the binomial functions, and the size of a queue, $n \log_2 n$, not is considered for on-chip memory comparision. Also, the memory efficiency ratio $R_{M,L}$ of M_M to M_L is

$$R_{M,L} = \frac{M_L}{2 \times \{\log_b n \cdot 1.44 n(\log_2(1/f)+\alpha)\}} = \frac{x}{2} \cdot \frac{w}{w+\alpha} \approx \frac{x}{2}. \qquad (10)$$

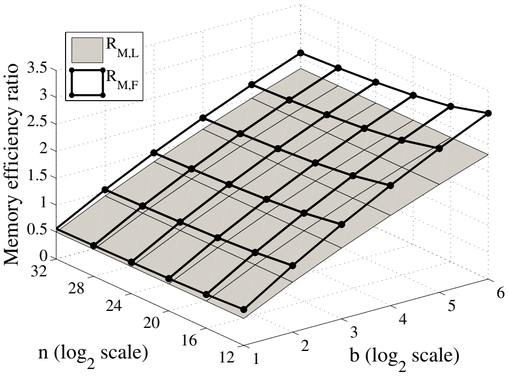


Fig. 7. Memory efficiency ratios of $R_{M,L}$ and $R_{M,F}$ with various b and n. f=2^{-20}.

Fig. 7 shows two ratios, $R_{M,F}$ and $R_{M,L}$, calculated from Eq. (9) and (10) in the range [2^1:2^6] for base-b and in the range [2^{12}:2^{32}] for n. In the figure, we see that without doubt the turning point for a better memory efficiency ratio surely begins at b=2^3 due to a set of two MBHTs. However, due to coefficients in binomial functions $B(b$-$1, f)$ and $B(b, f)$ the acquired memory gain demonstrates the less probability values of X^s and X^u at base-2^4 by increasing memory of each BF. Given b, the memory gain does not change much as shown in the figure, although the change rate of memory gain for a given n is manifested along the b axis. Thus, compared to an LHT and an FHT, Fig. 7 proves that our approach using a multi-predicate BF can gain much memory as long as a larger base number system is used.

In terms of off-chip memory usage, Fig. 10 in [1] shows an FHT has up to 3 times of duplicate keys than an LHT. By empirical testing on various k in a hundred runs as shown in Fig. 8, we realized that the average number of saved keys is proportional to k, which now is proportional to $\log_2(1/f)$ according to Eqs. (1) and (2). This means that given requirement of a high-precision search, i.e. small value of f, the larger is $\log_2(1/f)$, the larger is k, leading to more duplicate keys on off-chip memory. Note that the number of keys to save is the same as the # of saved keys in an MBHT as shown in two bottom lines of the figure.

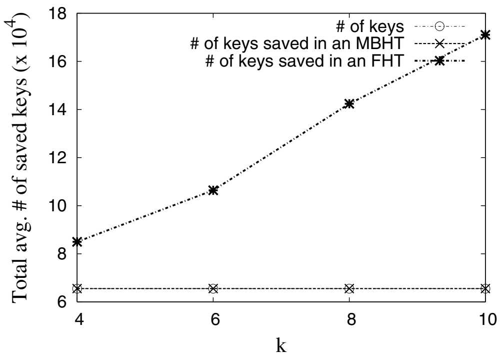


Fig. 8. Duplicate keys in off-chip memory according to k when n=2^{16}.

B. Average access of search

Let us define the average access of search (AAS) as the number of off-chip memory accesses under different successful search rates. For AAS of an LHT, an FHT, and an MBHT, denote $\mathbb{A}_v^s$ and $\mathbb{A}_v^u$, $v \in \{L, F, M\}$, average successful and unsuccessful search accesses, respectively, where L, H, and M are abbreviations of an LHT, an FHT, and an MBHT, respectively. Besides $\mathbb{A}_L$ and $\mathbb{A}_F$ calculated in [20] and [1], Eq. (6) and Eq. (5) for the average off-chip memory access in unsuccessful and successful searches, respectively, are used to get AAS $\mathbb{A}_M$ as following:

$$\mathbb{A}_M = p_s(E[X^s]+1) + p_u E[X^u] = p_s[1+(b\text{-}1)f]^r + (1\text{-}p_s)[bf]^r. \qquad (11)$$

