This full text paper was peer reviewed at the direction of IEEE Communications Society subject matter experts for publication in the IEEE INFOCOM 2009 proceedings.

Fast Dynamic Multiset Membership Testing Using Combinatorial Bloom Filters

Fang Hao Murali Kodialam T. V. Lakshman Haoyu Song
Bell Laboratories, Alcatel Lucent
791 Holmdel-Keyport Road, Holmdel, NJ 07733, USA
{fangh,muralik,lakshman,haoyusong}@bell-labs.com

***Abstract*—In this paper we consider the problem of designing a data structure that can perform fast multiset membership testing in deterministic time. Our primary goal is to develop a hardware implementation of the data structure which uses only embedded memory blocks. Prior efforts to solve this problem involve hashing into multiple Bloom filters. Such approach needs a priori knowledge of the number of elements in each set in order to size the Bloom filter. We use a single Bloom filter based approach and use multiple sets of hash functions to code for the set (group) id. Since a single Bloom filter is used, it does not need a priori knowledge of the distribution of the elements across the different sets. We show how to improve the performance of the data structure by using constant weight error correcting codes for coding the group id. Using error correcting codes improves the performance of these data structures especially when there are large number of sets. We also outline an efficient hardware based approach to generate the the large number of hash functions that we need for this data structure. The resulting data structure, COMB, is amenable to a variety of time-critical network applications.**

I. Introduction

Network packet processing, such as forwarding and measurement, often involves some sort of table lookups. In high speed networks, these applications require fast and deterministic lookup performance. For example, at 10Gbps line speed, up to 30 million packets can arrive at a line port per second. This essentially means all table lookups for a packet need to be finished in just 32 ns. While CAM can easily satisfy this requirement, it has long been blamed for its high power consumption and low density. It is interesting to explore the possibility of whether the same goal can be achieved by using only commodity memory devices, or even better, without using any external memory devices. This can greatly impact the system cost and power consumption. Current technologies allow the chip to embed tens of megabits of static random access memory. Although still scarce, the embedded memory exhibits extremely low access latency and high speed, which are desirable for high throughput packet processing. Efficient use of this resource is therefore a critical problem. Recent research results have demonstrated many promising applications in which the on-chip memory is efficiently used to assist high throughput packet processing, including packet buffer [1], statistical counter [2], packet classification [3], and deep packet inspection [4].

Bloom filters play an important role in supporting many of these applications. They are very efficient in terms of memory consumption. Hence fast hardware implementation is possible with embedded memory. However, a Bloom filter alone can only support membership queries, i.e, answer the question of whether an element is in the table or not. It is incapable of giving any extra information about the queried element. For many applications, such associated information needs to be retrieved for the member elements. In this paper, we attempt to address one such problem: Elements belong to different sets (or groups). In addition to query about the presence of an element, we want to know which group it belongs to.

Plenty of applications fall in this category. For example, in a Layer-2 switch, a MAC address is always associated with a port. To forward a frame, the search engine queries the MAC table with the frame's destination address. If the address is not in the table, the frame is flooded to all the output ports; otherwise, the table lookup returns an output port id. Other examples include multi-action packet classification and pattern matching. The former requires the lookup to return the action associated with the matching classifier and the latter requires the rule id of the matching pattern. Traditionally, if CAMs are not used, this information has to be stored in an additional hash table. The use of Bloom filters can only help to filter out the unnecessary hash table lookups if the element is not in the hash table at all. For the applications where the searched elements are all in the table, Bloom filters are useless. Hash tables can have poor performance due to hash collisions. Perfect hashing solutions also exist [5], but they are very computationally intensive, and thus not suitable for hardware implementation. In many high speed applications, people have to seek other alternative architectures.

This problem has been studied before. Solutions have been proposed to put extra information into Bloom filters in order to reduce or eliminate hash table lookups. Chang et al. propose multi-predicate Bloom filters where k Bloom filters are used to encode k groups [6]. One Bloom filter is assigned to each group. Each element is hashed into the

978-1-4244-3513-5/09/$25.00 ©2009 IEEE

Bloom filter corresponding to its group. This scheme can only support limited number of groups due to amount of hardware resources required for each Bloom filter. It is also sensitive to the element's group membership distribution so that it is not easy to plan for the capacity of each filter. Peacock hashing [7] and Segment hashing [8] use a similar scheme to implement a collision-free hash table.

On the other hand, binary encoded Bloom filters are proposed to reduce the number of Bloom filters used [6]. It is shown that $\log k$ Bloom filters are sufficient to represent k groups. One Bloom filter is assigned to each bit of the group id. The element is hashed into the Bloom filters for which the corresponding bits of the group id are one. The main problem of this scheme is that the total load of the Bloom filters are increased significantly since each element is hashed to half of all filters. Furthermore, elements in different groups may be hashed in different number of Bloom filters since their ids may contain different numbers of ones. Lu et al. propose to choose a fixed number of Bloom filters from a group of Bloom filters to encode the group id [9]. This scheme can significantly reduce the total load on Bloom filters compared to the binary-encoded method, but it requires to use more Bloom filters. Similar to other approaches, load balancing is also a problem.

