# Bloom Tree: A Search Tree Based on Bloom Filters for Multiple-Set Membership Testing

MyungKeun Yoon JinWoo Son Seon-Ho Shin
Department of Computer Engineering
Kookmin University, Seoul, Korea of Republic
Email: {mkyoon, sonjinwoo, shshin}@kookmin.ac.kr

***Abstract*—A Bloom filter is a compact and randomized data structure popularly used for networking applications. A standard Bloom filter only answers yes/no questions about membership, but recent studies have improved it so that the value of a queried item can be returned, supporting multiple-set membership testing. In this paper, we design a new data structure for multiple-set membership testing, Bloom tree, which not only achieves space compactness, but also operates more efficiently than existing ones. For example, when existing work requires 107 bits per item and 11 memory accesses for a search operation, a Bloom tree requires only 47 bits and 8 memory accesses. The advantages come from a new data structure that consists of multiple Bloom filters in a tree structure. We study a theoretical analysis model to find optimal parameters for Bloom trees, and its effectiveness is verified through experiments.**

## I. Introduction

Traffic classification and packet processing in high-speed networks involve fast table lookup. For a given key such as IP address, TCP 5-tuples, or flow id, its associated value should be returned fast in deterministic time. Recent studies show that this lookup process can be efficiently implemented with Bloom filter-based data structures [1], [2], [3], [4], [5], [6], [7]. In this paper, we design a new data structure by combining Bloom filters with a search tree, which can improve both memory space utilization and the number of memory accesses.

A Bloom filter is a compact and randomized data structure that supports membership testing [8]. As Bloom filters can save memory space and take a constant processing time, they are popularly used in general computer systems as well as in networking applications. A Bloom filter is implemented in a bit array. For a set of items, the key of each item is encoded into the bit array with multiple hashing. After the encoding process finishes, the Bloom filter can answer yes/no questions about the membership of queried items. A nice property is that the key length can be any size. In networking applications, the key can be an IP address, port number, flow id, URL, etc.

The classical Bloom filter can answer only a yes/no question. Recent studies propose variants of Bloom filters to solve problems associated with membership testing [1], [3], [6], [7], [9], [10], [11], [12], [13]. One such problem is fast table lookup in deterministic time where each item has its own key and associated value. We assume that the value may range from tens to several thousand depending on applications. Then, an independent Bloom filter can be built per value by encoding all the keys having that value into the bit array of the corresponding filter. When the value of a key is queried, we look up all the Bloom filters. If the key belongs to only one filter, we conclude that the item's value is correctly returned with a certain probability. This is called *multiple-set membership testing* [1], and many schemes have been proposed to solve the problem efficiently, based on Bloom filters [1], [2], [3], [7].

978-1-4799-3360-0/14/$31.00 © 2014 IEEE

Networking applications of multiple-set membership testing include frame forwarding in a layer-2 switch [1], IP routing [3], [5], [14], URL classification [1], approximate state machines [2], to mention a few. Various Bloom filter-based data structures have been proposed for the applications. The design goal is to make a compact data structure so that it can be located in fast but small on-chip memory such as SRAM, which can provide fast table lookup by eliminating accesses to large but slow off-chip memory. Although TCAMs can be used for the same purpose, we do not consider them in this paper, as not in the previous work [1], [2], [3], [4], because of their high power consumption and low density.

There are three performance metrics for Bloom filter-based data structures for multiple-set membership testing: capacity, error rate, and memory access. The capacity means the maximum number of items encoded into the data structure. We assume that the available memory size is fixed, generally less than a few mega bytes because of the size-limited on-chip memory. The capacity can also be measured by the number of bits required to store an item. The error rate can be seen as the probability that the data structure returns incorrect values about queried items. The memory access is required when a new item is encoded, or the value of an encoded item is queried. In any case, the memory access means the total number of memory accesses to perform the task, which is equivalent to the number of hash operations. Three performance metrics are closely related, and the design goal is to maximize the capacity and minimize the memory access and error rate.

The major technical challenge is how to maximize the capacity and minimize the memory access and error rate at the same time. The past research meets the challenge by designing new data structures with multiple Bloom filters and distinct encoding schemes [1], [2], [3], [7]. This paper adds a new member that not only increase the capacity but also reduce the memory access. Our major contribution is a new methodology to combine Bloom filters with a search tree, *Bloom tree*, which use the memory space more efficiently and require less memory accesses. For example, a Bloom tree can store 22,360 items, and process a query with 7 memory accesses, when the memory size is 1 mega bits, a tolerable error rate is $10^{-6}$, and there are 128 multiple-sets (or distinct return values). Under the same condition, the current state of the art can store 9,868 items, and process a query with 11 memory accesses. A Bloom tree can be implemented with one bit-array, shared by multiple hash functions, which can be processed in parallel, and the load is balanced to evenly scatter '1' bits over the bit-array. We study a theoretical analysis model to find optimal parameters for Bloom trees, and its effectiveness is verified through experiments.

The rest of this paper is organized as follows. Section II discusses related work. The motivation of this work is given in Section III. Section IV describes the proposed Bloom trees. Section V shows experimental evaluation, and Section VI draws conclusions.

## II. Related Work

### *A. Bloom Filter & Membership Testing*

A Bloom filter is a randomized data structure that supports membership testing [8]. For a set of $n$ items, $X = \{x_0, x_1, ..., x_{n-1}\}$, the Bloom filter first records all members. This step is also called *encoding* or programming. Once the encoding is completed, the Bloom filter can answer a membership query for a given item $x$, "does $x$ belong to X?"

A Bloom filter uses a bit array $B$ of size $m$, which is initialized to zeros. The $i$th bit in the array is denoted as $B[i]$, $0 \leq i \leq m-1$. The Bloom filter uses $k$ independent hash functions. We denote the $j$th hash function as $h_j$, $0 \leq j \leq k-1$. During the encoding step, we program each member $x_r$ in the bit array by setting $B[h_j(x_r)] := 1$ for $0 \leq j \leq k-1$ and $0 \leq r \leq n-1$. When a membership query is asked for $x$, every $B[h_j(x)]$ is checked for $0 \leq j \leq k-1$. If all the bits are set to one, the Bloom filter gives a positive response that $x$ may belong to the set.

A false positive may happen when a Bloom filter gives a positive response. A false positive means that the filter erroneously gives a positive answer to the membership query although the queried item is not a member of the set. The false-positive rate, denoted as $f$, can be computed as

$$f = (1-(1-\frac{1}{m})^{n\times k})^k \approx (1-e^{-\frac{n\times k}{m}})^k \tag{1}$$

where $f$ is minimized when $k = \frac{m}{n} \times ln2$ [15].