Fig. 9 shows the expected search accesses, $\mathbb{A}_L$, $\mathbb{A}_F$, and $\mathbb{A}_M$, in terms of the number of off-chip memory accesses for an LHT, an FHT, and an MBHT. For a fair comparison the memory size of M_L, M_F, and M_M is equal. As noted in Eq. (8), consideration of 4 bits for a counter affects $\mathbb{A}_F$ in a situation that M_L, M_F, and M_M are set equal each another. Therefore, $\mathbb{A}_F$ marked with 'corrected FHT' in Fig. 9 has a

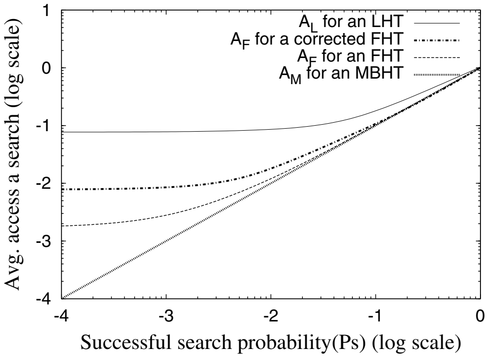


Fig. 9. Average access of search for an LHT, an FHT, and an MBHT. n=2^{16} and f=2^{-10} for an LHT and an FHT. b=2^4 for an MBHT.

larger value than that marked with 'FHT' used by Song *et al.* [1] because they did not consider counter bits in on-chip memory. The result in Fig. 9 indicates that the lower the successful search rate, the better the access time performance of an MBHT compared to the performance of other schemes.

Operation	`insert`	`query`	`delete`
LHT	$O(1)$	$O(1)$	$O(1)$
FHT	$O(nk^2/m + k)$	$O(1)$	$O(nk^2/m + k)$
MBHT	$\Theta(1)$	$O(1)$ †	$O(1)$

† In detail, $\Theta(p_s(1 + E[\mathcal{X}^s]) + p_u E[\mathcal{X}^u])$.

TABLE I

Operation comparisons of an LHT, an FHT, and an MBHT.

Table I summarizes the complexities of off-chip memory access regarding `insert, query` and `delete` operations in an LHT, an FHT, and an MBHT. The big difference is in an FHT, which involves the labored complexities of `insert` and `delete` operations depending on variables n, k, b, and m. In contrast, the complexities of an MBHT and an LHT are constant.

C. URL switching

As one application of an MBHT to packet processing, we used NePSim [21] for URL switching where all the incoming packets to a switch are parsed and forwarded according to URL. This kind of switching is a commonly used content-based load balancing mechanism [7, 9]. Kachris *et al.* [9] used a simlpe XOR hash to reduce the collisions among Block RAMs in connection manager for web switching, and Prodanoff *et al.* in [8] proposed URL signatures using CRC32 to reduce the size of routing tables and aggressive hashing with chaining of a linked list to speed-up routing lookups in large-scale content distribution networks.

Table II shows the memory size in bytes of an LHT, an FHT, and an MBHT for three trace databases on the condition that requirement of f is 2^{-20} and the load factor becomes 0.034 accordingly. Each trace of UC Berkeley, NLANR, and CA*netII has 149,344, 504,967, and 2,552,045 URLs, respectively. We can realize that we have at most 1.7 times on-chip memory reduction at an MBHT in base-16 against an LHT as shown in the table. If comparison is set for an FHT, about 2 times of the memory reduction is observed due to consideration of counter bits in an FHT.

URL Traces	LHT	FHT	MBHT base-8	MBHT base-16
UC Berkeley[22]	9024KB	11124KB	6860KB	5393KB
NLANR[2]	33634KB	40735KB	25570KB	20102KB
CA*netII[23]	190953KB	226841KB	1145171KB	114127KB

TABLE II

On-chip Memory usage for three traces. The load factor is 0.034, K=1024.

While authors in [21] validated NePSim with SDRAM, SRAM and six microengines against the IXP 12000 architecture in terms of performance and power, we measured the number of accesses to SDRAM with NLANR trace on the condition that an LHT, an FHT, and an MBHT were implemented in SRAMs. Especially, given a query an MBHT is to return indexes *in parallel* with a set of SRAMs. Table III shows the measured accesses to SDRAM in NePSim with NLANR. The first row is the average access for a successful search. While an FHT needs 2.4E-3 extra accesses on average for a successful search, our MBHT with b=16 asks 9.2E-5. Although this value could be minuscule, when it comes to the difference between the numbers of off-chip accesses in an FHT and an MBHT, the gap between them is 2117.

Schemes	LHT	FHT	MBHT(b=8)	MBHT(b=16)
AAS*	1.026306	1.002472	1.002411	1.000092
Total # of Acc.°	968861.7	946231.9	946303.9	944114.4

° It means the total number of off-chip accesses provided the URL queries of NLANR.