Bloomier filter [10] can be seen as another group-encoded data structure. It achieves load balancing but through a very inefficient way. Each information bit is assigned a pair of Bloom filters: 0-Bloom filter and 1-Bloom filter. For any element, if the information bit needs to be set, then the element is hashed into the 1-Bloom filter; otherwise, the element is explicitly hashed into the 0-Bloom filter. Therefore, each element is hashed into exactly 50% of the Bloom filters. The better group differentiation is gained at the cost of heavier load.

In general, all existing solutions face two hurdles. First, group sizes may vary a lot. Imbalance in group sizes makes the Bloom filter very difficult to engineer. Second and more important, the designers are forced to make a tough decision between memory efficiency and system complexity. At one extreme, the resulting data structure is no longer memory-efficient, defying the purpose of applying Bloom filters; at the other extreme, the architecture is over complicated and requires too much distributed resources. Both cases create serious scalability issues. In this paper, we propose a novel Bloom filter-based architecture that supports multi-set membership testing. Unlike all previous solutions, it does not require a priori knowledge of group size information. We show how to improve the performance of the data structure by using constant weight error correcting codes for coding the group id. Using error correcting codes improves the performance of these data structures especially when there are large number of sets. We also outline a efficient hardware based approach to generate the large number of hash functions that we need for this data structure. The resulting data structure, COMB, is amenable to a variety of time-critical network applications. We explore the full range of tradeoffs between the memory size, system complexity, and lookup performance.

II. Model

The problem we set to solve can be formally modeled as follows: Let S denote a set comprising of n elements. Each element of the set belongs to one of γ groups. Let G denote the set of groups. Given some $x \in S$ we use $g(x) \in G$ to represent the group that contains x. We refer to $g(x)$ as the *group id* of x. The elements of the set are not specified a priori but are given one at a time. Our objective is to maintain the elements along with the associated groups in a dynamic data structure that permits fast *classification* of x: Given any x we have to determine $g(x)$. If $x \notin S$ then we want to return $g(x) = \perp$. Therefore the valid range for the output is $G \cup \{\perp\}$. Given a group $k \in G$ we use n_k to denote the number of elements $x \in S$ such that $g(x) = k$. The values of n_j are *not known* a priori. We want to develop a fast and efficient data structure to be able to insert and query for the values of the given elements. We develop a probabilistic data structure called *COMbinatorial Bloom filter (COMB)* that permits fast insertion and querying of the group ids for the elements in S. Since the data structure is probabilistic, we now give the types of errors that can occur. One can also view this data structure as maintaining the elements of multiple sets where each set corresponds to the set of elements that map to a given group id.

A. Types of Errors

The types of error that can occur with a COMB can be classified into three categories.

- **False Positive:** Assume that we are querying for the group id for some $x \notin S$. Instead of returning $\perp$, the data structure returns some $k \in G$. This is similar to a false positive probability in a Bloom filter. We assume that the false positive probability of the COMB is upper bounded by α.
- **Misclassification:** In this case the data structure outputs the wrong group id for some $x \in S$. In other words, given some $x \in S$ whose group id is $g(x) = k \in G$. The COMB instead outputs $g(x) = j \neq k$. A particular case of misclassification is when given some $x \in S$, we declare that $g(x) = \perp$. This is a false negative. There are several applications where false negatives and misclassification are not desirable or acceptable. Throughout this paper, we assume that misclassification is not permitted.
- **Classification Failure:** If misclassification is not permitted, one approach is to declare that we do not know the group id for some $x \in S$. Unlike the misclassification,

where we give a wrong answer, in this case we declare out inability to give the correct value of $g(x)$. The COMB should be designed so that the probability of classification failure is less than β. Note that in some cases, the classification failure can be resolved through an off-chip hash table. It is simply translated into multiple hash table lookups.

We now outline the different performance metrics that will be used to evaluate a COMB.

B. Performance Metrics

Our primary goal is to implement a COMB in hardware. Therefore, the performance metrics are tailored to measure the performance of a hardware implementation of the COMB. There are two main metrics that can be used to compare different COMB are memory and processing overhead.

- **Capacity of a COMB:** There are two different capacities associated with a COMB. Given a fixed amount of memory, the number of elements (cardinality of the set S) that can be accommodated in the COMB while satisfying all the error bounds will be referred to as the *capacity* of the COMB. We use κ to denote the capacity of a COMB. The second is the number of group ids that can be accommodated in a COMB. This will be referred to as the *group capacity* of the COMB. We use ρ to refer to the group capacity of a COMB.
- **Number of memory accesses for insertion:** This refers to the number of times we have to hash into memory for each element that we insert into the COMB. A naive way of measuring the number of memory accesses is to measure the total number of accesses for each arrival. For example, in the case of a standard Bloom filter with h hash functions, the number of memory accesses for each arrival is h. This number can be misleading if many of these accesses can be done in parallel. If the memory is split into smaller chunks then it may be possible to access these chunks in parallel. If there are c chunks of memory and a_i denote the number of accesses that we have to make to chunk i, then $\max_{1 \leq i \leq c} a_i$ denotes the number of memory accesses that we have to perform. For example in a case of a standard Bloom filter with h hashes assume that we split the total amount of memory into h parts. When there is an arrival there is one hash into each of the h memory partitions. These accesses can be done in parallel. Therefore, in the case of a standard Bloom filter the number of memory accesses for insertion is one.
- **Number of memory accesses for classification:** As in the case of insertion, we measure the number of memory accesses for testing for the group id. In a regular Bloom filter with h hashes, where all the hashes access the same chunk of memory, the number of memory accesses for testing is h. However, if the implementation splits the memory into h chunks, then these h memory accesses can be done in parallel. In this case testing can be done with a single memory access.