### *B. Multiple-Set Membership Testing*

A classical Bloom filter supports membership testing, but it can only answer yes/no questions about membership. Recent studies propose Bloom filter-based data structures to support table lookup operations beyond membership checking [1], [2], [3], [7], [14]. Now, each item has a pair of (key, value), and the key is used for encoding. The data structures should return the value for the queried key in deterministic time.

Packet classification and forwarding is a good example to show how multiple-set membership testing can be applied to networking applications [3], [14]. For an incoming packet, switches and routers need to select one of their interface to forward it. Some information from the packet is used as a key, based on which the classification is performed. The destination IP address, TCP 5-tuple, or flow id can be the key, and the outgoing interface for the packet can be seen as the value of the corresponding key. Then, an independent Bloom filter can be built for each outgoing interface, or each value, and all packets have been sent over the interface are encoded into the corresponding Bloom filter. Therefore, a Bloom filter cache is built for every outgoing interface. If a packet arrives, the switch first looks up all the Bloom filters. If exactly one filter returns a positive response, the packet is forwarded over the corresponding interface.

When a return value is static and of a small range, a Bloomier filter can be used. This is a group-coded data structure using multiple Bloom filters, but the mechanism is inefficient for fast processing [16]. Bonomi *et al.* propose methods to approximate state machines, which can track the state of a large number of flows simultaneously. A stateful Bloom filter approach is proposed, but its efficiency is not better than $d$-left hashing [2].

The table lookup of IP routing is based on longest prefix matching. Using multiple Bloom filters can help fast lookup over various prefix bits of different length in parallel. Song *et al.* recently study this problem for IPv6, and propose a new data structure called distributed and load balanced Bloom filters (DLB-BF) [7]. The multiple Bloom filters are combined into virtually one filter where every Bloom filter has the same load.

### *C. Combinatorial Bloom Filter (COMB)*

Hao *et al.* apply multiple-set membership testing to a web traffic classification and measurement system. The system classifies URLs into different nonoverlapping groups such as "e-commerce," "search," etc. The number of groups may range tens to several thousand, and each URL is assigned a group id. The objective is to measure the number of packets belonging to each group, in which combinatorial Bloom filters (COMB) are proposed for multiple-set membership testing [1].

To the best of our knowledge, COMBs are the current best data structure for multiple-set membership testing. Suppose that there are $g$ groups. Unlike the previous approaches of [3], [14] that would build $g$ independent Bloom filters, a COMB builds $f$ Bloom filters such that $\binom{f}{\theta} \geq g$ should be satisfied; $\theta$ is a tuning parameter. Note that each of $f$ Bloom filters represents a bit; exactly $\theta$ bits are set to '1' and others to '0', which forms a fixed-weight code for all groups. Note that each of $\binom{f}{\theta}$ codes matches a distinct group id. Therefore, the value of any item can be represented by $f$ bits. The main idea is that a false-positive error of any Bloom-filter can be detected if the number of '1' bits is not equal to $\theta$. An error corrected COMB (ECOMB) is a more sophisticated version in which a fixed-weight error-correcting code is used. The memory access is traded with the capacity.

## III. Motivation and Definition

We first motivate the concept of Bloom trees in which multiple Bloom filters form a search tree. We then introduce some definitions and frequently used notations.

### A. Motivation

Existing Bloom filter-based methods for multiple-set membership testing build multiple Bloom filters to represent different groups, or distinct return values. A recent study shows that adding a coding theory can increase the capacity and decrease the memory access [1]. Our motivation for this study is that adding the concept of a search tree could further improve the methods for multiple-set membership testing.

Our basic solution is to create a binary search tree in which each node is a Bloom filter. The leaf nodes represent distinct values. For a pair of (key, value) given, a unique path is determined from the root to a leaf node, based on the value. Therefore, the tree's height is determined by the number of different groups. Each node on the path remembers the key by encoding it into the node's Bloom filter. When the value of a given key is required, the membership testing is performed repeatedly from the root to a leaf node. A membership testing reduces the remaining search space by half if a false positive does not happen. A nice property is that the effect of a false positive at parent nodes can be diluted at the membership testing of their child nodes with a high probability. The binary tree can be extended to a general $d$-ary tree, which can efficiently be implemented as hardware. To the best of our knowledge, Bloom trees are the first for the problem of multiple-set membership testing by combining Bloom filters with a search tree.

### B. Definition

We define the problem of multiple-set membership testing. The performance metrics are defined, and some notations are introduced.

In this paper, we study the problem of membership testing for multiple-sets. Every item has a pair of (key, value), and we assume that the value is represented as one of $g$ integers from {0,1,...,g-1} without loss of generality. Therefore, we can classify items into $g$ groups by their value. The $i$th group is denoted as $S_i$, and all the items of value $i$ belong to $S_i$. We assume that the value of a key does not change once the (key, value) is associated, as assumed in [1]. However the value change can be supported if counting Bloom filters are used instead of Bloom filers [17].

Our goal is to design an efficient data structure that can return the value of a queried key. The correct value should be returned with a certain probability. This approximate lookup table involves two different types of errors, false positives and classification failures. A false positive happens when the table returns an arbitrary value for the queried item, but the item was never encoded a priori in the table. This is the same type of false-positive errors as in classical Bloom filters. A classification failure happens when the table finds more than one value matching the queried item. In that case, the table cannot determine which one is the right value, and it returns an error message. We denote the tolerable probabilities for false positives and classification failures by $p_f$ and $p_c$, respectively.

TABLE I
Frequently used notations

| | |
|---|---|
| $g$ | number of multiple groups, or distinct possible values |
| $m$ | available memory size in bits |
| $p_f$ | probability of false positives |
| $p_c$ | probability of classification failures |
| $u_c$ | upper bound of $p_c$ |
| $n$ | capacity, number of items encoded in a Bloom tree |
| $a_e$ | number of memory accesses for encoding a new item |
| $a_{cm}$ | number of memory accesses for checking the value of a member item that has been encoded |
| $a_{cn}$ | number of memory accesses for checking the value of a nonmember item that has not been encoded |
| $l$ | height of a complete $d$-ary tree |
| $p_0$ | probability that a bit is '0' in a Bloom tree |
| $p_1$ | probability that a bit is '1' in a Bloom tree, $p_1 = 1 - p_0$ |
| $k_i$ | number of hash functions for a Bloom filter node at level $i$ |
| $k$ | number of hash functions for encoding an item, $k = \sum_{i=0}^{l} k_i$ |
| $t$ | number of memory accesses processed in parallel |

We use $m$ to denote the available memory size in bits. In this paper, we assume that $m$, $g$, $p_f$, and $p_c$ are given parameters when data structures for multiple-set membership testing are designed. We say a data structure satisfies $(m, g, p_f, p_c)$ condition if the data structure can be built with $m$ bits that correctly returns the value of a queried item with a probability greater than $(1-p_c)$, and a false positive does not happen with a probability greater than $(1-p_f)$.