TABLE III

AAS in a successful search of NLANR trace about three schemes. f=2^{-10}

V. Related Works

Since the burgeoning interest in a BF in late 1990's, several types of BFs have been suggested in various application domains, despite disadvantages such as false positives and the incapability of *delete* operation. Besides our main comparative literature [1] and packet processing literature [14, 15, 24], literature on several applications using BFs will be discussed in this section as in [17].

A legacy BF does not support deletion operation because a bit-location in a bit-vector can be overlapped by more than one key. To avoid this problem, Fan *et al.* [25] introduced the idea of a counting BF, in which each entry in the BF is a counter of 4 bits costing 4 times memory size. However, our MBHT does not adapt to use counters. Bonomi *et al.*

[26] introduced Approximate Concurrent State Machines. While similar in spirit to BFs, the scheme is based on a combination of hashing and fingerprints, using *d*-left hashing to obtain a near-perfect hash function in a dynamic setting. Surprisingly, its data structure takes much less space than a comparable counting BF. Cohen and Mattias [27] introduce Spectral Bloom Filter, an extension of the original BF to multi-sets, allowing the filtering of elements whose multiplicities are below a threshold given at query time. Using memory only slightly larger than that of the original BF, SBF supports queries on the multiplicities of individual keys with a guaranteed, small error probability.

Besides theoretical approaches, Metwally *et al.* [28] provide duplicate detection in data streams of the World Wide Web utilizing various applications, including fraud detection. In an application of overlay networks, continuous reconfiguration of virtual topology by overlay management strives to establish paths with the most desirable end-to-end characteristics. The approximate reconciliation tree of Byers *et al.* [29] uses BFs on top of a tree structure to minimize the amount of data transmitted for verification.

VI. Conclusion and Future Work

We have proposed a novel hash architecture, generating an off-chip memory address with a set of multi-predicate BFs in base-$b(=2^x)$ number system with n items to hash. The BFs work systematically, or *in parallel*, in the `query` operation so that a subset of them in row i determine A_i, which is a part of a whole address $A^b{=}A_0 \cdots A_{r-1}$. From Lemmas 1 and 2, we realized that adapting a larger base number system saves significant on-chip memory by approximately $\frac{x(\log_2 n+4)}{2\cdot\log_2 n}$ and $\frac{x}{2}$ times compared to an FHT and an LHT, respectively. The saved memory, therefore, can be utilized to decrease the probabilities of $\mathcal{X}^s$ and $\mathcal{X}^u$ for the `query` operation In addition, we provided the `insert` and `delete` operations on MBHTs with complexities of $\Theta(1)$ and $O(1)$, respectively, while those of an FHT are $O(nk^2/m + k)$ and $O(nk^2/m + k)$, respectively. This benefit manifests itself in incremental updates on an HT for packet processing because frequent updates by `insert` and `delete` are as important as fast operation of `query` in that environment [18]. In analysis, we derived a memory efficiency ratio of an MBHT over an LHT and an FHT, and showed that base-2^3 is the starting point of better memory efficiency for an MBHT. Furthermore, simulation with NLANR showed an MBHT had less off-chip memory accesses in URL switching by utilizing the saved memory. As a future work, we plan to use our scheme in wireless sensor network where enormous data centric routing is necessary.

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 *SIGCOMM '05*.

[2] NLANR Sanitized Cache Access Logs. [Online]. Available: ftp://ircache.nlanr.net/Traces

[3] F. Baboescu and G. Varghese, "Scalable Packet Classification," *IEEE/ACM Trans. Netw.*, vol. 13, no. 1, pp. 2–14, 2005.

[4] A. Basu and G. Narlikar, "Fast Incremental Updates for Pipelined Forwarding Engines," *IEEE/ACM Trans. Netw.*, vol. 13, 2005.

[5] J. Xu and M. Singhal, "Cost-effective flow table designs for high-speed routers: Architecture and performance evaluation," *IEEE Trans. Comput.*, vol. 51, no. 9, pp. 1089–1099, 2002.

[6] W. F. F. Chang and K. Li, "Approximate Caches for Packet Classification," in *INFOCOM 2004*, vol. 4, March 2004, pp. 2196–2207.

[7] A. Apostolopoulos, D. Aubespin, V. Peris, P. Pradhan, and D. Saha, "Design, Implementation, and Performance of a Content-Based Swith," in *INFOCOM 2000.*

[8] Z. G. Prodanoff and K. J. Christensen, "Managing Routing Tables for URL Routers in Content Distribution Networks," *Int. J. Netw. Manag.*, vol. 14, no. 3, 2004.