Different schemes perform well on one or more of these metrics while performing poorly on others. We pick a scheme that is matched to the application. For example, in the case of traceback applications where testing is done relatively infrequently, it is much more important to keep the number of memory accesses low for insertion. For packet classification or deep packet inspection where insertions are relatively infrequent but testing is done frequently the number of memory accesses for testing has to be kept low.

C. Hash Functions for a COMB

A COMB uses many more hash function than a standard Bloom filter. Therefore we have developed a fast hardware based hashing algorithm that can efficiently generate a few hundred hashes. As in the case of any hash function, this hash function is a pseudo-random functions. In Section VII-A, outline an algorithm for efficiently generating several hundred hash functions which behave like independent random hash functions. In spite of the fact that a large number of hash functions are needed, we show that the computation of the hash functions is not a bottleneck in a COMB.

III. Approaches using Multiple Filters

There are some straightforward approaches to the multiset membership testing problem. In this section, we discuss two main categories of existing approaches in details and point out the problems with these approaches.

A. Separate Bloom Filter for Each Group

One approach to address this problem is to have a separate Bloom filter for each group id [6]. When a new element $x \in S$ is to be added to the Bloom filter, then we hash the element (multiple times) into the Bloom Filter for $g(x)$. There are two drawbacks with this approach. The total amount of available memory has to be partitioned between the filters for the different groups. In order to do this efficiently, we have to know the number of elements that will hash into each filter. The number of elements that will hash into the filter representing group $k \in G$ will be n_k. Since this value is not known in advance, there is no good way to partition the available memory. An alternate approach is to allocate memory in an incremental fashion but this is not easy to implement in hardware. Another drawback of this approach is the fact that when we are checking, we have to check against all $t = |G|$ Bloom filters. This can be very expensive if the number of groups is large.

B. Coded Bloom Filter

One way to reduce the overhead due to checking is to have a fixed number f of Bloom filters and code the group id into these f Bloom filters [6], [9]. Each set index is coded into a f bit binary vector and an element of the group is hashed into each of the filters where the code has a one. In order to check the set that includes a given element x, we hash x into each of the f filters and determine the filters where we get a one. If there are no false positives, this gives the code of the group that contains x. Note that the case where $f = k$ represents the approach where there is one Bloom filter for each group. Since $f < t$ this method can reduce the number of memory accesses for testing. However it still has the following problems:

- **Sizing the Filters:** The number of elements that will be hashed into a given filter is a function of the sum of the number of elements in the group id that map to that filter. Since this is not known in advance, it is not possible to allocate the given memory between the different filters.
- **Decoding with False Positives:** If an arbitrary coding is used to map group ids into the filters, then there will be misclassification of group id due to false positives in one or more filters.

In order to avoid partitioning memory between multiple filters with unknown loads, we design a mechanism with a single filter and code over different hash sets to identify the group ids. We refer to this data structure as a combinatorial Bloom filter.

IV. Combinatorial Bloom Filter

We now outline the operation of a COMB for group id determination. We first describe a simple COMB. In Section V we describe a Partitioned Combinatorial Bloom Filter (PCOMB) that gives some additional degrees of freedom for making the tradeoffs between amount of memory and number of memory accesses. In COMB, each group id mapped to a non-zero f bit binary vector which is the group's code. We use $C(g)$ to be the code for group $g \in G$. The code for $\perp$ is the f bit zero vector.

Corresponding to each bit in the code is a set of h hash functions. Each set of h hash functions will be referred to as *hash set.* Since there are f hash sets, each with h hash functions, a COMB requires fh hash functions. We now describe the insertion and checking operations.

- **Insertion:** Assume that we have to insert $x \in S$ with group id $g(x)$ into the COMB. We first compute $C(g(x))$. We hash x using each of the hash sets where $C(g(x))$ is one and set the bits in the COMB. Since each of the hash sets has h hash functions, we hash x a total of $hw_{C(g(x))}$ times into the Bloom filter, where w_v is defined as the *weight* of a binary vector v, i.e., the number of ones in the binary vector. We now illustrate this with an example. Assume that we have to insert x whose group id is 7 into the COMB. Assume that we use a $f = 5$. The code for each group is a 5 bit binary vector. Let $C(7) = (0, 1, 1, 0, 1)$ denote the code for group 7. Let the number of hash functions in each hash set be $h = 10$. To insert x, we hash using hash sets $2, 3$, and 5. Since each hash set has 10 hash functions, we hash 30 times into the COMB and set all these bits to one. Note that there are a total of 30 memory accesses for inserting x into the COMB.
- **Checking:** In order to determine the group id of a given x, we hash x using all f hash sets ($f \times h$ hashes in all). We initialize a f bit binary vector v to zero. Bit i in vector v is set to one if all h hash functions in hash set i results in a one. The group id of x is then $C^{-1}(v)$. There are fh memory accesses for checking. In our example, if we have to determine the group id for some given x, then we hash into the COMB using all 5 hash sets (50 hashes in all). Let $v = (0, 0, 1, 1, 0)$ denote the vector of hash results. In this case the memory location of all 10 hashes from hash set 3 and hash set 4 were one. We now have to determine the group id whose code is v and output that as the code id of x. Checking for group id therefore has 50 memory accesses.