The performance can be measured by two metrics, capacity and memory access. The capacity, $n$, is defined as the maximum number of items encoded with $m$ bits, while $(m, g, p_f, p_c)$ condition is satisfied. The memory access is of two types; 1) when a new item is inserted and encoded, 2) when the value of a queried item is checked. We use $a_e$ and $a_c$ to denote the number of memory accesses for insertion and checking, respectively. Table I shows some notations frequently used in this paper.

## IV. Design of Bloom Tree

We first propose the basic structure of Bloom trees, and the encoding and checking processes are explained in detail. Then, we study an analysis model and show how to maximize the capacity by using the analysis. Finally, we propose a method for load balancing and hardware-friendly implementation.

### A. Basic Structure

A Bloom tree is a $d$-ary complete search tree. Storing the information about (key, value) pairs for items, a Bloom tree supports approximate table lookup. When the key of a queried item is given, it should return the correct value with a certain probability. A distinguished feature is that a Bloom tree is constructed with $values$ instead of keys unlike traditional search trees.

Each node in a Bloom tree is a Bloom filter. Every internal node has $d$ independent Bloom filters, or $d$ distinct sets of hash functions, each of which represents one edge from that node to its child node. If all of the hash functions in a set indicate

'1' bits, we move on to the corresponding child node to further continue an encoding or checking process.

Suppose that we build a Bloom tree that should satisfy $(m, g, p_f, p_c)$ multiple-set membership testing. First, a complete $d$-ary tree is built with $g$ leaf nodes. We assume that $g$ is a power of $d$ for simplicity, and then the height of the tree is $l = log_d(g)$. For the case when $g$ is not a power of $d$, our solution is to increase the tree height by using $l = \lceil log_d(g) \rceil$. Since the number of leaf nodes is larger than $g$, we choose the first $g$ leaf nodes from the left. Fig. 1 shows the construction of a binary Bloom tree ($d = 2$) for $g = 6$.

Each node of a Bloom tree is a Bloom filter. We assume that nodes at the same level of the tree have the same size in bits and the equal number of hash functions as Bloom filters. We use $m_i$ and $k_i$ to denote the number of bits and the number of hash functions assigned to a node at level $i$, $0 \le i \le l$, and every bit is initialized to '0'. Therefore, $m = \sum_{i=0}^{l} d^i \times m_i$ for a complete full tree.

Each internal node at level $i$, $0 \le i < l$, has $d$ sets of hash functions. Each set has its unique hash functions. We use $H_j^i = \{h_{(j,0)}^i, h_{(j,1)}^i, ..., h_{(j,k_i-1)}^i\}$ to denote the $j$th set of hash functions at level $i$, $0 \le j < d^{i+1}$; $h_{(j,r)}^i$ is the $r$th hash function in the hash set. Unlike the internal nodes, each leaf node at level $l$ has only one set of $k_l$ hash functions, denoted as $H_0^l = \{h_0^l, h_1^l, ..., h_{k_l-1}^l\}$ because only one membership testing is performed at a leaf node. The total number of hash sets is $O(d^l)$. We propose a *virtual key* that can reduce the number of hash sets from $O(d^l)$ to $O(l)$, which is explained in the next section.

Efficient hash value generators should be supported for Bloom trees. This problem has already been solved by previous work [1], [18], [19], and we prefer to use it. In this paper, we adopt the idea of [1] in which lots of hash values can be generated from XORing a subset of the hash values from seed hash functions.

### B. Encoding

Suppose that a new item, $x = (y, v)$, arrives to be encoded into a Bloom tree where $y$ and $v$ are the key and value. In a Bloom tree, the path from the root to a leaf node is determined by $v$. At each internal node, we choose only one of the hash function sets that represents the edge from the current node to its child node following the path, and $y$ is encoded into that Bloom filter of the hash function set at the current node. For example, if the $j$th hash set is chosen at the root node, we set all the indexing bits to '1's that are computed from the hash functions belonging to $H_j^0$. We denote this as $H_j^0(y) := 1$. We repeat this encoding process from the root to the last internal node at level $l-1$. Finally, $y$ is encoded into the Bloom filter of the leaf node at level $l$ that represents $v$, by performing $H_0^l(y) := 1$. Fig. 1 shows the encoding process for $(y, v = 3)$.

Since one Bloom filter encoding is required for each level in the tree, we can count the total number of memory accesses per encoding, $a_e$, by

$$a_e = \sum_{i=0}^{l} k_i. \tag{2}$$

We should consider load-balancing in encoding items, and the '1' bits should be evenly distributed over the memory space. In a Bloom tree, this is achieved by using one large bit-array, $B[m]$, shared by all the multiple hash functions from any level in the tree. A hash result indicates a random position in the bit array, $B[i]$, $0 \le i \le m-1$. This encoding strategy will evenly distribute '1' bits over $B[m]$ even when only a small number of the values are associated with a majority of the keys. Although multiple bit arrays seem to exist at different levels in the tree in Fig. 1, they actually form a large bit-array.

### C. Checking

For a queried key, $y$, we check the Bloom tree to return its corresponding value. There can be two cases for checking; 1) $y$ has already been encoded, and 2) $y$ has not. For the first case, the Bloom tree is able to return the right value for $y$ with a certain probability. Even in the second case, a false-positive can be returned.

For the given $y$, a search operation begins at the root node by checking the independent $d$ sets of hash functions, $H_0^0$, $H_1^0$, ..., $H_{d-1}^0$. Suppose that the key were encoded before and its value is $v$. Then, at least one hash set will return all '1' bits, $H_j^0 = 1$, which means that the edge may be on the path from the root to the leaf node. We continue this edge-based membership testing until the node of level $l-1$ is reached. The node of level $l-1$ may have less than $d$ edges because of $l = \lceil log_d(g) \rceil$. Finally, we perform a membership testing with $H_0^l$ at the leaf node for $y$. If the final membership testing is passed, the Bloom tree returns the value represented by the leaf node. Note that the checking may become invalid if false positives occur at those Bloom filters that are not on the edge to the correct leaf node. We explain this problem in detail.

False-positive errors may occur at any level of nodes. If the error occurs from $H_r^i$ at level $i$, we need to check the next child node associated with the $r$th edge. This type of errors may cause two problems; first, extra memory accesses are added. Second, more seriously, a classification failure may occur if the errors continue to occur until reaching any leaf node and to pass the membership test at the leaf node. Then, the Bloom tree finds more than one return value, and it cannot determine which one is the correct value. We call this a classification failure, the probability of which, $p_c$, is definitely related with the number of bits and hash functions in the Bloom tree.

