When Bloom Filters Are No Longer Compact: Multi-Set Membership Lookup for Network Applications

Yan Qiao, Shigang Chen, *Fellow, IEEE*, Zhen Mo, and Myungkeun Yoon

***Abstract*— Many important network functions require online membership lookup against a large set of addresses, flow labels, signatures, and so on. This paper studies a more difficult, yet less investigated problem, called multi-set membership lookup, which involves multiple (sometimes in hundreds or even thousands) sets. The lookup determines not only whether an element is a member of the sets but also which set it belongs to. To facilitate the implementation of multi-set membership lookup in on-die memory of a network processor for line-speed packet inspection, the existing work uses the variants of Bloom filters to encode set IDs. However, through a thorough analysis of the mechanism and the performance of the prior art, much to our surprise, we find that Bloom filters—which were originally designed for encoding binary membership information—are actually not efficient for encoding set IDs. This paper takes a different solution path by separating membership encoding and set ID storage in two data structures, called index filter and set-id table, respectively. With a new ID placement strategy called uneven candidate-entry distribution and a two-level design of an index filter, we demonstrate through analysis and simulation that when compared with the best existing work, our new approach is able to achieve significant memory saving under the same lookup accuracy requirement, or achieve significantly better lookup accuracy under the same memory constraint.**

***Index Terms*— Computational efficiency, approximation algorithms, compression algorithms.**

I. Introduction

MANY important network functions require online *membership lookup* against a large set of addresses, flow labels, signatures, etc. For instance, some routing-table lookup algorithms must determine whether a given destination address prefix belongs to a set of prefixes extracted from the routing table [1]. In another example, a router (or gateway, firewall) can be configured to collect information of per-user activity for a set of user addresses [2]. For each arrival packet, the router performs membership lookup to see if the source address of the packet belongs to the user set. For traffic measurement, a router may be instructed to identify the set of current flows. This requires the router to collect flow labels [3], e.g., address/port tuples that identify TCP flows. Since each flow label should be collected only once, when a new packet arrives, the router must check whether the flow label extracted from the packet belongs to the set that has already been collected before. In the final example, to support the CBAC (context-based access control) function in Cisco routers, when a router receives a packet, it may want to determine whether the addresses/ports in the packet has a matching entry in the CBAC table before performing the CBAC lookup.

The above online membership lookup problems have been extensively studied. Many influential solutions [1], [3]–[6] are designed using Bloom filters [7], which are compact data structures suitable for hardware implementation in fast but small on-die SRAM memory. A Bloom filter encodes the membership of a set in a memory-efficient way. It is a bit array initialized to zeros. When inserting an element, we hash its identifier to k bits in the array and set them to ones. To look up for the membership of an element, we also hash it to k bits in the array and see whether these bits are all ones. If they are, we claim that the element is in the encoded set; if any of the bits is zero, we claim that the element is not in the set. (Note that a standard Bloom filter is originally designed to encode a single set and return binary information for each element under lookup — whether the element is in the set or not.)

This paper studies a more difficult, yet less investigated problem, called multi-set membership lookup, which involves multiple sets (sometimes in hundreds or even thousands). It determines not only whether an element is a member of the sets but also which set it belongs to. In addition to the binary information that the standard Bloom filters can provide, this lookup function also returns a set ID.

The multi-set membership lookup function can be used to classify an incoming packet stream into different categories, based on addresses, ports, a combination of them and other fields in packet headers, or even packet content. It has many important applications. For example, the routers of an ISP may classify packets for differentiated services based on their sources which are placed into different sets according to different types of service contracts. A firewall may be supplied with an action list for addresses that are collected by an intrusion detection system. Depending on the suspicious

Manuscript received February 10, 2015; revised October 28, 2015; accepted January 1, 2016; approved by IEEE/ACM TRANSACTIONS ON NETWORKING Editor A. X. Liu. Date of publication March 11, 2016; date of current version December 15, 2016. This work was supported in part by the Natural Science Foundation under Grants CNS 1115548 and CNS-1409797 and in part by the Chinese Natural Science Foundation under Grant 61472256.

Y. Qiao is with Google Inc., Mountain View, CA 94043 USA (e-mail: yqiao@google.com).

S. Chen and Z. Mo are with the Department of Computer and Information Science and Engineering, University of Florida, Gainesville, FL 32611 USA (e-mail: sgchen@cise.ufl.edu; zmo@cise.ufl.edu).

M. Yoon is with Kookmin University, Seoul 02707, South Korea (e-mail: mkyoon@kookmin.ac.kr).

Digital Object Identifier 10.1109/TNET.2016.2536618

levels of source addresses, some packets may be logged, some may be content inspected, while others may be dropped. The firewall must check each arrival packet to see if the source address is a member of the sets, and if it is, what further action it should take, depending on which set the packet belongs to. In another example, as the VMs are placed onto the servers of a data center, the gateway can be tasked to determine which incoming packets should be forwarded to which server, as each server hosts a different set of VMs.

Traditional exact-match data structures such as binary search tree [8], trie [9], and hash table [10] have to store both keys and values (i.e., set IDs in the context of this paper). Some of them also need additional space for pointers to maintain the tree structure or resolve hash collision by using a linked list for collided keys. These data structures are often considered to be too expensive to be implemented entirely in cache memory of a network processor for line-speed packet inspection. To address this challenge, researchers have strived to adopt "compact" data structures. Along this line of research, variants of Bloom filters are introduced to encode set IDs in the filters such that not only can we determine whether an element is a member but also which set the element belongs to [11] and [12]. However, through a thorough analysis of the mechanism and the performance of existing work, quite to our surprise, we find that Bloom filters are actually not efficient for encoding set IDs. More specifically, each set ID is represented in the prior work as *multiple bits in a code*, and each of those bits is encoded by *multiple bits in a Bloom filter*. This multiplying factor not only causes significant memory overhead, but also results in numerous memory accesses per lookup, particularly under stringent lookup accuracy requirements. In short, Bloom filters are not compact in the current approaches for encoding set IDs. Moreover, encoding IDs directly in the filters makes each lookup expensive in processing.

This paper takes a different solution path by separating membership encoding and set ID storage into two data structures, called *index filter* and *set-id table*, respectively. The former is a variant of Bloom filter with an efficient two-level design, which does only what a Bloom filter can do best, encoding membership (instead of set IDs).[1] The latter is a multi-hash table, whose compactness is ensured by allowing each element to store its set ID in one of multiple candidate entries in the table. Notably, we discover a new ID placement strategy of *uneven candidate-entry distribution*, which is able to reduce the insertion-failure probability by orders of magnitude sometimes. We demonstrate through analysis and simulation that when comparing with the best existing work, our new approach is able to achieve significant memory saving under the same lookup accuracy requirement, or achieve significantly better lookup accuracy under the same memory constraint.

The rest of the paper is organized as follows: Section II defines the problem and the performance metrics. Section III discusses the related work and compares their theoretical bounds. Section IV motivates our idea and explains the technical details of our proposed solution. Theoretical analysis on the overhead and accuracy of our proposed data structure is in Section V. Section VI presents the evaluation results. Section VII concludes this paper.

[1]The paper title is not to question that Bloom filters are compact data structures in general but suggest that they are not compact for encoding set IDs.

II. Problem Definition

A. Multi-Set Membership Lookup

Consider a number g of disjoint sets whose IDs are $1, 2, \ldots, g$, respectively. Each set contains a number of member elements; Each member element is in a single set. Given an arbitrary element e, the multi-set membership lookup function is to find the set ID that e belongs to. Suppose we want to implement the lookup function in the on-die memory (such as SRAM) of a network processor for high throughput, which requires compact data structures to represent the sets. When comparing with online lookup (which may happen on a per-packet basis), inserting or deleting an element of a set is considered to be infrequent, and is thus allowed to access off-chip memory. Following the assumption of previous works [11], [12], we consider the sets to be disjoint, which is the case for all applications mentioned in the introduction. (For other potential applications where some sets share elements, we may take their intersections out new sets to make them disjoint and possibly merge the resulting sets based on the common actions to be performed on them — in this case, the number of final sets is limited by the number of possible actions.) The output of lookup for an element is either the ID of a set that e belongs to, or 0 if e does not belong to any set.

B. Performance Metrics

Modern high-speed routers forward packets from incoming ports to outgoing ports via switching fabric. In order to keep up with the line speed, the trend is to implement online network functions for packet classification, access control, and traffic measurement using on-die cache memory and bypassing main memory and CPU almost entirely [1], [13], [14]; note that the three applications of multi-set membership lookup in the introduction are cases of packet classification.