The insertion and checking procedure outlined above have a couple of problems: First let us compute the number of hashes that are made into the COMB when inserting all $x \in S$. This is needed in order to size the COMB for achieving the desired accuracy. Let τ represent the total number of hashes into the COMB.

$$\begin{aligned} \tau &= h \sum_{x \in S} w_{C(g(x))} \\ &= h \sum_{k \in G} n_k w_k \end{aligned}$$

where n_k is the number of elements that have group id k and w_k is the weight of the code for group id k. Since n_k are not known a priori, we can only upper bound this expression with the maximum weight code and that can lead to a significant decrease in the capacity of the COMB. The second problem with the scheme outlined above, is the fact that there can be misclassification of the group id if there are false positives for one or more of the hash sets.

A. Fixed Weight Group Codes

One way to fix both the problems is the use a fixed weight code for all groups. The code for all the groups have the same number of ones, i.e. $w_{C(k)} = \theta$ for all $k \in G$. In this case the number of hashes into the COMB is

$$\begin{aligned} t &= h\sum_{k\in G} n_k w_k \\ &= h\theta \sum_{k\in G} n_k = h\theta n. \end{aligned}$$

Fixed weight codes also eliminate the problem of misclassification. There will be a false positive if some $x \notin S$ ends up with exactly θ bit level false positives. However note that if $x \in S$, then there will certainly be θ ones when we check for membership. If there are additional ones due to false positives, then we immediately declare our inability to determine the group id for x. By designing the COMB appropriately, we will ensure that the accuracy requirements are met. We call a COMB with f hash sets and a group id code with weight θ as an (f,θ)-COMB. Note that the number of memory accesses for insertion is $h\theta$. The checking procedure for a fixed weight group code is the following:

Checking: In order to determine the group id of a given x, we hash x using all f hash sets (fh hashes in all). We initialize a f bit binary vector v to zero. Bit i in vector v is set to one if all h hash functions in hash set i results in a one. If $w(v) < \theta$ then we declare $g(x) = \perp$. If $w(v) > \theta$, then we declare a classification failure. If $w(y) = \theta$, then the group id of x is then $C^{-1}(v)$. There are fh memory accesses for checking.

B. Computing Error Probabilities

The false positive probability as well as the classification failure probabilities are increasing functions of the probability that we get a false positive probability with any given hash set. Before computing the false positive and classification failure probabilities, we first determine the optimal number of hashes h that minimizes the probability that we get a false positive with any hash set. Let p denote the probability that a particular hash set returns a one in error. Let m denote the number of bits of memory that we have for the Bloom filter. The false positive probability p for any given hash set is given by

$$\begin{aligned} p &= \left(1 - \left(1 - \frac{1}{m}\right)^{\theta n h}\right)^h \\ &\approx \left(1 - e^{-\frac{\theta n h}{m}}\right)^h \end{aligned}$$

To find the value of h that minimizes p we differentiate the above expression with respect to h and set to zero to get

$$h = \frac{m}{n\theta}\log 2$$

and the minimum value of p is

$$p = (0.6185)^{\frac{m}{n\theta}}.$$

This analysis is similar to the false positive analysis of a standard Bloom filter with $n\theta$ elements. In the rest of the paper, we assume that each hash set has $h = \frac{m}{n\theta}\log 2$ hash functions.

C. False Positive Probability

There is a false positive if exactly θ hash sets have a false positive. Each hash set has a false positive probability of p. Therefore, the probability that exactly θ out of f hash sets result in a one is given by

$$\begin{aligned} \text{False Positive Probability} &= \binom{f}{\theta} p^\theta (1-p)^{f-\theta} \\ &\leq \binom{f}{\theta} p^\theta \\ &\leq f^\theta p^\theta = (fp)^\theta \end{aligned}$$

We typically use a value of $\theta > 1$, therefore the false positive probability is typically quite small.

D. Classification Failure Probability

Classification failure occurs if even one of the $f - \theta$ hash sets returns a one when we are checking for the group id of some $x \in S$.

$$\begin{aligned} \Pr[\text{Classification Failure}] &= 1 - (1-p)^{f-\theta} \\ &\leq (f-\theta)p \leq fp \end{aligned}$$

Note that the classification failure probability is usually far greater than the false positive probability. If the values of the target false positive probability α and the classification failure β are similar in magnitude, then the classification failure probability will determine the memory size.