Next, we consider the case when $y$ is given for search, but it has not been encoded a priori. Although $y$ is not stored, false-positive errors may occur continuously at the internal nodes to reach any leaf nodes. If the membership testing is erroneously passed in the leaf nodes, the Bloom tree will return a false value for $y$. We call this a false-positive error. The probability of false-positive errors, $p_f$, is determined by parameter settings at Bloom filter nodes.

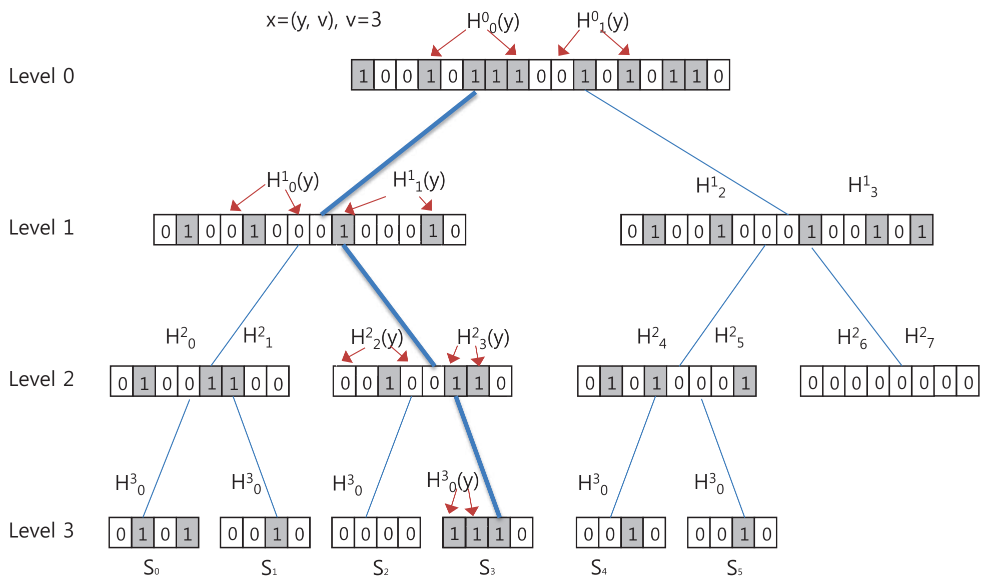

Fig. 1. A binary Bloom tree when the number of goups is 6 ($d = 2$ and $g = 6$). The solid line represents the value path for $(y, v)$.

When designing a Bloom tree, the given condition of $(m, g, p_f, p_c)$ should be satisfied. The capacity and the number of memory accesses are also considered together. We use $a_{cm}$ to denote the number of memory accesses required for checking an encoded item, a *member item*, while $a_{cn}$ is for an item that has never been encoded before, a *non-member item*.

Because many parameters are related together in designing a Bloom tree, we first need to build an analysis model. Based on the model, we guide an optimal setting of configuration parameters, which are explained in the next section.

## D. Analysis and Parameter Setting

We build an analysis model that can provide an optimal parameter setting for a Bloom tree. We first derive the upper bound for the probability of classification failures, and the probability of false positives is computed. Then, the optimal number of hash functions in each hash set is derived. The capacity and memory accesses can be estimated by the analysis model.

In a Bloom tree, a bit array is shared by multiple hash functions, and each hash function evenly distributes '1' bits over $B[m]$ according to the encoding process. Suppose that the encoding process is finished for $n$ keys. Then, the probability that a bit remains zero in $B[m]$, $p_0$, becomes as follows:

$$p_0 = (1 - \frac{1}{m})^{n \times k} \approx e^{-\frac{n \times k}{m}} \tag{3}$$

where $k = \sum_{i=0}^{l} k_i$.

*1) Probability of classification failure:* We assume that a queried key has been encoded a priori, and the checking process begins. We use $n_j^i$ to denote the $j$th node from the left at level $i$ in the Bloom tree. Let $I_j^i$ be the event that an internal node, $n_j^i$ $(0 \le i \le l-1)$, is reached during the checking process. Let $L_j^l$ be the event that a leaf node, $n_j^l$, is reached and the membership testing in that leaf node is successfully passed. If $n_j^i$ is on the path from the root to the leaf node representing the key's value, called *value path*, both events always happen. Otherwise, they may happen only when false positives have occurred at all the Bloom filters of $n_j^i$'s precedent nodes.

Suppose that $n_j^i$ is not on the value path. We define a *distance* of $n_j^i$ to be the length of the shortest path from $n_j^i$ to the value path in the Bloom tree. In Fig. 1, the distance of $n_2^3$ representing $S_2$ is one because $n_2^3$ is one hop away from $n_1^2$ that is on the value path. The distance of $n_2^2$ is two because the root node is the nearest node on the value path from it, and there are two edges between them.

A classification failure occurs only when at least one of $L_j^l$ happens. Therefore, the probability of a classification failure becomes

$$p_c = Prob(L_0^l \cup L_1^l ... L_{x-1}^l \cup L_{x+1}^l ... \cup L_{g-1}^l) \tag{4}$$

where $n_x^l$ is the leaf node representing the queried key's value.

We derive an upper bound for $p_c$ since equation 4 cannot be solved directly. Using the union inequality, the upper bound $u_c$ becomes

$$u_c = \sum_{i=0, i \neq x}^{g-1} Prob(L_i^l) \ \ge \ p_c. \tag{5}$$

We can classify $L_i^l$ into $(l-1)$ groups by the distance of $n_i^l$. Suppose that $n_i^l$ is not on the value path. Then, the distance can be from 1 to l. We use the distance as a group id. Then, the number of events in group 1 is $(d-1)$, and group 2 has $d \times (d-1)$ events. In general, group $j$ has $d \times (d-1)^{j-1}$ events. For simplicity, we assume that $log_d(g)$ is an integer. The probability that $L_i^l$ of group $j$ happens is $p_1^{\sum_{r=l-j}^{l} k_r}$ because all the precedent nodes should generate false positive errors. Since the number of events in group $j$ is $(d-1)\times d^{j-1}$, we have

$$\begin{aligned}u_c &= (d-1)\times p_1^{k_{l-1}+k_l} + (d-1)\times d^1 \times p_1^{k_{l-2}+k_{l-1}+k_l} + \\ &\quad ... + (d-1)\times d^{l-1}\times p_1^{k_0+k_1+...+k_l} \\ &= \sum_{i=0}^{l-1}(d-1)\times d^i \times p_1^{\sum_{j=l-1-i}^{l} k_j}. \end{aligned} \quad (6)$$