However, fitting online network functions in fast but small on-die memory represents a major technical challenge [15]. The commonly-used cache memory on network processor chips is SRAM, which has limited size. There is a huge incentive to keep on-die memory small because smaller memory can be made faster and cheaper. To make the matter worse, on-die memory may have to be shared by multiple routing/performance/measurement/security functions that are implemented on the same chip. When multiple network functions share the same memory, each of them can only use a fraction of the available space, whereas the amount of data they have to process and store can be extremely large in high-speed networks. The disparity in memory demand and supply requires us to implement these online functions, including the ones based on multi-set membership lookup, as compact as possible. The processing overhead for on-chip implementation is characterized by number of memory accesses and computation overhead.

TABLE I

PERFORMANCE COMPARISON OF EXISTING APPROACHES, WHERE n IS THE NUMBER OF ELEMENTS IN ALL SETS, g IS THE NUMBER OF SETS, ϵ IS THE ACCURACY REQUIREMENT, f IS THE CODE LENGTH IN [11] AND [12], AND θ IS THE NUMBER OF ONES IN EACH CODE

	pre-known set sizes	mis-classification	space requirement	memory accesses per lookup
one filter per set (OFPS)	yes	no	$-\frac{n\ln(\epsilon/g)}{(\ln 2)^2}$	$-\frac{g\ln(\epsilon/g)}{\ln 2}$
coded Bloom filters [17]	yes	yes	$-\frac{n\log_2 g\ln(\epsilon/\log_2 g)}{2(\ln 2)^2}$	$-\frac{\log_2 g\ln(\epsilon/\log_2 g)}{\ln 2}$
sparsely coded filters [11]	yes	no	$-\frac{\theta n\ln(\epsilon/(f-\theta))}{(\ln 2)^2}$	$-\frac{f\ln(\epsilon/(f-\theta))}{\ln 2}$
combinatorial Bloom filters [12]	no	no	$-\frac{\theta n\ln(\epsilon/(f-\theta))}{(\ln 2)^2}$	$-\frac{f\ln(\epsilon/(f-\theta))}{\ln 2}$
Bloomier filter [18]	no	no	$-\frac{n(\log_2 g+1)\ln\epsilon}{(\ln 2)^2}$	$-\frac{\ln\epsilon}{\ln 2}$

TABLE II

NUMERICAL COMPARISON OF EXISTING APPROACHES WITH PROPOSED iSet, WHERE $n = 500{,}000$, $g = 5{,}000$, $\epsilon = 0.001$, $f = 16$, AND $\theta = 6$

	pre-known set sizes	mis-classification	space requirement	space per element	memory accesses per lookup
one filter per set (OFPS)	yes	no	1.6×10^7	32	115000
coded Bloom filters [17]	yes	yes	4.3×10^7	86	126
sparsely coded filters [11]	yes	no	4.8×10^7	96	224
combinatorial Bloom filters [12]	no	no	4.8×10^7	96	224
Bloomier filter [18]	no	no	7.2×10^7	144	10
Proposed iSet	no	no	1.5×10^7	30	$2 \sim 10$ (6.5 on average)

The performance criteria considered in our design of the lookup function are given below:

- **Space:** We should reduce the number of bits it takes to encode each member element and its set ID. This is extremely important when the data structures are placed in on-die SRAM, which is small.
- **Memory Access:** We should reduce the number of memory accesses per membership lookup. This is particularly important if the lookup operation is performed very frequently, e.g., on a per-packet basis in a router.
- **Computation:** We also want to reduce the hash complexity for each membership lookup. This helps reduce the computational overhead.

This paper investigates probabilistic lookup data structures which can be made very compact. We define a few concepts on evaluating accuracy:

- **False Positive:** For a non-member element that does not belong to any set, false positive happens if the lookup function mistakenly believes the element belongs to a set.
- **Conflict Classification:** For a member element of a set, conflict classification occurs if the lookup function cannot definitively determine the right set ID but knows the element must belong to one of several candidate sets.
- **Mis-Classification:** For a member of a set, mis-classification occurs when (1) the lookup function claims that the element belongs to a different set, or (2) the lookup function claims that the element does not belong to any set.

Most prior work does not have the problem of mis-classification; neither does our solution in this paper.

Let *the false-positive ratio* be the probability for false positive to occur to a non-member after lookup, and *the conflict classification ratio* be the probability for conflict classification to occur to a member. From the space point of view, there are two ways to compare the performance of different designs of the lookup function. First, given a preset accuracy requirement ϵ, we want to minimize the amount of memory it takes to bound both false-positive ratio and conflict classification ratio by ϵ. Second, if the amount of memory that can be allocated for the lookup function is fixed, we want to have a lookup design that reduce the false-positive ratio and the conflict classification ratio as much as possible. Regardless of fixed space or not, it is important to reduce the number of memory accesses and the amount of computation per lookup for high throughput applications.

III. RELATED WORK

Bloom filter [7] encodes the membership of a set. It has false positive. From the results of [16], we can easily derive the following: To bound the false positive ratio by ϵ, the size of the bit array should be at least $-n'\ln\epsilon(\ln 2)^2$ with $k = -\ln\epsilon/\ln 2$, where n' is the number of elements in the set. For example, when $\epsilon = 0.001$, the filter size is $14.4n'$ with $k = 10$.

A Bloom filter encodes a single set. To handle multiple sets, a naive approach is to use one separate filter for each set. This method has two serious problems: First, without knowing the size of each set, it is hard to divide memory among the filters in appropriate proportions (such that a larger set will get a bigger portion). Hence, the method works well only when the set sizes are pre-known. Second, each membership lookup has to examine all filters, which takes too many memory accesses. Let g be the number of groups and ϵ be the accuracy requirement for false positive (and conflict classification as well when applicable). If each filter has a false positive ratio of $\frac{\epsilon}{g}$, it takes $-\frac{\ln(\epsilon g)}{\ln 2}$ memory accesses per lookup according to [16]. The overall false position ratio for all g filters is $1-(1-\frac{\epsilon}{g})^g$, which is approximately ϵ when g is large and ϵ is very small. Because there are g filters, the total number of memory accesses per lookup by the method of one filter

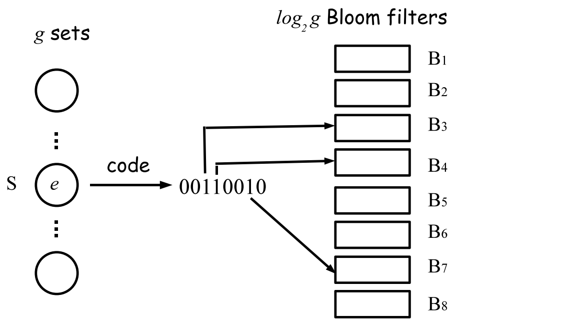


Fig. 1. Coded Bloom filter [17] — There are g sets. Each circle on the left represents a set. Consider an arbitrary set S, which is assigned a code of $\lceil \log_2(g+1) \rceil$ bits, e.g., 00110010. Each bit in the code corresponds to a Bloom filter represented by rectangles on the right. For instance, eight bits in the code correspond to eight filters, B_1 through B_8. For every element e in set S, if and only if a bit in the code is one, e is inserted (encoded) in the corresponding filter by hashing to k bits in the filter and setting them to ones. In this example, e will be encoded in B_2, B_3 and B_6.

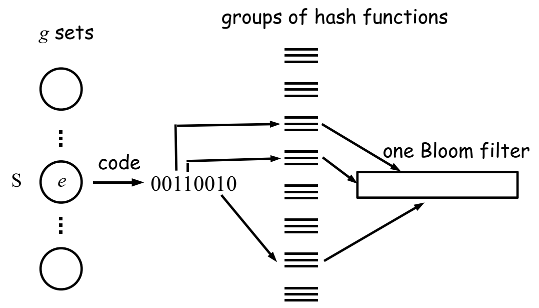


Fig. 2. COMB [12] — Consider an arbitrary set S, which is assigned a sparse code with θ ones, e.g., 00110010 with $\theta = 3$. Each bit in the code corresponds to a group of k hash functions, which is represented by three bars in the middle of the figure. For instance, eight bits in the code correspond to eight groups of hash functions. There is only a single Bloom filter. For every element e in set S, if and only if a bit in the code is one, the corresponding group of hash functions will be used to encode e in the filter by hashing e to k bits and setting those bits as ones.

per set (OFPS) is thus $-\frac{g\ln(\epsilon g)}{\ln 2}$, as shown in in Table I. To make the comparison intuitive, we also give numerical results in Table II for a typical parameter setting. OFPS makes 115000 memory accesses for each element lookup.