E. Capacity of an (f,θ)-COMB

We now compute the capacity of a (f,θ)-COMB. Since we use a constant weight code of θ out of a possible f bits to represent the group id, the group capacity of the COMB represented by $\rho(f,\theta)$ is given by

$$\rho(f,\theta) = \binom{f}{\theta}.$$

The capacity of an (f,θ)-COMB with m bits of memory is the number of elements that it can accommodate while satisfying the error bounds. From the analysis of the error probabilities, we get $(fp)^\theta < \alpha$ and $fp < \beta$. Substituting $p = (0.6185)^{\frac{m}{n\theta}}$ and solving for n, we get the capacity $\kappa(f,\theta)$ of a (f,θ)-COMB.

$$\kappa(f,\theta) = 0.48\frac{m}{\theta \log\frac{1}{\lambda}} \text{ where } \lambda = \min\{\ \frac{\alpha^{\frac{1}{\theta}}}{f}, \frac{\beta}{f}\ \}.$$

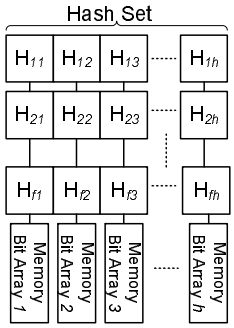

Fig. 1. Load-Balanced Parallel Memory Access

If the value of $\theta > 1$, as is typically the case to keep the number of memory accesses for checking low, the second term in minimization is lower and hence the classification failure probability dominates. In addition note that the number of hashes $h = 1.44 \log\left(\frac{\theta}{\beta}\right)$, the number of memory accesses for insertion is θh and the number of memory accesses for checking is fh.

F. Parallelizing Memory Accesses

In a Bloom filter with h hashes, the total memory of m can be partitioned into chunks of m/h each. Each hash function has access to its own memory chunk. Therefore access can be parallelized and both insertion and deletion can be done in time proportional to one memory access. In our case, the partitioning has to be done more carefully since we want to maintain the load on each chunk of memory to be the same. In a (f, θ) COMB there are f sets of hashes, each having h hash functions. Assume that we partition the total memory of m into h chunks of m/h each. Assume that the first hash function of each hash set has access to the first chunk. The second hash function in each hash set accesses the second hash function and so on. The architectue is illustrated in Figure 1. During insertion, we use θ hash sets. Therefore, we hash θ times in parallel into each of the h memory chunks. The number of memory accesses per chunk is θ. Therefore the amount of memory accesses for insertion is θ and not θh when we use a single chunk of memory. Similarly during checking, we hash f times in parallel into each chunk of memory and therefore the memory access overhead for checking is f. Parallelizing reduces insertion as well as checking memory overhead by a factor of f. When k-port memory is available, there will be another k times reduction on the memory access overhead.

V. Partitioned COMB

There are variants of a COMB that can tradeoff memory and lookup complexity. We outline one such variant called a Partitioned Combinatorial Bloom Filter (PCOMB). A PCOMB is characterized by f hash functions that are partitioned into p subsets. Let f_j denotes the number of hash functions in partition j, then we have $\sum_j f_j = f$. Associated with partition j is a fixed length code with weight θ_j. Therefore, a PCOMB is represented as $\{(f_1, \theta_1), (f_2, \theta_2), \ldots, (f_p, \theta_p)\}$. Assume that memory is partitioned into p blocks where block j of memory is allocated to partition j. Let m_j be the amount of memory allocated to partition j. We set

$$m_j = \frac{\theta_j}{\sum_{i=1}^{p} \theta_i} m.$$

Therefore, memory is allocated proportional to the code weight of the partition. Each memory partition is further partitioned into h equal chunks, one for each hash function as in the COMB case. Therefore, all the j^{th} hash functions partition i hash into one section of memory for $j = 1, 2, \ldots, h$ and $i = 1, 2, \ldots, p$. The group capacity of a PCOMB is

$$\rho = \prod_{i=1}^{p} \binom{f_i}{\theta_i}.$$

Since memory accesses are parallelized at two levels, memory access overhead for insertion is $\max_{1 \le i \le p} \theta_i$ and the overhead for testing is $\max_{1 \le i \le p} f_i$. Also note that the capacity

$$\kappa = 0.48 \min_{1 \le i \le p} \frac{m_i}{\theta_i \log \frac{1}{\lambda}}.$$

Since the value of m_i proportional to θ_i, the capacity

$$\kappa = 0.48 \frac{m}{\theta \log \frac{1}{\lambda}}.$$

Note however that partitioning typically leads to a larger value $\sum_i \theta_i$ that represents the load on the COMB. This increases the load on the memory leads to a decrease in insertion and checking overhead.

A. Numerical Example

In this section we give a numerical example to illustrate the design choices that we have to make while constructing a COMB. Consider a set S comprising of elements that can belong to one of 1024 groups. Assume that we have 1 megabits of memory for the COMB and both the false positive probability and classification failure probabilities are 10^{-6}. We can use a $(1024, 1)$-COMB but checking for membership will require 1024 memory accesses which is impractical but the capacity of this COMB is 23160. We can use a $(46, 2)$-COMB and insertions will involve 2 memory accesses and checking will be 46 memory accesses. The capacity of a $(46, 2)$-COMB is 13650 since each arrival hashes twice into the COMB. We can also use a $(20, 3)$-COMB which involves only 20 memory accesses although its capacity is only 9629. Another alternative is to use a partitioned $\{(9, 2), (9, 2)\}$-PCOMB since

$$\binom{9}{2}\binom{9}{2} \ge 1024.$$

This will only have 9 memory accesses and its capacity is 7790. Thus there is a natural trade-off between memory access and capacity. We next explore if we can correct for errors and improve the capacity of COMBs. Instead of using arbitrary constant weight codes to code the group id, we use constant weight error correcting codes. This gives additional flexibility in trading off memory access and capacity.