[9] C. Kachris and S. Vassiliadis, "Design of a Web Switch in a Reconfigurable Platform," in *ANCS '06*. New York, NY, USA: ACM Press, 2006.

[10] S. Dharmapurikar, P. Krishnamurthy, and D. E. Taylor, "Longest Prefix Matching using Bloom Filters," in *SIGCOMM '03*, 2003.

[11] V. Ravikumar and R. Mahapatra, "TCAM Architecture for IP Lookup using Prefix Properties," *MICRO, IEEE*, vol. 24, no. 2, 2004.

[12] V. C. Ravikumar, Rabi N. Mahapatra and Laxmi Narayan Bhuyan, "EaseCAM: An Energy and Storage Efficient TCAM-Based Router Architecture for IP Lookup," *IEEE Trans. Comput.*, vol. 54, 2005.

[13] K. Lakshminarayanan, A. Rangarajan, and S. Venkatachary, "Algorithms for Advanced Packet Classification with Ternary CAMs," in *SIGCOMM '05*. New York, NY, USA: ACM Press, 2005.

[14] T. S. Sarang Dharmapurikar, Praveen Krishnamurthy and J. Lockwood, "Deep Packet Inspection using Parallel Bloom Filters," in *MICRO 37*. New York, NY, USA: ACM Press, 2004, pp. 52–61.

[15] H. C. Deke Guo, Jie Wu and X. Luo, "Theory and Network Applications of Dynamic Bloom Filters," in *INFOCOM 2006*.

[16] D. E. Knuth, *The Art of Computer Programming*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 1978.

[17] A. Broder and M. Mitzenmacher, "Network Applications of Bloom Filters: A Survey," pp. 485–509, 2002. [Online]. Available: citeseer.ist.psu.edu/broder02network.html

[18] A. Basu and G. Narlikar, "Fast Incremental Updates for Pipelined Forwarding Engines," *IEEE/ACM Trans. Netw.*, vol. 13, 2005.

[19] E. F. M. V. Ramakrishna and E. Bahcekapili, "A Performance Study of Hashing functions for Hardware Applications," in *Proceedings of Int. Conf. on Computing and Information*, 1994, pp. 1621–1636.

[20] T. H. Cormen, C. E. Leiserson, and R. L. Rivest, *Introduction to Algorithms*. New York: McGraw-Hill, 1990.

[21] Y. Luo, J. Yang, L. N. Bhuyan, and L. Zhao, "NePSim: A Network Processor Simulator with a Power Evaluation Framework," *IEEE Micro*, vol. 24, no. 5, pp. 34–44, 2004.

[22] Monthly Log Files 2000, Computer Science Division, University of California, Berkeley. [Online]. Available: http://www.cs.berkeley.edu/logs/http

[23] Sanitized Log Files from Canada's Coast to Coast Broadband Research Network (CA*netII). [Online]. Available: http://ardnoc41.canet2.net/cache/squid/rawlogs

[24] J. X. Abhishek Kumar and E. W. S. Zegura, "Efficient and Scalable Query Routing for Unstructured Peer-to-Peer Networks," in *INFOCOM 2005*, 2005, pp. 13–17.

[25] L. Fan, P. Cao, J. Almeida, and A. Z. Broder, "Summary Cache: a Scalable Wide-Area Web Cache Sharing Protocol," *IEEE/ACM Trans. Netw.*, vol. 8, no. 3, pp. 281–293, 2000.

[26] Flavio Bonomi, M. Mitzenmacher, R. Panigrahy, S. Singh and G. Varghese, "Beyond Bloom Filters: From Approximate Membership Checks to Approximate State Machines," in *SIGCOMM '06*.

[27] S. Cohen and Y. Matias, "Spectral Bloom Filters," in *SIGMOD '03: Proceedings of the 2003 ACM SIGMOD international conference on Management of data*. New York, NY, USA: ACM Press, 2003.

[28] A. Metwally, D. Agrawal, and A. E. Abbadi, "Duplicate Detection in Click Streams," in *WWW '05: Proceedings of the 14th international conference on World Wide Web*, 2005.

[29] J. Byers, J. Considine, M. Mitzenmacher, and S. Rost, "Informed Content Delivery Across Adaptive Overlay Networks," in *SIGCOMM '02*. New York, NY, USA: ACM Press, 2002, pp. 47–60.