To deal with the second problem above, Chang *et al.* [17] proposes a coded Bloom filter approach, which assigns each of the g sets a distinct code from 1 to g, as illustrated in Figure 1. The code is $\lceil \log_2(g+1) \rceil$ bits long. There is one Bloom filter for each bit position in the code. As an example, for 8-bit codes, there will be 8 Bloom filters, B_1 through B_8, for the first through eighth bits, respectively. Suppose the code of a set S is 00110010. Its third, fourth and seventh bits are ones. Each element in S will be inserted into B_3, B_4, and B_7, respectively. To look up an element, we check all $\lceil \log_2(g+1) \rceil$ Bloom filters (instead of g filters in the previous approach). In fact, the number of memory accesses is reduced by a factor more than $\frac{g}{\lceil \log_2(g+1) \rceil}$ when the same ϵ is enforced, as shown in Table I where we substitute $\lceil \log_2(g+1) \rceil$ with $\log_2 g$ for clarity. Continue with our example. For any element in S, the lookup results from the 8 filters are expected to be '0', '0', '1', '1', '0', '0', '1', and '0', respectively, where '0' means not in the filter and '1' means in the filter. Putting them together yields the code of S. This approach still requires preknown set sizes, but it alleviates the memory access problem from 115000 to 126 in Table II. However, it causes a new problem of misclassification: a false positive in any of the $\lceil \log_2(g+1) \rceil$ filters will misclassify an element to a wrong set. In the example above, if the lookup result from the first filter is '1', the code will be 10110010, referring to a different set than S.

Lu *et al.* [11] solve the misclassification problem by using sparse codes whose length f is much longer than $\lceil \log_2(g+1) \rceil$, but each code carries a small, fixed number θ of ones. Still using the previous example, if each code is supposed to have exactly three ones, 10110010 will not refer to any set. The element will be treated as a case of conflict classification. Misclassification is thought to be more harmful than conflict classification [11]; the latter can at least be addressed through a slow path of checking the sets directly.

Hao *et al.* also use sparse codes in their combinatorial Bloom filter (COMB) [12]. It removes the requirement of preknown set sizes by a radically different way of using the codes. They use a single Bloom filter, so that they do not need to worry about proportions in memory allocation for different filters. Each bit position in an f-bit code corresponds to a group of k hash functions, as illustrated by Figure 2, where $k = -\frac{\ln(\epsilon/(f-\theta))}{\ln 2}$. As an example, for each element in a set S of code 00110010, the third/fourth/seventh groups of hash functions are used. They together map the element to $3k$ bits in the filter, and those bits will be set to ones. To look up an element, all f groups of hash functions are used. Each group maps the element to k bits in the filter; the lookup result is '1' if all k bits for one group are ones or '0' otherwise. Putting the lookup results together yields the code of the set, or if it is not the code of any set, the element is rejected.

Because COMB can handle a dynamic set, we should interpret n as the maximum number of elements in all sets that can meet the accuracy requirement ϵ. Similar to [11], COMB needs to use θ groups of hash functions to set θk bits when encoding each element, and use all f groups of hash functions to fetch fk bits when looking up an element. The problem of COMB is that it can only support $\binom{f}{\theta}$ groups, meaning both θ and f cannot be too small if g is large, which will translate into substantial space requirement and memory accesses per lookup, as shown in Table II. COMB requires 96 bits per element, comparing with 32 bits by OFPS. More importantly, the number of memory accesses per lookup by COMB is still very high, 224. To reduce conflict classification, COMB may use error-correction codes (such as Hamming code), which will however increase the code length f, resulting in more memory accesses per lookup.

Our goal is to use only one group of hash functions when inserting or looking up an element, as what OFPS does, yet we use far fewer memory accesses, more specifically, a small number of memory accesses per lookup. The new lookup function should not assume pre-known set sizes and does not have mis-classification. To achieve this goal, we will have to move away from the traditional coded approaches and resort to a very different design in data structures.

Another related work is the Bloomier filter [18], which expands each bit in a Bloom filter with a multi-bit entry, storing a value, e.g., a set ID. The efficient encoding design in [18] assumes the pre-knowledge of all elements in the sets. However, we may modify it to support dynamic sets: To insert an element e whose set ID is X, we hash e to k entries in the Bloomier filter, and make sure that the XOR of these entries is equal to X. There is one bit per entry indicating whether the entry has been used to encode a previous element. Insertion is successful only when at least one of the k entries is unused. To perform lookup on e, we again hash e to its k entries. If all entries are used, their XOR will give the set ID. This approach uses more memory than others.

IV. Design of iSet

We propose a new design of efficient multi-set lookup function called *iSet*. The name comes from its two key data structures, $i$ndex filter and set-id table.

A. Motivation

What's in common for the prior work [11], [12], [17], [18] is that they try to encode both membership and set IDs in the same data structures — variants of Bloom filters — which were originally designed for membership only. A Bloom filter is efficient for encoding the binary membership information (in or not in the set) for each element. Even though the encoding takes multiple bits, they are fewer than directly storing the identifier of an element. Moreover, it eliminates the explicit indexing overhead that maps each element to an address where the information is stored.

However, when the set IDs are also stored in this data structure, it becomes inefficient in both storage and accessing. Each ID contains multiple-bit information, no matter whether it is in the form of the original set ID or a code. Each ID bit is stored in the filter as binary information, in the same way as the membership is stored, which takes multiple bits in the filter (e.g., 10 bits for a false positive ratio of 0.001). This multiplying effect — multiple bits in an ID and each of them requiring multiple bits in the filter — not only costs memory but also results in numerous memory accesses to the filter.

Moreover, in order to keep the false positive ratio of a Bloom filter down, the filter must have a significant portion of zeros, more specifically, one half of all bits should be zeros for an optimal filter. When we need a lot of bits to encode each set ID, it also means we need more bits to keep the same proportion of zeros, which causes even more storage overhead.

With these observations, our insight is to separate membership encoding and ID storage in two data structures, *index filter* and *set-id table*, as illustrated in Figure 3. The index filter is a variant of Bloom filter, which does only what it does the best, encoding membership once per element. The set-id table is a multi-hash table, which stores the set IDs of member elements. Each member element e is mapped to a number λ of entries in the table through hash functions H_1, ..., H_λ, where λ is a small preset constant. The element's set ID, X, can be stored in *any* of the λ entries as long as the entry is unused, where X is

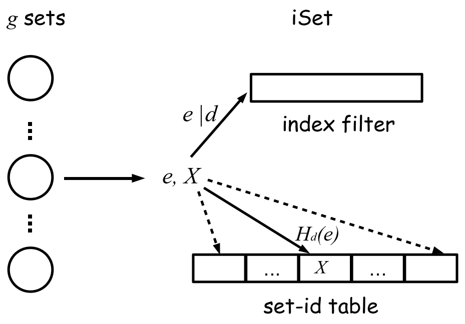


Fig. 3. iSet — Consider an arbitrary element e from an arbitrary set whose ID is X. The element e is hashed to a number λ of entries in the set-id table; see the dashed lines. As long as one of these entries is unused, X can be stored; see the solid line, where X is stored at the entry hashed to via H_d, the dth hash function. We encode the membership of $e|d$ in the index filter, establishing a connection between the index filter and the set-id table through hash index d, which tells where in the table we can find e's set ID.

TABLE III
Notations

g	Number of sets
n	Total number of elements in all sets
q	Number of segments in the set-id table
l	Number of entries in the set-id table
λ	Number of candidate entries per element in the set-id table
$C(e)$	Checksum of an element e
m	Number of bits in the index filter
d	Primary index specifying which entry e's set ID is stored at
$B(e)$	A block in the index filter where $e|d$ is encoded
k	Number of bits to encode $e|d$ in $B(e)$

$\log_2 g$ bits long. The reason for using λ entries is to increase the probability of finding an unused one; alternatively, it is to make sure that the set-id table is about fully used with only a small fraction of wasted entries. Suppose X is stored at the entry which the dth hash function, H_d, maps e to, where d is referred to as a *hash index* or simply *index*.

Not only do we want to encode the membership of e in the index filter, but also we need to establish a connection between the index filter and the set-id table, such that once we find from the filter that e is a member (in some set), we will also learn where to find its set ID in the table. To establish this connection, we encode $e|d$ (instead of e alone) in the index filter by hashing them together to k bits and set those bits to ones, where '$|$' is the concatenation operator.

During lookup, we check e together with all possible hash indices, 1 through λ, to find which one is in the filter. When we find $e|d$ in the filter (for a certain hash index d), we know not only that e is a member, but also where to find its set ID in the table.

Comparing with COMB which encodes θ memberships per element, our index filter only encodes one membership, $e|d$, for each element e. The lookup of e involves λ membership checks, comparing with g membership checks in OFPS and f checks in COMB. However, the value of λ can be made much smaller than g and f. We will show that the space in the set-id table is well utilized even for small λ.