VI. Error Corrected COMB

In Section IV-E, we determined that the performance measure that determines the capacity of a COMB is classification failure. Classification failure occurs when we are determining the group id of some $x \in S$, and we get a false positive from some hash set j where component j of $C(g(x))$ is zero. One way to improve the performance of a COMB is to code the group id using an error correcting code. Instead of $C(g())$ being an arbitrary f bit binary code, we will use a f bit error correction code. Depending on the number of errors corrected, we can tolerate one or more false positives. The trade-off is that using an error correcting code reduces the number of valid codewords and hence the number groups that can be handled by the COMB. The *weight spectrum* of an error correcting code gives the code weight distribution of an error correcting code. Consider the $(7, 4)$ Hamming code. This code can correct one error. It has 4 data bits and 3 error correcting bits. It has $2^4 - 1 = 15$ non-zero code words. It has 7 code words with weight 3, 7 code words with weight 4 and 1 code word with weight 7. If we want to maintain the property that there are no misclassification, then we can only use constant weight codes. If we have less than 7 possible group id, then we can use code words of weight 3 in a $(7, 4)$ Hamming code to represent the 7 group ids. If there is one error then the code corrects the error. If there is more than one error, then we will see 5 ones when we check for the group id for some $x \in S$ and we declare a classification failure. We use (f, θ, t)-ECOMB to denote an $(f\theta)$-COMB that corrects t errors.

A. Effect of Error Correction

The effect of error correction is to reduce the probability of classification failure. Consider the when we correct one error. Let p_m denote the false positive probability for any given hash set. The probability that we have an classification failure is the probability that we have greater than 1 false positive. This is given by

$$\begin{aligned}\Pr[>1 \text{ false pos.}] &= 1-(1-p)^{f-\theta}-fp(1-p)^{f-\theta-1}\\ &\leq 1-(1-p)^{f}-fp(1-p)^{f-\theta}\\ &\leq 1-(1-p(f-\theta))(1+p(f-\theta)f)\\ &= (p(f-\theta))^2 \leq (pf)^2.\end{aligned}$$

This reduction in classification failure results in an improvement results in an improvement in capacity. Assume that we have a standard (f, θ)-COMB. If the classification failure is the bottleneck, then the capacity of the $(f, \theta, 1)$-ECOMB is

$$\kappa(f,\theta,1) = 0.48\frac{m}{\theta \log \frac{1}{\lambda}} \text{ where } \lambda = \min\left\{\frac{\alpha^{\frac{1}{\theta}}}{f}, \frac{\beta^{\frac{1}{2}}}{f}\right\}.$$

Note the factor of $\beta^{\frac{1}{2}}$ instead of β for a standard COMB. This leads to a doubling of the capacity compared to a COMB (with no error correction) with the same parameters. Whether error correction helps in a particular case depends on the existence of a light constant weight code with enough code words to code all the groups. We address this issue as well as the effect of correcting multiple errors now.

B. Constant Weight t Error Correcting Codes

We now address the issue of whether it helps to correct multiple errors. In order to still maintain the property of not having misclassification, the error correcting code has to be a constant weight code. Constant weight codes have been studied extensively in coding theory literature and there is a large body of work in this area [12]. Let $\pi(f, \theta, t)$ denote the maximum number of code words in fixed length binary code of length f, weight θ that can correct t errors. The value of $\pi()$ is known for several combinations of f, θ and t [12]. We have to ensure that $\pi(f, \theta, t)$ is greater than the number of groups. If the code corrects for t errors then the classification failure probability is

$$\begin{aligned}\Pr[>t \text{ false positive}] &= \sum_{j=t+1}^{f-\theta}\binom{f-\theta}{j}p^j(1-p)^{f-\theta-j}\\ &\leq \sum_{j=t+1}^{f-\theta}\binom{f-\theta}{j}p^j\\ &\leq \sum_{j=t+1}^{f-\theta}\frac{(f-\theta)^j p^j}{j!}\\ &= \frac{(fp)^{t+1}}{(t+1)!}+o\left((fp)^{t+1}\right)\end{aligned}$$

Therefore the leading term is

$$\frac{(fp)^{t+1}}{(t+1)!}$$

Note that for single error correction, there is an additional factor of 2 since the analysis is tighter. The capacity of an (f, θ, t)-ECOMB is

$$\kappa(f,\theta,t) = 0.48\frac{m}{\theta \log \frac{1}{\lambda}} \text{ where } \lambda = \min\left\{\frac{\alpha^{\frac{1}{\theta}}}{f}, \frac{[(t+1)!\beta]^{\frac{1}{t+1}}}{f}\right\}$$