From the inequality of arithmetic and geometric means, $u_c$ is minimized when all the terms are equal. Therefore, we have

$$\begin{aligned}(d-1)\times p_1^{k_{l-1}+k_l} &= (d-1)\times d^1 \times p_1^{k_{l-2}+k_{l-1}+k_l} = ... \\ &= \sum_{i=0}^{l-1}(d-1)\times d^i \times p_1^{\sum_{j=l-1-i}^{l} k_j}. \end{aligned} \quad (7)$$

Solving equation 7 gives hints on the optimal number of hash functions at each level in the tree as follows:

$$k_0 = k_1 = ... = k_{l-2} = log_{p_1}(\frac{1}{d}) = log_{(1-(1-\frac{1}{m})^{n\times k})}(\frac{1}{d}). \quad (8)$$

Using equation 8, $u_c$ becomes,

$$\begin{aligned}u_c &= l\times(d-1)\times p_1^{k_{l-1}+k_l} \\ &= l\times(d-1)\times(1-e^{-\frac{n\times k}{m}})^{k+log_{(1-e^{-\frac{n\times k}{m}})}(d)^{(l-1)}} \end{aligned} \quad (9)$$

In appendix A, we prove that $u_c$ is minimized when $k = ln(2)\times\frac{m}{n}$ and $p_0 = p_1 = \frac{1}{2}$. It is interesting that this result is similar to the optimization condition of standard Bloom filters. Now, equations 8 becomes simple:

$$k_i = log_2(d), \ 0 \le i \le l-2 \quad (10)$$

According to equation 10, all $k_i$s are determined except $k_{l-1}$ and $k_l$. We set $k_{l-1}$ to $log_2(d)$ because this helps Bloom trees efficiently implemented in hardware. We further explain this issue in the next section. Finally, all $k_i$ values can be determined as follows:

$$k_i = \begin{cases} log_2(d) & 0 \le i \le l-1 \\ log_2(\frac{l\times(d-1)}{u_c\times d}) & i = l \end{cases} \quad (11)$$

In the rest of this paper, we assume that $k_i$ is set according to equation 11. When $(m, g, p_f, p_c)$ is given, we consider this condition as $(m, g, p_f, u_c)$. If we set $k_i$ according to equation 11, the original condition is also satisfied.

The capacity, $n$, can be estimated when $k_i$'s are set by equation 11. According to $k = ln(2)\times\frac{m}{n}$, we have $n = ln(2)\times\frac{m}{k}$ that satisfies $(m, g, p_f, u_c)$.

*2) Probability of false positive:* We derive the probability of false positives, $p_f$, and prove that this is smaller than the probability of classification failures.

A false positive occurs when any $L_j^l$ happens, but a queried key has not been encoded a priori. Since the key is not in the Bloom tree, only consecutive false positives can return a false value, and therefore $Prob(L_j^l) = p_1^k$. The probability that none of $L_j^l$s happen is $(1-p_1^k)^g$ where $p_1 = 1/2$. Hence

$$p_f = 1-(1-p_1^k)^g \approx g\times(\frac{1}{2})^k \quad (12)$$

where the approximation is reasonable because $(\frac{1}{2})^k << g$.

In appendix B, we prove that $p_f$ is smaller than $u_c$. In the rest of this paper, we consider $u_c$ as a major error factor in Bloom trees. The condition of $(m, g, p_f, u_c)$ is abbreviated as $(m, g, u_c)$, which means that both probabilities of classification failures and false-positives should be equal to or less than $u_c$.

*3) memory access for a member item:* We can estimate the number of memory accesses required to check the value of a queried key. We assume that the key is a member, which means that it has been encoded before the checking. Let $N_j^i$ be the event that node $n_j^i$ is reached during the checking process. If $n_j^i$ is on the value path, the node is definitely accessed. Therefore, it is clear that all the nodes on the value path require $k$ memory accesses in total. For the root node, $i = 0$, $(d-1)$ Bloom filters are also checked. Therefore, $k+(d-1)\times k_0$ hash operations and memory accesses are required when the queried key is already encoded.

We count the number of memory accesses when $n_j^i$ is not on the value path and $i \ne 0$. The node can be reached only when false-positives happen at its precedent nodes. Hence $Prob(N_j^i)$ becomes

$$Prob(N_j^i) = p_1^{\sum_{r=0}^{i-1} k_r}, \ 1 \le i \le l. \quad (13)$$

Since the number of nodes not on the value path at level $i$ is $(d-1)\times d^{i-1}$, $1 \le i \le l$, $a_{cm}$ is

$$a_{cm} = k+(d-1)\times k_0 + \sum_{i=1}^{l}(d-1)\times d^{i-1}\times k_i \times p_1^{\sum_{r=0}^{i-1} k_r} \quad (14)$$

where $k$ is added for the nodes on the value path.

*4) memory access for a non-member item:* Even when the queried key has not been encoded, some memory accesses are required to confirm its non-membership. We always need to check $d$ Bloom filters at the root node, therefore $d\times k_0$ memory accesses. Node $n_j^i$, $1 \le i \le l$, can be accessed only when false-positives happen at all of its precedent nodes. This probability is $p_1^{k_0+k_1+...+k_{i-1}}$. Therefore, the number of memory accesses to check a non-member key, $a_{cn}$, is as follows:

$$a_{cn} = d\times k_0 + \sum_{i=1}^{l} d^i \times k_i \times p_1^{\sum_{r=0}^{i-1} k_r}. \quad (15)$$

## *E. Virtual Key*

We propose a virtual key that can reduce the number of hash-function sets in a Bloom tree from $O(d^l)$ to $O(l)$. The observation is that each node in a Bloom tree knows its unique path from the root, and therefore a unique path id can be defined for every node. By appending the path id to the pure key, we can coin a unique virtual key at each node, which results in the same effect of using distinct sets of hash functions. Then, a set of $k_i$ distinct hash functions are enough for level $i$ in a Bloom tree instead of $d^{i+1}\times k_i$.

In this paper, we define a path id for a node at level $i$ to be a string of length $i$. The first digit from the right in the string means the edge number from the root node to the node on the path at level 1. The second digit represents the edge number from the node of level 1 to the node of level 2 on the path. The $j$th digit represents the edge number from the node of level $(j-1)$ to the node of level $j$ on the path. For example, the path id of $n_1^2$ is "01" in Fig. 1, and the virtual key of the node becomes "01$\|y$" where $y$ is the given key and "$\|$" is a string concatenation.