In the following, we will present the set-id table with its interesting design for *uneven candidate-entry distribution* and the index filter with a two-level design to reduce the number

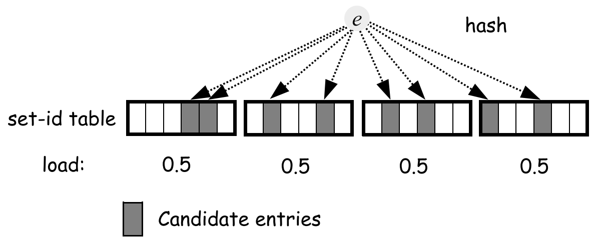


Fig. 4. An example with even candidate-entry distribution, where $q = 4$ and $\lambda = 8$. When the load of each segment is 0.5, the insertion-failure probability is about 3.9×10^{-3}.

of memory accesses per lookup to one. Notations used in the description can be found in Table III for quick reference.

B. Set-id Table and Uneven Candidate-Entry Distribution

Let l be the total number of entries in the set-id table, which is divided into q segments, each having $\frac{l}{q}$ entries. Technically speaking, we may allow variable-sized segments [19], [20]. However, equal-sized segments are easier to handle in practice for multi-banked memory and dynamic memory allocation. Each entry consists of two fields: a checksum and a set ID. The checksum is used for resolving conflict classification, which will be discussed when we present the index filter. If the set-id field is zero, it means the entry is unused because valid set IDs range from 1 to g. The *load* of a segment is the fraction of its entries that have been used for storing set IDs.

When we want to insert a member element e whose set ID is X, we hash e to λ *candidate entries* in the table, i.e., $\frac{\lambda}{q}$ entries per segment on average, where $\lambda \geq q$.[2] As long as one of these candidate entries is unused, we will be able to store X. Insertion failure will be handled separately. For now, our concern is to minimize the probability of insertion failure.

• **Argument for uneven candidate-entry distribution:** If $\lambda > q$, it is natural to evenly distribute candidate entries in the segments. However, we show that a natural approach may not be a good one through an example in Figure 4 where $q = 4$, $\lambda = 8$, and each segment has two candidate entries. When inserting a new element, we randomly choose an unused candidate entry. Suppose half of all entries are unused and the loads of all segments are 0.5. Given a new element, because the probability of any candidate entry being used is 0.5, the probability of insertion failure, i.e., all eight candidate entries are used, is $0.5^8 \approx 3.9 \times 10^{-3}$.

In contrast, we show that an uneven candidate-entry distribution can drastically reduce the insertion-failure probability if the loads of the segments are *biased*, which can be easily created by always inserting a new set ID in the *leftmost* unused candidate entry [21]–[23], as illustrated in Figure 5, where we refer to the order from the first segment to the last segment as from *left* to *right*. Because any new set ID is stored in the leftmost segment whenever possible, that segment will certainly have a higher load. Our idea is that we should assign fewer candidate entries to heavily loaded segments and more

[2]The reason to allow $\lambda > q$ is that, with more than one candidate entry in a segment, we are more likely to find unused entries for new insertions, so as to improve the utilization of the segment.

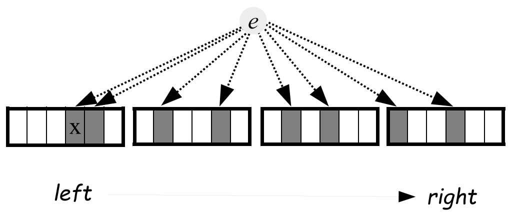


Fig. 5. Storing a set ID in the leftmost unused candidate entry.

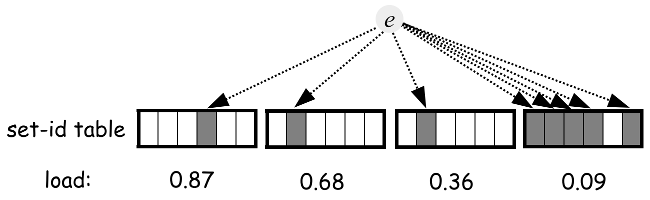


Fig. 6. An example with uneven candidate-entry distribution, $q = 4$ and $\lambda = 8$. After 250,000 elements are inserted into a table of 500,000 entries, the loads of the segments become 0.87, 0.68, 0.36, and 0.09, respectively, with the insertion-failure probability being just 1.3×10^{-6}.

to lightly loaded segments. Since the rightmost segment has the lightest load, a good strategy is to assign one candidate entry to every segment except for the rightmost one, which has the remaining $\lambda - q + 1$ candidate entries, as illustrated in Figure 6, where $q = 4$ and $\lambda = 8$. If we adopt this strategy, the loads of the segments will become 0.87, 0.68, 0.36, and 0.09, respectively, according to our simulation that continuously inserts new elements into a set-id table of 500,000 entries until half of all entries are used. In this case, the insertion-failure probability will become $0.87 \times 0.68 \times 0.36 \times 0.09^5 \approx 1.3 \times 10^{-6}$, a 3000-fold reduction from the even distribution of candidate entries! We will formally evaluate the performance of uneven candidate-entry distribution through analysis and simulation in later sections.

• **Recap on Insertion:** When inserting an element e, we use λ hash functions, $H_i(e), 1 \leq i \leq \lambda$, to map e to one candidate entry in each segment, except for the last segment, which has $\lambda - q + 1$ candidate entries. The set ID of e will be inserted into the first unused candidate entry encountered, which is called the element's *primary entry*. We also insert a checksum computed from e into that entry; the checksum may be another hash of e, denoted as $C(e)$. The index d of the specific hash function $H_d(e)$ that maps e to the primary entry is called the *primary index*.

• **Impact of Insertion Failure:** Now suppose we insert n elements into a set-id table of l entries. We define the *insertion-failure ratio* as the number of elements that failed in insertion divided by n. We stress that it is a different concept than the insertion-failure probability in the example of Figure 6: As we insert 250,000 elements one after another, even though the insertion-failure probability is about 1.3×10^{-6} in the end, it is almost surely 0 at the beginning when the first elements are inserted into the empty table. Hence, the insertion-failure ratio over all 250,000 elements should be much smaller than 1.3×10^{-6}. In fact, we only observed 9 insertion failures during our simulations after inserting 250,000 elements into the table of 500,000 entries for 1000 independent runs, which

Algorithm 1 Insert a New Element

```
INPUT: element $e$, set ID $X$
OUTPUT: N/A
$\{SEG[i]$ is the segment that $H_i$ maps to$\}$

$d := -1$
for $i := 1 \to \lambda$ do
  $j := H_i(e) \bmod (l/q) + SEG[i] \times l/q$
  if sidtable[$j$].id = 0 then
    sidtable[$j$].id := $X$
    sidtable[$j$].checksum := $C(e)$
    $d := i$
    break   // exit for loop
  end if
end for
if $d > 0$ then
  encode $e|d$ in the index filter
else
  store $(e, X)$ in a supplement hash table or TCAM
end if
```

translates into a measured insertion-failure ratio of 3.6×10^{-8}. After inserting 400,000 elements for an average load of 0.8 over the entire table, the insertion-failure ratio is still just 2.5×10^{-3} on average. This suggests that we can control the insertion-failure ratio low by letting l be modestly larger than the maximum number n of member elements to be handled by the system.

When the insertion-failure ratio is very small, for the few elements that cannot be inserted to the set-id table, we can store these elements and their set IDs in a TCAM (ternary content-addressable memory [24]). If TCAM is not available, we can store these elements in a supplement hash table whose size is larger than the expected number of failed elements, so as to ensure that hash collision is negligibly small and thus most lookups take just one memory access. As long as the number of failed elements is very small, the impact of this hash table on the overall space usage will be negligible, when comparing with the much larger set-id table.

C. Index Filter

When inserting a new member element e, we first store its set ID as previously described. If it is in the supplement hash table (or TCAM), nothing more needs to be done. Otherwise, let d be the primary index, and recall that $H_d(e)$ tells where the ID was stored in the set-id table. We encode $e|d$ in the index filter by hashing $e|d$ to k bits and setting those bits to ones, where k will be formally determined based on the accuracy requirement ϵ and other parameters. The key difference between the index filter and the prior work [11], [12], [17] is that the former only encodes membership once for each element in the form of $e|d$, whereas the latter encodes membership multiple times for each element — $\log_2 g$ times in [17] and θ times in [11] and [12]. Encoding membership just once means that we set much fewer bits per element, which in turn means a much smaller filter.

The pseudo code for inserting a member element e with set ID X is given in Algorithm 1, where sidtable[j].*id* and sidtable[j].*checksum* refer to the ID field and the checksum field of the jth entry in the set-id table.

A two-level design of the index filter that optimizes the lookup performance will be introduced after we discuss the lookup procedure below.

D. Multi-Set Membership Lookup

Multiple membership checks are needed for lookup. Given an element e under query, we check the supplement hash table (or TCAM) first, and if it is not there, we perform membership checks on $e|i$, for $1 \leq i \leq \lambda$, in the index filter. The element will be rejected as a non-member if all membership checks turn out to be negative. It is considered as a member if $\exists i \in [1..\lambda]$, $e|i$ is in the filter and e passes a checksum test.