The number of memory accesses for insertion is f and the number of memory accesses for checking is θ. We now give

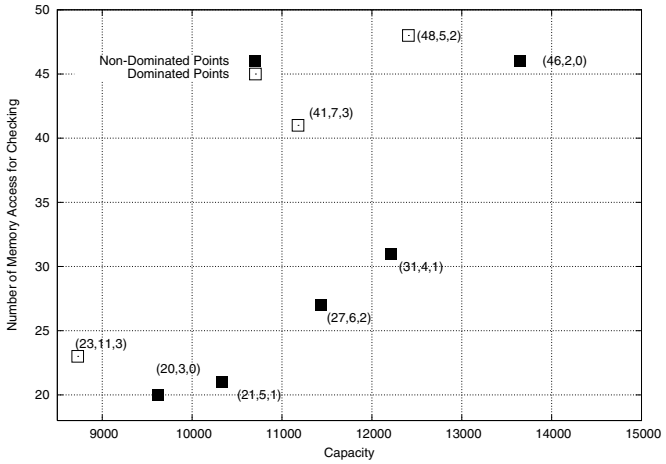

Fig. 2. Number of Groups 128: Capacity versus Checking Memory Accesses

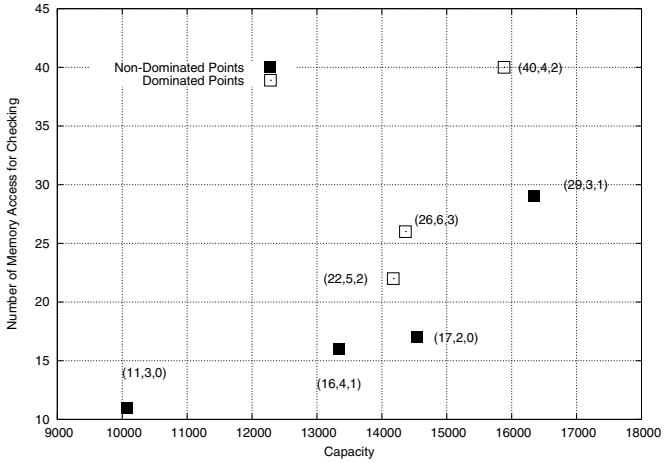

Fig. 3. Number of Groups 1024: Capacity versus Checking Memory Accesses

examples for the number of code words $\pi(f, \theta, t)$ for some cases that are useful.

C. Numerical Examples

We now give three examples to show the performance of COMB and ECOMB. We consider three group id sizes, $128, 1024$ and 8192. In each case we use different COMB and ECOMB that can accommodate the group id space. For constant weight error correcting codes, we use [12] to find codes of the appropriate size. The results are shown in Figures 2, 3 and 4. We assume that memory is 1 megabits and both false positive probability as well as classification failure probabilities are 10^{-6}. The x-axis gives the capacity of the COMB or ECOMB and the y axis gives the number of memory accesses for checking. Note that in general the capacity increases with checking memory access. The code is written next to the data point. It is of the form (f, θ, t) where

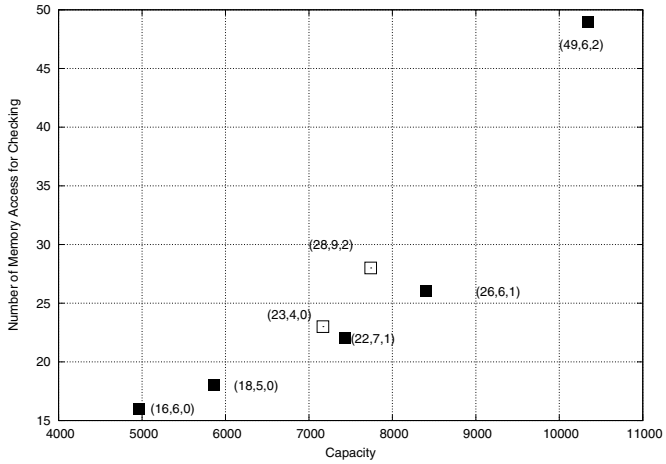

Fig. 4. Number of Groups 8192: Capacity versus Checking Memory Accesses

f is the number of hash sets, θ is the number of memory accesses for insertion and t is the number of errors corrected by the code. We say that a code is *dominated* if there is another code whose capacity is greater and the amount of memory accesses is smaller. Dominated codes are shown with a hollow square and the non-dominated codes are shown in solid. As the group size increases, error correction provides a significant increase in capacity. See for example the $(49, 6, 2)$ code in Figure 4. This is due to the fact that when the code weight is high for non-error correcting code, the overhead for error correction is low.

Summary: Error Correction offers additional flexibility and allows better tradeoff between number of memory accesses and capacity. What parameter to select depends on requirement of the application. For example, for group capacity of 1024 (Figure 3), to maximize the filter capacity, one can choose (29, 3, 1), i.e., using 29 hash sets, 3 memory accesses for insertion, and 1 error correction. If memory check speed is the main concern, one should choose (11, 3, 0), or even use PCOMB to further reduce number of memory accesses.