### F. Multiple Access & Load Balancing

Data structures for multiple-set membership testing have been proposed, based on multiple hash functions and Bloom filters. A hardware implementation may accelerate the processing speed because recent hardware can compute multiple hash functions and access to fast embedded memory in parallel. If the memory is split into smaller chunks, then it may be possible to access these chunks in parallel. This is the basic assumption of previous work in this literature [1], [3], [4], [14], and we make the same assumption here to design a hardware-friendly version of Bloom trees.

We propose how to redesign a Bloom tree to support parallel processing and memory access. This version is called a *multiple-access Bloom tree*. A new system parameter, $t$, is defined to be the number of hash functions and memory accesses that can be processed in parallel. The value of $t$ is dependent on hardware features. We denote the condition of multiple-set membership testing as $(m, g, u_c, t)$.

For given $t$, we choose maximum $d$ such that $t \geq d \times \lceil log_2(d) \rceil$. With this configuration, all edges from a node to its $d$ child nodes can be processed at a time because a Bloom filter representing an edge has $\lceil log_2(d) \rceil$ hash functions at most according to equation 11. The only exception is $k_l$. If $k_l$ is larger than $t$, we need to repeat the hash operations and memory accesses $\lceil \frac{k_l}{t} \rceil$ times. Otherwise, only one processing is required to cover $k_l$. This setting is conservative in following the analysis model of the previous section. Note that the hardware may not be fully utilized when $d \times \lceil log_2(d) \rceil$ is not equal to $t$. If the design goal were to maximize the processing speed, we would assign extra hash operations such that $t$ hash operations and memory accesses are always executed in parallel. In the next section, we show that this conservative configuration is enough for Bloom trees to outperform the current best method.

The total memory is partitioned into $t$ chunks of $m/t$ bits. Hash sets belonging to the same node can be processed in parallel by exclusively assigning a hash function to only one of the chunks. Fig. 2 shows an implementation example where $t$ hash functions per level in a tree are processed in parallel. This forms a 4-ary Bloom tree where two hash functions represents an edge in the tree.

A multiple-access Bloom tree can be implemented in hardware that supports multiple access and parallel processing. However, a serious problem may occur if a load balancing is not considered. For example, a special case may happen where most keys are associated with the same value. Since each hash function can access an exclusive part of the bit array, a few

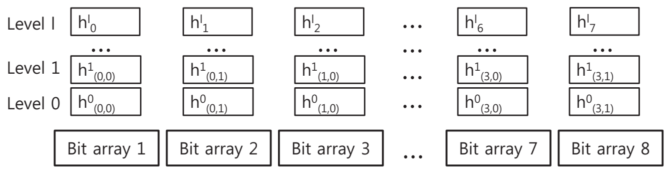

Fig. 2. Multiple hash processing and memory accessing in a Bloom tree. Eight hash functions are processed in parallel, $t = 8$, and $d$ is set to 4 according to $d \times log_2(d) = 8$. The bit array is divided into 8 non-overlapping chunks.

chunks of bits are overpopulated with '1' bits while the others have only '0' bits. This definitely degrades the performance of memory access and accuracy. We call this a skewed-(key,value) problem.

We propose a new load-balancing scheme that can solve the skewed-(key,value) problem by evenly distributing keys, or '1' bits, over the distinct bit chunks. For a given key-value pair, $(y, v)$, the new scheme replaces it with $(y, v')$ by using

$$v' = v \ XOR \ h(y) \tag{16}$$

where $h$ is a hash function and $XOR$ is a bit-wise exclusive OR operation. The value of $v'$ should be equal to one of $g$ values, and this mapping should be one-to-one. This can be easily implemented by setting the bit length of $h(y)$ equal to that of $v$. Then, the replacement is performed for both encoding and checking processes.

*1) encoding:* For $(y, v)$ to be encoded, $v'$ is computed and $(y, v')$ is encoded instead. Since only one of the leaf nodes represents $v'$, we can use the same encoding process as in a Bloom tree. Each node on the value path encodes $y$ in its Bloom filters. If a virtual key is used, the node should concatenate its path id to $y$ before encoding.

*2) checking:* When the value of $y$ is checked, the value path is searched by using the same checking process as in a Bloom tree. Since $y$ was used for encoding, we can find the value path by using $y$. This checking process will lead to the leaf node that represents $v'$ when classification failures do not happen. The correct value, $v$, can be computed by $v = v' \ XOR \ h(y)$ according to equation 16.

For a multiple-access Bloom tree, the numbers of memory accesses are derived new from equations 2, 14, and 15, as follows:

$$a_e = l + \frac{\lceil k_l \rceil}{t} \tag{17}$$

$$a_{cm} = l + \frac{\lceil k_l \rceil}{t} + \sum_{i=1}^{l} (d-1) \times d^{i-1} \times p_1^{\sum_{r=0}^{i-1} k_r} \tag{18}$$

$$a_{cn} = 1 + \sum_{i=1}^{l} d^i \times p_1^{\sum_{r=0}^{i-1} k_r}. \tag{19}$$

## V. Experiments

In this section, Bloom trees are evaluated by simulations. we confirm that Bloom trees outperform the current best method, COMBs [1], in terms of memory access as well as capacity.

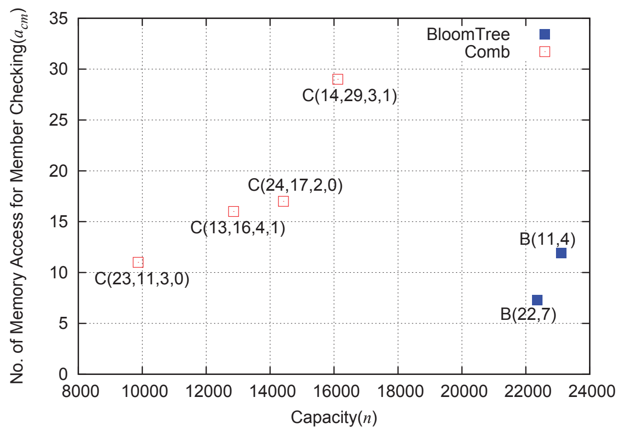

Fig. 3. Capacity vs speed (number of memory accesses for checking a member item). $(m,\ g,\ u_c) = (2^{20},\ 128,\ 10^{-6})$.

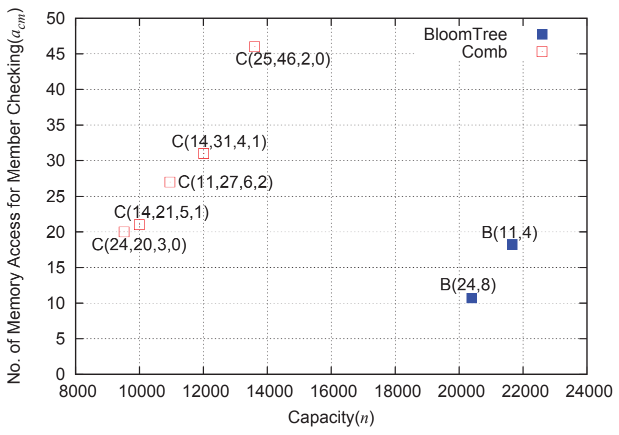

Fig. 4. Capacity vs speed (number of memory accesses for checking a member item). $(m,\ g,\ u_c) = (2^{20},\ 1,024,\ 10^{-6})$.