First, our design does not have *mis-classification*: If e is a member, because $e|d$ has been encoded in the filter during insertion, we will find it in the filter and thus the lookup function will not reject the element as a non-member, where d is the primary index of e.

Like [11] and [12], our design has *conflict classification*: For a member e, not only will we find $e|d$ in the index filter, but it is also possible that $e|i$ turns out positive for another index i — the k bits for $e|i$ in the filter happen to be all ones. Unlike the prior work, we have an additional efficient mechanism to mitigate conflict classification. For any $e|i$ found in the filter, we identify the entry in the set-id table using $H_i(e)$, and compare $C(e)$ with the checksum of that entry. The two will match if i is the primary index for e. Otherwise, the chance of matching is $\frac{1}{2^s}$, where s is the length of the checksum. In other words, we reduce the conflict classification ratio by a factor of $\frac{1}{2^s}$ by using the checksum. If the checksum of only one entry matches $C(e)$, we report the set ID in that entry as the lookup result. If there are more than one match, we report conflict classification. If necessary, the lookup may proceed on a slow path to access off-chip memory where the original data of all sets are stored.

The checksum field also helps reduce the *false positive* ratio: For a non-member e, we may find an index i such that $e|i$ is in the filter. However, e will not be accepted as a member right away. We will verify the checksum of the corresponding entry in the set-id table against $C(e)$, which reduces the false positive ratio by a factor of $\frac{1}{2^s}$.

The pseudo code for looking up for an element e is given in Algorithm 2.

E. Example of Insertion and Lookup

We use a simple example to to illustrate the operations of iSet. Suppose there are two sets: set 1 has two elements, A and B; set 2 has two elements, X and Y. The set-id table has two segments, each having three entries. The index filter contains one word of 16 bits. Each element is mapped to two candidate entries in the set-id table and two bits in the index filter.

Algorithm 2 Look Up for an Element

```
INPUT: element $e$
OUTPUT: a list of set IDs, denoted as *RESULT*.
  – If *RESULT* = $\emptyset$, $e$ is a non-member;
  – If *RESULT* = $\{X\}$, $e$ belongs to set $X$;
  – Otherwise it is a conflict classification.

if $(e, X)$ is found in TCAM then
  return $\{X\}$
end if
RESULT := $\emptyset$
for $i := 1 \rightarrow \lambda$ do
  if $e|i$ is encoded in the index filter then
    $j := H_i(e)\ \mathrm{mod}(l/q) + SEG[i] \times l/q$
    if sidtable[$j$].id != 0 and sidtable[$j$].checksum = $C(e)$
    then
      add sidtable[$j$].id to RESULT
    end if
  end if
end for
return RESULT
```

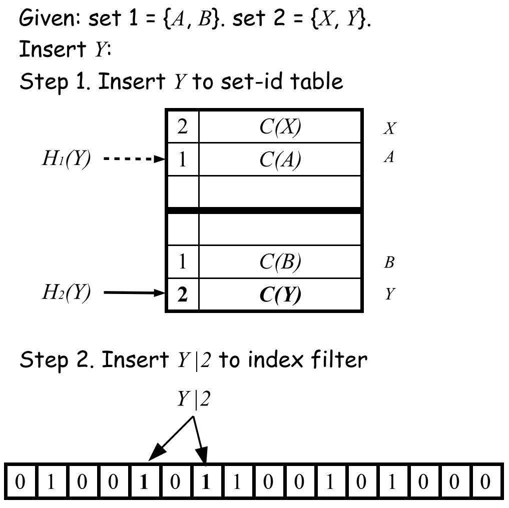

Given: set 1 = {A, B}. set 2 = {X, Y}.
Insert Y:
Step 1. Insert Y to set-id table
H1(Y)
H2(Y)
2 C(X) X
1 C(A) A
1 C(B) B
2 C(Y) Y
Step 2. Insert Y|2 to index filter
Y|2
0 1 0 0 1 0 1 1 0 0 1 0 1 0 0 0

Fig. 7. Insertion of Y in iSet.

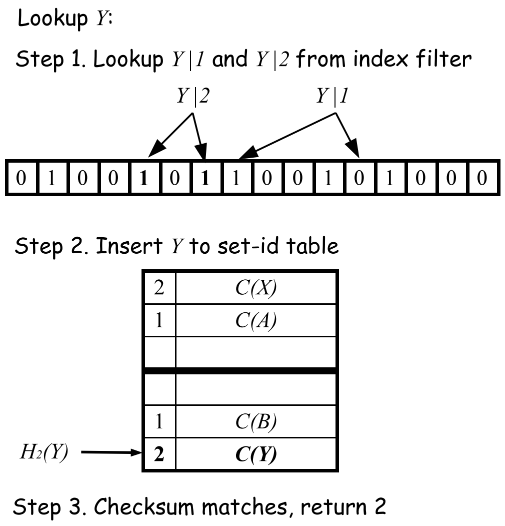

Lookup Y:
Step 1. Lookup Y|1 and Y|2 from index filter
Y|2
Y|1
0 1 0 0 1 0 1 1 0 0 1 0 1 0 0 0
Step 2. Insert Y to set-id table
2 C(X)
1 C(A)
1 C(B)
H2(Y)
2 C(Y)
Step 3. Checksum matches, return 2

Fig. 8. Look up for Y in iSet.

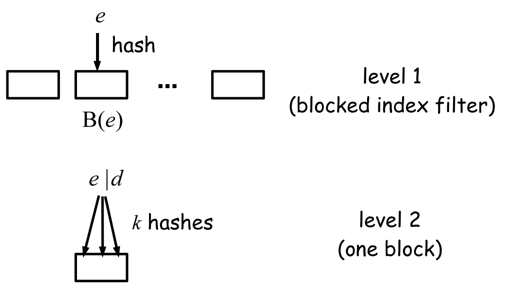


Fig. 9. Index Filter — To encode $e|d$, we first hash e to a block in the filter. After fetching the block to the processor, we hash $e|d$ to k bits in the block and sets the bits to ones.

1) Inserting Y to iSet: Fig. 7 shows iSet with A, B, and X already encoded. We now go through the steps to encode Y. First, we map Y to two candidate entries in the set-id table, using two hash functions, H_1 and H_2. As shown in Fig. 7, the first candidate entry (in the first segment) is already taken; its set-id field is non-zero with the information of element A. So we store the set-id and the checksum of Y to the second candidate entry. Next, we insert $Y|2$ to the index filter by setting the mapped bits to ones, where "2" refers to the "second" candidate entry.

2) Lookup Y From iSet: In order to look up the set-id of Y, we first query the index filter with $Y|1$ and $Y|2$. As shown in Fig. 8, only $Y|2$ is encoded in the filter. So we check the second candidate entry of Y in the set-id table. By verifying the checksum stored in the entry, which matches $C(Y)$, we return 2 as the set-id of Y.

F. Refined Design of Index Filter

Although encoding an element in the filter sets k bits, looking up an element requires λ membership checks, which together makes $k\lambda$ memory accesses. The value of k determines the false positive ratio of the filter. Thanks to the checksum, we can afford a much larger false positive ratio than ϵ, which means a much smaller value for k (such as $2 \sim 3$ in our simulations). A larger value λ helps improve the utilization of the set-id table, but we observe through simulation that it does not bring any significant gain by setting λ beyond 10. Ten candidate entries per element will ensure that the table is largely occupied, and trying more candidate entries do not help much because the table is already very full.

Nevertheless, we want to further reduce the number of memory accesses per lookup from $k\lambda$ to just one through a two-level filter design based on the block idea from [25] and [26]. As illustrated in Figure 9, at the first level, the filter is divided into consecutive blocks, each of which may be 64 bits long and can be retrieved in one memory access. At the second level, each block consists of consecutive bits, which encode membership. The insertion of element e with primary index d consists of two steps: First, we hash e to a block in the index

Algorithm 3 Encode $e|d$ in the Index Filter

```
INPUT: element $e$, primary index $d$
OUTPUT: N/A
{$H''$, $H'_{1..k}$ are hash functions; assume each block has 64 bits.}

read $B(e)$ as the $H''(e)$th block of the index filter
for $i := 1$ to $k$ do
    $B(e)[H'_i(e|d) \bmod 64] := 1$
end for
write $B(e)$ back
```

filter and fetch the block to the processor, where the block is treated as a small Bloom filter, denoted as $B(e)$. Second, the processor hashes $e|d$ to k bits in $B(e)$ and sets them to ones. After that, the block is written back to memory. The pseudo code is given in Algorithm 3. This approach works only if membership encoding sets a small number of bits. If a large number of bits are set for each element as the prior work does, the block of 64 bits will contain mostly ones, resulting in poor false-positive ratio.

While insertion takes two memory accesses (one read and one write), looking up an element e takes just one (read) for fetching $B(e)$ to the processor, where the membership checks are locally performed.