VII. ECOMB Implementation

In this section we briefly discuss the hardware implementation of ECOMB.

A. Efficient Hash Value Generator

In addition to block memories, our design also needs to implement many independent hash functions. The brute-force implementation may take enormous hardware resource. We remedy this problem by using an area efficient scheme that can produces n "independent" hash values using just $O(\log n)$ independent seed hash functions.

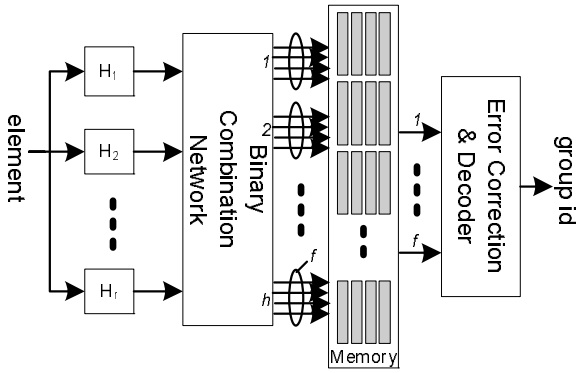

Fig. 5. ECOMB Hardware Implementation

The scheme is simple. Each new hash value is acquired from XORing a subset of the hash values produced by the seed hash functions. These synthetic hash values exhibit excellent random and independency. The experiments we have conducted prove that, when all the other configurations are identical, the Bloom filters implemented with this hashing scheme provide the same false positive rate as the Bloom filters using independent hash functions.

We use the H_3 hash functions [13] as seeds because they are simple and fast. Since our scheme uses just simple logic operations, the hash value can be generated in just one clock cycle. When generating 256 16-bit hash values, our scheme needs only 13K registers, accounting for only 5% of the resource consumed by the scheme that implements 256 H_3 hash functions.

B. Error Correction and Decoding

Binary code error correction is a well studied problem. The parallel error correction cicuit using Linear Feedback Shift Registers (LFSRs) can correct errors in a block in just one clock cycle. We can simply use the constant-weight code as the group id. In case we have to map the code to some arbitrary group id, we can use a CAM to achieve this easily.

C. Overall Architecture

Figure 5 shows the block diagram of the ECOMB hardware implementation. The pipeline architecture enables a throughput of $\frac{f}{ks}$ clock cycles per element lookup, where f is the number of hash groups, k is the port number of memory block, and s is the memory speedup factor.

VIII. Conclusions

We have designed a data structure for fast multi-set membership testing. We have shown how coding using multiple hash sets as opposed to multiple Bloom filters leads to data distribution oblivious structure. We demonstrated the use of constant weight error correcting code to improve the performance of the data structure especially when there are a large number of groups. The data structure is targeted for hardware implementation which uses purely embedded memory blocks. It is amenable to a variety of time-critical network applications.

References

[1] S. Iyer, R. R. Kompella, and N. McKeown, "Designing packet buffers for router linecards," *IEEE/ACM Transactions on Networking*, vol. 16, no. 3, 2008.

[2] Q. Zhao, J. Xu, and Z. Liu, "Design of a novel statistics counter architecture with optimal space and time efficiency," in *SIGMETRICS*, 2006.

[3] S. Dharmapurikar, H. Song, J. S. Turner, and J. W. Lockwood, "Fast packet classification using Bloom filters," in *ANCS*, 2006.

[4] S. Dharmapurikar and J. W. Lockwood, "Fast and scalable pattern matching for network intrusion detection systems," *IEEE Journal on Selected Areas in Communications*, vol. 24, no. 10, 2006.

[5] M. Dietzfelbinger, A. Karlin, K. Mehlhorn, F. Heide, H. Rohnert, and R. Tarjan, "Dynamic perfect hashing: Upper and lower bounds," in *SIAM J. Computing*, 1990.

[6] F. Chang, K. Li, and W. chang Feng, "Approximate caches for packet classification," in *IEEE INFOCOM*, 2004.

[7] S. Kumar, J. Turner, and P. Crowley, "Peacock hashing: Deterministic and updatable hashing for high performance networking," in *IEEE INFOCOM*, 2008.

[8] S. Kumar and P. Crowley, "Segmented hash: An efficient hash table implementation for high performance networking subsystems," in *ANCS*, 2005.

[9] Y. Lu, B. Prabhakar, and F. Bonomi, "Bloom filters: Design innovations and novel applications," *Allerton Conference*, 2005.

[10] B. Chazelle, J. Kilian, R. Rubinfeld, and A. Tal, "The Bloomier filter: an efficient data structure for static support lookup tables," in *ACM-SIAM Symposium on Discrete Algorithms (SODA)*, 2004.

[11] H. Yu and R. Mahapatra, "A memory-efficient hashing by multi-predicate Bloom filters for packet classification," in *IEEE INFOCOM*, 2008.

[12] E. M. Rains and N. J. A. Sloane, "Table of constant weight binary codes," at http://www.research.att.com/ njas/codes/Andw/.

[13] M. Ramakrishna, E. Fu, and E. Bahcekapili, "A performance study of hashing functions for hardware applications," in *Proc. 6th Int'l Conf. Computing and Information*, 1994.