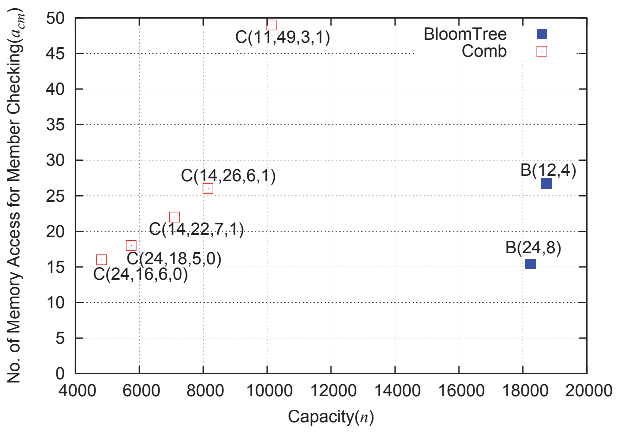

Fig. 5. Capacity vs speed (number of memory accesses for checking a member item). $(m,\ g,\ u_c) = (2^{20},\ 8,192,\ 10^{-6})$.

We use the same simulation settings as in [1]; 1 Mbits memory is assigned, $m = 2^{20}$, and both false-positive and classification failure probabilities should not be greater than $10^{-6}$. Three experimental results are given with different group numbers, $g = 128,\ 1,024$, and $8,192$.

In each experiment, we run different versions of COMBs and Bloom trees. Because $t$ memory accesses and hash operations are processed at once, therefore counted as one memory access, we should compare COMBs and Bloom trees of similar $t$ values. We first run COMBs with the same $t$ value used in [1], and then Bloom trees are built and executed. We would stress that an advantage is given to COMBs by using smaller $t$ values for Bloom trees when the same $t$ cannot be assigned to both schemes. For given parameters of $m$, $t$, and tolerable error probability, the maximum capacity, $n$, can be computed for COMBs and Bloom trees.

The experimental results are shown in Figs. 3 $\sim$ 5; the $x$-axis gives the capacity, $n$, and the $y$-axis gives the number of memory accesses for checking a member item. In the figures, different versions of COMBs and Bloom trees are plotted. We denote a COMB as $C(t, f, \theta, r)$ in which $f$ is the number of Bloom filters, $\theta$ is the weight of the code, or number of '1' bits, and $r$ is the number of digits used for error correction [1]. When $r$ is not zero, the COMB is called an ECOMB. A Bloom tree is denoted as $B(t, d)$ where $d$ is the number of edges at an internal node.

In the first experiment, $g$ is set to 128, and therefore $(m,\ g,\ u_c) = (2^{20},\ 128,\ 10^{-6})$. The result is shown in Fig. 3 that compares the number of memory accesses for checking a member item, $a_{cm}$, for different versions of COMBs and Bloom trees. In terms of $t$, we can compare two COMBs of $C(13, 16, 4, 1)$ and $C(14, 29, 3, 1)$ with the Bloom tree of $B(11, 4)$. Note that the two COMBs have larger $t$ values. Despite the disadvantage, $B(11, 4)$ outperforms the COMBs; its capacity is 23,105 while $C(13, 16, 4, 1)$ and $C(14, 29, 3, 1)$ have 12,857 and 16,115. Besides, $B(11, 4)$ requires less memory accesses for checking. When checking a member item, only 11.90 memory accesses are required by the Bloom tree while the COMBs require 16 and 29 memory accesses, respectively. For checking a non-member item, $B(11, 4)$ requires only 3.21 memory accesses while the COMBs require the same number of memory accesses as for checking a member item. Only for encoding, the COMBs are better than the Bloom tree; $C(13, 16, 4, 1)$ and $C(14, 29, 3, 1)$ requires 3 and 4 memory accesses for encoding an item, respectively, while 6 memory accesses are required for $B(11, 4)$.

We compare the next group in Fig. 3, $C(23, 11, 3, 0)$ and $C(24, 17, 2, 0)$ vs $B(22, 7)$. Comparing two COMBs, we see that $C(23, 11, 3, 0)$ is faster, but $C(24, 17, 2, 0)$ has a larger capacity. We stress that $B(22, 7)$ is 1.51 times faster than $C(23, 11, 3, 0)$ for checking a member item, and its capacity is 1.55 times larger than $C(24, 17, 2, 0)$. Since both capacity and memory access should be considered together, we stress that $B(22, 7)$ is 2.33 times faster than $C(24, 17, 2, 0)$, and its capacity is 2.27 times larger than $C(23, 11, 3, 0)$. Besides, the Bloom tree can check a non-member item 4.96 and 7.67 times faster than $C(23, 11, 3, 0)$ and $C(24, 17, 2, 0)$, respectively. The only edge of COMBs is the number of memory accesses to encode an item; they require 3 and 2 memory accesses while the Bloom tree needs 4 accesses.

We repeat the experiments with different numbers of groups,

TABLE II
CALCULATED AND EXPERIMENTAL RESULTS FOR MEMORY ACCESSES ($a_{cm}$)

| g | 128 | | 1,024 | | 8,192 | |
|---|---|---|---|---|---|---|
| B.T. | B(11,4) | B(22,7) | B(11,4) | B(24,8) | B(12,4) | B(24,8) |
| Cal. | 13.50 | 8.14 | 18.25 | 13.75 | 30.00 | 19.1 |
| Exp. | 11.90 | 7.29 | 18.25 | 10.70 | 26.72 | 15.45 |

$g = 1,024$ and $8,192$, as the same experiments from [1]. Through the experiments, we confirm that Bloom trees outperform COMBs even for different group numbers. In Fig. 5, $C(24,16,6,0)$ requires the number of memory accesses close to $B(24,8)$. However, the capacity of $B(24,8)$ is 3.17 times larger. To the best of our knowledge, Bloom trees are the best data structure for multiple-set membership testing.

Table II compares the numbers of memory accesses computed from calculations and experiments when member items are checked in Bloom trees. Equation 18 is used for the calculations. The difference between calculated and experimental values is generally small, and the calculated value is always larger than experimental one. This is because the analysis model assumes that Bloom trees are full and complete. In practice, this assumption is not always true, and therefore theoretical numbers of memory accesses are larger than experimental results. In table II, some values are very close to each other, B(11,4) of $g = 1,024$ and B(22,7) of $g = 128$ for example, because the Bloom trees are close to a full and complete tree in those cases.