G. Deletion

None of the prior work [11], [12], [17] considers deletion. However, deletion can be supported by using counting Bloom filters [27], [28], which replace each bit in the filters with a counter. When inserting an element, instead of setting a certain number of bits in the filters to ones, we increase the corresponding counters, each by one. When deleting an element, we simply decrease those counters by one. We can set the size of each counter sufficiently large to prevent overflow, and store the full-sized counters in the off-chip memory. Scaled-down counters of c bits each are stored in the on-die memory. It has been shown by [27] that when $c = 4$, overflow of these scaled-down counters happens rarely. Increase/decrease of counters mostly happen on chip. Only when overflow happens, we will write the overflowed counter to its off-chip counterpart. If a deletion requires to decrease an overflowed counter of value 11…1, we need to access the corresponding counter off-chip to decrease there. The impact will be negligible when this operation is infrequent [29].

The same method can be used to support deletion of an element from the index filter. We remember which entry in the set-id table stores which element's set ID in off-chip memory, which helps us remove the set ID when an element is deleted.

V. Performance Analysis and Parameter Setting

A. False Positive Ratio

Recall that the *false positive ratio* is the probability for a non-member element e to be considered as a member. False positive happens only if both of the following conditions are satisfied:

(a) $\exists i \in [1..\lambda]$, $e|i$ is found in the index filter.

(b) The checksum field of the corresponding entry in the set-id table fails in resolving the false positive, i.e. the checksum is the same as $C(e)$.

We define p as the probability for an arbitrary index $e|i$, $i \in [1..\lambda]$, to be found in the index filter. Suppose n member elements have been encoded in the filter, each setting k bits. In total, kn bits are set (with replacement) to ones. The probability for an arbitrary bit in the filter to be one is $1-(1-\frac{1}{m})^{kn}$, where m is the number of bits in the index filter. The probability for all k bits of $e|i$ to be ones by chance is

$$p = \big(1-(1-\frac{1}{m})^{kn}\big)^k \approx (1-e^{-kn/m})^k. \quad (1)$$

With the refined design of the index filter, the exact form of p is different due to the block idea. However, when k is small, it can be approximated by (1) with a negligible difference [26], which we have also confirmed by simulation.

Let X be the random variable for the number of different indices, $e|i$, $1 \le i \le \lambda$, found in the index filter. It follows a Binomial distribution, $Bino(\lambda, p)$. Hence,

$$Prob\{X = x\} = \binom{\lambda}{x} p^x (1-p)^{\lambda-x}, \quad \forall\ 1 \le x \le \lambda. \quad (2)$$

For any index found in the filter, the probability for the corresponding entry in the set-id table to have a checksum equal to $C(e)$ is $\frac{1}{2^s}$. Hence, the false-positive ratio is

$$\begin{aligned} P_{false} &= \sum_{x=1}^{\lambda} Prob\{X = x\}\big(1-(1-\frac{1}{2^s})^x\big) \\ &= 1 - \sum_{x=0}^{\lambda} \binom{\lambda}{x} p^x (1-p)^{\lambda-x}(1-\frac{1}{2^s})^x \\ &= 1-(1-\frac{p}{2^s})^{\lambda} \approx \frac{\lambda p}{2^s} = \frac{\lambda(1-e^{-kn/m})^k}{2^s}. \quad (3) \end{aligned}$$

B. Conflict Classification Ratio

The conflict classification ratio is the probability that the lookup function can not determine a unique set ID for a member element. Conflict classification happens when both of the following conditions are satisfied:

(a) $\exists i \in [1..\lambda]$, $i \neq d$, $e|i$ is found in the index filter.

(b) The checksum field of the corresponding entry in the set-id table fails in resolving the false positive, i.e. the checksum is the same as $C(e)$.

The conditions are almost identical to those for false positive, except that $e|d$ is in the filter. Hence, based on a similar analysis, we know that the number of different indices, $e|i$, $1 \le i \le \lambda$ and $i \neq d$, found in the index filter follows a Binomial distribution, $Bino(\lambda - 1, p)$. The conflict classification ratio is

$$P_{conflict} = 1-(1-\frac{p}{2^s})^{\lambda-1} \approx \frac{(\lambda-1)(1-e^{-kn/m})^k}{2^s}. \quad (4)$$

Comparing (3) and (4), it is easy to see that $P_{conflict} \le P_{false}$. Hence, for a given accuracy requirement ϵ, we should only make sure that

$$P_{false} \le \epsilon. \quad (5)$$

C. Insertion Failure

With n elements being inserted into the set-id table, we give a procedure to approximately compute the number of elements that fail during insertion and have to be placed in the supplement hash table. According to the procedure of insertion, the elements are first mapped to the first segment of $\frac{l}{q}$ entries through hashing. For an arbitrary entry, it will be used to store a set ID as long as at least one element is mapped to it, the probability of which is $1-(1-\frac{q}{l})^n$. The expected number of used entries in the first segment, denoted as u_1, is

$$u_1 = (1-(1-\frac{q}{l})^n)\frac{l}{q} \approx (1-e^{-\frac{qn}{l}})\frac{l}{q}. \qquad (6)$$

Suppose exactly u_1 elements store their set IDs in the first segment. As we map $n-u_1$ to the second segment of $\frac{l}{q}$ entries, by a similar analysis, we can derive the expected number of used entries in the second segment. Through induction, we have the general formula for the approximated number of used entries in the ith segment as follows

$$u_i = (1-e^{-\frac{q(n-\sum_{j=1}^{i-1}u_j)}{l}})\frac{l}{q}, \quad 2 \le i \le q-1. \qquad (7)$$

Suppose u_i elements, $1 \le i \le q-1$, store their set IDs in the ith segment. The remaining elements are mapped to the last segment, each being hashed to $\lambda-q+1$ candidate entries. Let u_i, $q \le i \le \lambda$, be the estimated number of elements stored in the last segment by using the ith candidate entry. Following a similar analysis as previously done, we have

$$\begin{aligned} u_i &= (1-(1-\frac{q}{l})^{(n-\sum_{j=1}^{i-1}u_j)})(\frac{l}{q}-\sum_{j=q}^{i-1}u_j) \\ &\approx (1-e^{-\frac{q(n-\sum_{j=1}^{i-1}u_j)}{l}})(\frac{l}{q}-\sum_{j=q}^{i-1}u_j), \quad q \le i \le \lambda, \end{aligned} \qquad (8)$$

where we define $\sum_{j=q}^{q-1}u_j = 0$. Hence, the number U of elements to be placed in the supplement hash table is approximately

$$U = n - \sum_{j=1}^{\lambda} u_j. \qquad (9)$$

Alternatively, given the values of n, q and λ, we may use simulations to find the average number of elements in the supplement table with respect to l, and store the results in a table for lookup.

D. Space Requirement

The memory space taken by the index filter is m. Each entry in the set-id table has $\lceil \log_2(g+1) \rceil + s$ bits, where $\lceil \log_2(g+1) \rceil$ bits are used to store set ID, and s bits is used for checksum. There are l entries in the set-id table. So the memory space taken by the set-id table is $l\big(\lceil \log_2(g+1) \rceil + s\big)$. As a result, the overall memory usage of iSet is,

$$M = m + l\big(\lceil \log_2(g+1) \rceil + s\big). \qquad (10)$$

Given an accuracy requirement (5), we will show how to compute the system parameters, including m, l, s and thus M, shortly.

E. Memory Access per Lookup

To look up for an element e, we first access one block $B(e)$ from the index filter, and then fetch data from $0 \sim \lambda$ entries of the set-id table, depending on the lookup result from $B(e)$. As a result, the number of memory accesses for each lookup falls in the range of $[1, 1+\lambda]$.

F. Setting System Parameters

Given an accuracy requirement ϵ, the maximum number n of elements under which the accuracy requirement should be met, a bound b on the maximum number of memory accesses per lookup for both index filter and set-id table, and a desired insertion-failure ratio α, we want to determine the values of the system parameters, including l, λ, q, m, k, and s.

To look up an element, the index filter and the supplementary hash table take two memory accesses. In the worst case, all λ indices of the element are found in the filter and thus the set-id table takes λ memory accesses. Hence, we should set $\lambda = b-2$. For example, if we allow 10 memory accesses per lookup for both index filter and set-id table in the worst case, then λ should be set to 8. Note that the average number of memory accesses to the set-id table is much smaller than λ. Once λ is decided, q should be smaller than λ (e.g., by two) to allow multiple candidate entries in the last segment of the set-id table for reduction of the insertion-failure probability.

We want to control the number of elements stored outside the set-id table due to insertion failure by

$$U \le \alpha n, \qquad (11)$$

where the ratio α is determined based on the size of TCAM if that is available or is set sufficiently small (e.g., 1%) such that the size of the supplement hash table does not impose a significant space overhead. Using (6)-(9) or the lookup table from simulation, we can numerically compute the minimum value of l that satisfies (11) through bi-section search in a range $[0, 2n]$, where $l = 2n$ means an average load of 0.5 and at this load we can rarely see any insertion failure as we have discussed previously.

Next, we determine the values m, k and s together numerically based on the accuracy constraint of (5). We set the range of k from 1 to 64 (the block size), the range of s from 0 to $\log_2 \frac{\lambda}{\epsilon}$ (which by itself can ensure a false positive ratio of ϵ). Because the ranges of k and s are small, we use two loops for exhaustively search. For each pair of k-s values, we set $m = \frac{nk}{\ln 2}$, which is the size of a Bloom filter with optimal k; we then find the smallest overall space M that satisfies the accuracy requirement (5), where $M = m + l(\lceil \log_2(g+1) \rceil + s)$ is the size of the index filter plus the size of the set-id table. In summary, we try to find

$$\min_{k \in [1..64], s \in [0..\log_2 \frac{\lambda}{\epsilon}]} \{M \mid P_{false} \le \epsilon\}. \qquad (12)$$

As a numerical example, going back to the comparison of Table II, if $n = 500{,}000$, $\epsilon = 0.001$, $b = 10$ and $\alpha = 1\%$, the system parameters will be set as $\lambda = 8$, $q = 6$, $l = 568{,}182$, $m = 7.2 \times 10^5$, $k = 1$, and $s = 12$. The space requirement is $M = 1.5 \times 10^7$, the space per element is 30 bits, and the average number of memory accesses per lookup is 6.5 for member elements or 6.0 for non-members.

VI. Numerical Evaluation

A. Simulation Setup

The analysis has studied the case of minimizing the memory usage under a given accuracy requirement. In practice, we often face the situation that the amount of on-die memory that can be allocated for the lookup function is limited, and we want to reduce the false-positive ratio and the conflict classification ratio. We use simulation to compare the performance of different lookup function designs under the same space constraint.

Let the amount of available memory be $M = 16$ Mbits. The number g of sets is 5000. The total number n of elements, which are randomly distributed to the sets, varies from 320000 to 800000, such that the average memory per member element changes from 20 bits to 50 bits. After we insert the member elements to the sets, we test the data structure using 800000 non-member elements and 800000 randomly chosen member elements to obtain false positive ratio and conflict classification ratio. The final results are averaged among 10 runs with different random seeds. We use MD5 as the hash function to generate hash bits and CRC32 to generate checksum.

We compare three lookup approaches in terms of average number of memory accesses per lookup, false positive ratio, and conflict classification ratio. The parameter settings are given below. Readers are referred to the original papers for details.

COMB: We choose COMB with $f = 16$ and $\theta = 6$, which can encode up to $\binom{16}{6} = 8008$ sets. We use a group of $k = \ln 2\frac{M}{6n}$ hash functions for each bit in a code to obtain the fewest false positives [12].

Bloomier: We hash each member to $k = \ln 2\frac{l}{n}$ entries in the filter for encoding, where $l = \frac{M}{\lceil \log_2 g+1 \rceil}$ is the number of entries. This will minimize both false positive ratio and insertion failure ratio [16].

iSet: We set the number λ of candidate entries to 8 and divide the set-id table to $q = 6$ segments to bound the worst-case memory accesses per lookup to 10. The supplement hash table is able to hold 1% of member elements. We compute other system parameters using the procedure in Section V-F except that when determining the values of m, k and s jointly, we try to find the optimal values that minimize the false positive ratio under the constraint of the total available memory M.

B. Memory Accesses per Lookup

Figure 10 compares the three approaches in terms of the number of memory accesses per lookup. The horizontal axis is the available memory in bits per member element encoded. As the bits per member increase from 20 to 50, because the total available memory is fixed, the number of member elements encoded decreases linearly from 800000 to 320000. COMB or Bloomier incurs the same number of memory access per lookup for a member element and a non-member. iSet incurs slightly different memory access overhead for member and non-member. The figure shows that the overhead of COMB is much larger than those of iSet and Bloomier, with the latter having the smallest overhead. More specifically, the number of memory accesses per lookup for COMB ranges from 32 to 96. The number of memory accesses per lookup for iSet ranges from 6.7 to 5.2 for a member element and from 6.2 to 4.4 for a non-member. The number for Bloomier ranges from 2 to 3. However, as we will shown next, the false-positive ratio of Bloomier is the highest.

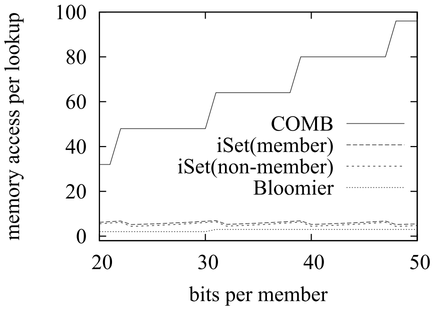


Fig. 10. Average number of memory access per lookup. The horizontal axis is the available memory in bits per member encoded. As the bits per member increase from 20 to 50, the total number of member elements decreases linearly from 800000 to 320000.

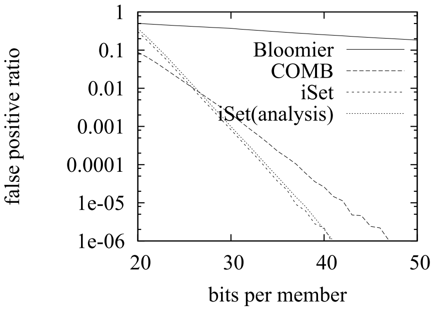


Fig. 11. False positive ratio measured as the fraction of all non-member elements misclassified as members.

With more bits per member, COMB and Bloomier will take advantage of additional space to drive down their false-positive and conflict-classification ratios with more memory accesses to their filters. The design of iSet is completely different. The number of accesses to the index filter is a constant. More memory for the filter would mean its false-positive ratio becomes smaller, which in turn means fewer accesses to the set-id table. However, our parameter setting attempts to minimize the overall false positive ratio (not the number of memory accesses). It may sometimes reduce the index filter and increase the length of the checksum field in the set-id table, causing up-and-down of the iSet curve in Figure 10.

C. False Positive and Conflict Classification

Our analysis shows that iSet takes less memory than others under the same accuracy requirement. Here, if we fix the amount of available memory, it is expected that iSet achieves better accuracy than others. This is confirmed by Figures 11 – 13, where Figure 11 compares the three approaches in terms of the false positive ratio, Figure 12 in terms of the conflict classification ratio, and Figure 13 in terms of the error ratio (the combination of false positive ratio and

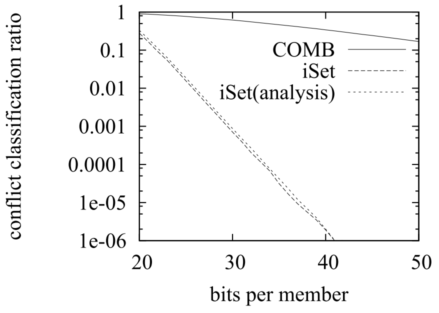


Fig. 12. Conflict classification ratio measured by the fraction of all member elements whose set IDs cannot be uniquely determined. Note that Bloomier does not incur any conflict classification by its design, which however results in high false positive ratio as shown in Figure 11.

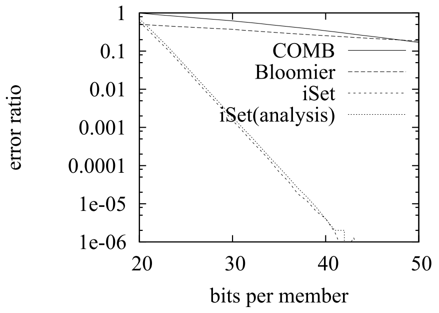


Fig. 13. Error ratio measured summary of false positive ratio and conflict classification ratio.

conflict classification ratio). Notice the log scale on the vertical axis.

From Figure 11, Bloomier has very high false-positive ratio under the tight memory condition in our simulation. COMB has the lowest false positive ratio among the three when the number of bits per member is small, but iSet outperforms COMB when the available memory is more than 26 bits per member. For example, when the memory is 30 bits per member, the false-positive ratio of Bloomier is 3.7×10^{-1}, that of COMB is 2.2×10^{-3}, and that of iSet is 8.2×10^{-4}.

From Figure 12, Bloomier does not incur any conflict classification. The conflict-classification ratio of COMB is very high. It is true that using Hamming can bring down the conflict classification ratio, but that will increase the number of memory accesses, which is already high as Figure 10 has shown. iSet achieves the best balance in its false-positive ratio and conflict-classification ratio. Its conflict classification ratio is slightly smaller than its false positive ratio, as our theoretical analysis has suggested. For numerical examples, when the memory is 30 bits per member, the conflict-classification of Bloomier is 0, that of COMB is 6.12×10^{-1}, and that of iSet is 7.1×10^{-4}.