## VI. CONCLUSIONS

This paper proposes a new data structure for multiple-set membership testing that is able to increase capacity and decrease memory accesses. It not only achieves space compactness, but also operates more efficiently than the existing work. Our main technical contributions include a novel data structure, Bloom tree, and the corresponding formula for setting parameters, capacity, and memory accesses, which are theoretically analyzed and experimentally verified.

## ACKNOWLEDGMENT

This work was supported by the National Research Foundation of Korea (NRF) grant funded by the Korea government (MEST) (No. 2011-0016246).

## REFERENCES

[1] F. Hao, M. Kodialam, T. Lakshman, and H. Song, “Fast Dynamic Multiple-Set Membership Testing Using Combinatorial Bloom Filters,” *IEEE/ACM Transactions on Networking*, vol. 20, no. 1, pp. 295–304, February 2012.
[2] F. Bonomi, M. Mitzenmacher, R. Panigrahy, S. Singh, and G. Varghese, “Beyond Bloom Filters: From Approximate Membership Checks to Approximate State Machines,” *Proc. of ACM SIGCOMM*, pp. 315 – 326, September 2006.
[3] F. Chang, W. Feng, and K. Li, “Approximate Caches for Packet Classification,” *Proc. of IEEE Infocom’04*, March 2004.
[4] H. Song, S. Dharmapurikar, J. Turner, and J. Lockwood, “Fast Hash Table Lookup Using Extended Bloom Filter: An Aid to Network Processing,” *Proc. of ACM SIGCOMM*, August 2005.
[5] S. Dharmapurikar, P. Krishnamurthy, T. Sproull, and J. Lockwood, “Deep Packet Inspection Using Parallel Bloom Filters,” *IEEE Micro*, pp. 52–61, January-February 2004.
[6] Y. Qiao, T. Li, and S. Chen, “One Memory Access Bloom Filters and Their Generalization,” *Proc. of IEEE INFOCOM’11*, April 2011.
[7] H. Song, F. Hao, M. Kodialam, and T.V. Lakshman, “IPv6 Lookups using Distributed and Load Balanced Bloom Filters for 100Gbps Core Router Line Cards,” *Proc. of IEEE INFOCOM’09*, pp. 2518 – 2526, 2009.
[8] B. H. Bloom, “Space/Time Trade-offs in Hash Coding with Allowable Errors,” *Communications of the ACM*, vol. 13, no. 7, pp. 422–426, 1970.
[9] A. Kumar, J. Xu, J. Wang, O. Spatschek, and L. Li, “Space-Code Bloom Filter for Efficient Per-Flow Traffic Measurement,” *Proc. of IEEE INFOCOM*, March 2004.
[10] O. Rottenstreich and I. Keslassy, “The Bloom Paradox: When not to Use a Bloom Filter?,” *Proc. of IEEE INFOCOM’12*, April 2012.
[11] O. Rottenstreich, Y. Kanizo, and I. Keslassy, “The Variable-Increment Counting Bloom Filter,” *Proc. of IEEE INFOCOM’12*, April 2012.
[12] I. Moraru and D. Andersen, “Exact Pattern Matching with Feed-Forward Bloom Filters,” *Proc. of the Workshop on Algorithm Engineering and Experiments*, January 2011.
[13] S. Cha, I. Moraru, J. Jang, J. Truelove, D. Brumley, and D. Andersen, “SplitScreen: Enabling Efficient, Distributed Malware Detection,” *Proc. of NSDI’10*, April 2010.
[14] M. Yu, A. Fabrikant, and J. Rexford, “Buffalo: Bloom filter forwarding architecture for large organizations,” *Proc. of ACM CoNEXT*, pp. 313 – 324, 2009.
[15] A. Broder and M. Mitzenmacher, “Network Applications of Bloom Filters: A Survey,” *Internet Mathematics*, vol. 1, no. 4, June 2002.
[16] B. Chazelle, J. Kilian, R. Rubinfeld, and A. Tal, “The Bloomier Filter: An Efficinet Data Structure for Static Support Lookup Tables,” *Proc. of ACM-SIAM SODA*, pp. 30–39, 2004.
[17] L. Fan, P. Cao, J. Almeida, and A. Broder, “Summary Cache: A Scalable Wide-Area Web Cache Sharing Protocol,” *IEEE/ACM Trans. on Networking*, vol. 8, no. 3, June 2000.
[18] M. Ramakrishna, E. Fu, and E. Bahcekapili, “A performance study of hashing functions for hardware applications,” *Proc. of Int. Conf. on Computing and Information*, pp. 1621–1636, 1994.
[19] S. Iyer, R. R. Kompella, and N. McKeown, “Designing packet buffers for router linecards,” *IEEE/ACM Transactions on Networking*, vol. 16, no. 3, pp. 705–717, June 2008.

## APPENDIX A: MINIMUM OF $u_c$

We derive the minimum of $u_c$. Minimizing $u_c$ is equivalent to minimizing $ln(u_c)$. From equation 9,

$$ln(u_c) = ln(l \times (d-1)) + k \times ln(1-e^{-\frac{n\times k}{m}}) + ln(d)^{(l-1)}. \quad (20)$$

We take the derivative, $\partial ln(u_c)/\partial k$, set it to 0, and solve for $k$ to find the minimum.

$$\frac{\partial ln(u_c)}{\partial k} = ln(1-e^{-\frac{n\times k}{m}}) + \frac{n\times k}{m} \times \frac{e^{-\frac{n\times k}{m}}}{(1-e^{-\frac{n\times k}{m}})}. \quad (21)$$

According to equation 21, we find the minimum of $u_c$ when $k = ln(2) \times \frac{m}{n}$. It is interesting that the derived formula is similar to that of the famous Bloom filter optimization. Note that the optimization condition of Bloom filters, $p_0 = p_1 = \frac{1}{2}$, is also valid for Bloom trees.

## APPENDIX B: $p_f < u_c$

We prove that $u_c$, the upper bound for the probability of classification failures, is larger than $p_f$, the probability of false positives. According to equations 9 and 11, $u_c$ becomes

$$\begin{aligned} u_c &= l \times \frac{d-1}{d} \times (\frac{1}{2})^{k-(l-1)\times log_2 d} \\ &= l \times \frac{d-1}{d} \times 2^l \times p_f \end{aligned} \quad (22)$$

where $p_f = g \times (\frac{1}{2})^k$ from equation 12. Since $\frac{u_c}{p_f} > 1$, $u_c$ is greater than $p_f$.