In Figure 11-12, we also plot the computed false-positive ratio and conflict-classification ratio of iSet based on the analytical formulas in Section V under the same parameter setting. They match very well with the simulation results.

Overall from Figure 13, the error ratio of iSet is much smaller than those of the COMB and Bloomier.

Using Fig. 11, we can also compare different approaches in terms of memory cost under the same false positive ratio.

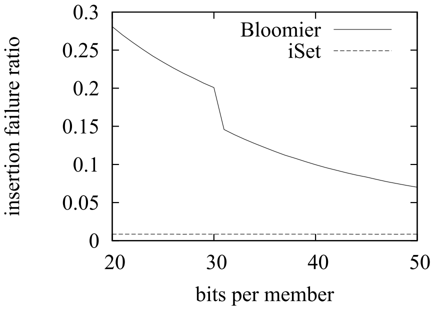


Fig. 14. Insertion-failure ratios of Bloomier and iSet. COMB does not have insertion failure by its design, which however results in high conflict classification ratio in tight memory as shown in Figure 12.

More specifically, given any false positive ratio, we can find from the figure the memory cost for iSet or COMB to achieve this ratio. According to the figure, when the false positive ratio is larger than 0.01, iSet incurs smaller memory cost than COMB. When the false positive ratio is smaller than 0.01, COMB incurs smaller memory cost. However, in the region where COMB outperforms iSet in Fig. 11, it performs much worse on conflict classification than iSet in Fig. 12. When we consider both false positive and conflict classification, iSet consistently outperforms COMB, as Fig. 13 shows.

D. Insertion Failure

After inserting $n = 320000 \sim 800000$ elements, we measure the insertion-failure ratios of the three approaches. The results are shown in Figure 14. The design of COMB does not incur any insertion failure. The insertion-failure ratio of iSet is less than 1%. That translates into 8000 elements in maximum stored outside of the set-id table. Bloomier has very high insertion-failure ratio, which is consistent with the results in Tables I-II, where Bloomier requires the most number of bits for encoding each member element and therefore when the available memory is too tight, a significant number of members will not be accommodated in its filter. For numerical examples, when the memory is 30 bits per member, the insertion-failure rate of Bloomier is 2.0×10^{-1}, that of iSet is 8.6×10^{-3}, and that of COMB is 0.

E. Hash Complexity

We compare the hash complexity of the three approaches, which is an indication of computational cost in practical scenarios. The approaches require multiple hash functions. We can logically treat all hash functions as a single one that produce a hash output stream with a sufficient number of bits for mapping, indexing, and generating checksum purposes [30]. For example, suppose we need to map e to 10 bits in a filter whose size is 2^{20} and each hash computation gives 128-bit output. We may compute 10 hash functions and take 20 bits of each output for indexing one bit in the filter, or make two hash computations for 256-bit output stream (possibly by feeding the output of the first hash computation as the input of the second) and use the first 200 bits from the stream to index the 10 bits in the filter. Extensive work in [30] has demonstrated that the second approach works very well.

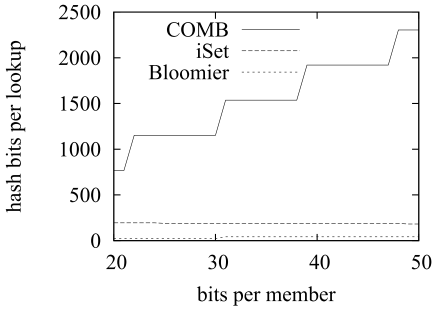


Fig. 15. Number of hash bits required for each membership lookup by COMB, Bloomier, or iSet.

Therefore, instead of comparing the number of hash functions each approach uses, we compare the total number of hash bits each approach needs for a lookup. It is easy to derive the number of hash bits needed:

COMB: $f \times k \times \log_2 M$

Bloomier: $k \times \log_2 l$

iSet: $\lambda \times k \times \log_2 64 + \lambda \times \log_2(l/q) + s$, where $\lambda \times k \times \log_2 64$ bits are used for the index filter (assuming the processor fetches 64 bits at a time), $\lambda \times \log_2(l/q)$ bits are used to access the set-id table, and s bits are used as checksum.

Figure 15 shows the numerical results from our simulations: COMB requires 768–2304 hash bits per lookup, which translates to 6–18 hash computations if each hash computation generates 128 hash bits; iSet requires 185–224 hash bits per lookup, which translates to 2 hash computations; Bloomier requires 21–42 hash bits per lookup, which translates to 1 hash computation. Bloomier is clearly the most light-weighted approach, but iSet also performs well, given that its error ratio is much smaller than the other two approaches.

VII. Conclusion

In this paper, we propose a novel design for multi-set membership lookup. Through detailed analysis, we point out that using Bloom filters to store set IDs is inefficient in both memory space and lookup overhead. Our strategy is to separate membership encoding and ID storage in two separate data structures, called index filter and set-id table, respectively. The former is a two-level block Bloom filter, and the latter is a multi-hashing table. We show that uneven candidate-entry distribution for the set-id table can drastically reduce insertion failure and thus reduce memory consumption by fully utilizing the space in the table. Analysis and simulation show that this new design significantly outperforms the existing work under tight memory conditions. It achieves balanced performance in terms of number of memory accesses per lookup, false-positive ratio, conflict-classification ratio, and number of hash bits required per lookup.

References

[1] H. Song, F. Hao, M. Kodialam, and T. V. Lakshman, "IPv6 lookups using distributed and load balanced Bloom filters for 100 Gbps core router line cards," in *Proc. IEEE INFOCOM*, Apr. 2009, pp. 2518–2526.

[2] J. Brodkin, "A wireless router that tracks user activity—But for a good reason," *Ars Technica.*, Jan. 2013. [Online]. Available: http://arstechnica.com/gadgets/2013/01/a-wireless-router-that-tracks-user-activity-but-for-a-good-reason/.

[3] Y. Lu and B. Prabhakar, "Robust counting via counter braids: An error-resilient network measurement architecture," in *Proc. IEEE INFOCOM*, Apr. 2009, pp. 522–530.

[4] S. Dharmapurikar, P. Krishnamurthy, and D. E. Taylor, "Longest prefix matching using Bloom filters," *IEEE/ACM Trans. Netw.*, vol. 14, no. 2, pp. 397–409, Apr. 2006.

[5] H. Song, S. Dharmapurikar, J. Turner, and J. Lockwood, "Fast hash table lookup using extended Bloom filter: An aid to network processing," *ACM SIGCOMM Comput. Commun. Rev.*, vol. 35, no. 4, pp. 181–192, 2005.

[6] X. Tian and Y. Cheng, "Bloom filter-based scalable multicast: Methodology, design and application," *IEEE Netw.*, vol. 27, no. 6, pp. 89–94, Nov./Dec. 2013.

[7] B. H. Bloom, "Space/time trade-offs in hash coding with allowable errors," *Commun. ACM*, vol. 13, no. 7, pp. 422–426, Jul. 1970.

[8] E. Horowitz, S. Sahni, and S. Rajasekaran, *Computer Algorithms C++*. San Francisco, CA, USA: Freeman, 1996, ch. 3.2.

[9] E. Fredkin, "Trie memory," *Commun. ACM*, vol. 3, no. 9, pp. 490–499, 1960.

[10] M. Dietzfelbinger *et al.*, "Dynamic perfect hashing: Upper and lower bounds," *SIAM J. Comput.*, vol. 23, no. 4, pp. 738–761, 1994.

[11] Y. Lu, B. Prabhakar, and F. Bonomi, "Bloom filters: Design innovations and novel applications," in *Proc. 43rd Annu. Allerton Conf.*, 2005, pp. 1006–1015.

[12] F. Hao, M. S. Kodialam, T. V. Lakshman, and H. Song, "Fast dynamic multiple-set membership testing using combinatorial Bloom filters," *IEEE/ACM Trans. Netw.*, vol. 20, no. 1, pp. 295–304, Feb. 2012.

[13] Y. Lu, A. Montanari, B. Prabhakar, S. Dharmapurikar, and A. Kabbani, "Counter braids: A novel counter architecture for per-flow measurement," in *Proc. ACM SIGMETRICS*, Jun. 2008, pp. 121–132.

[14] Q. Zhao, J. Xu, and A. Kumar, "Detection of super sources and destinations in high-speed networks: Algorithms, analysis and evaluation," *IEEE J. Sel. Areas Commun.*, vol. 24, no. 10, pp. 1840–1852, Oct. 2006.

[15] C. Hermsmeyer, H. Song, R. Schlenk, R. Gemelli, and S. Bunse, "Towards 100G packet processing: Challenges and technologies," *Bell Labs Tech. J.*, vol. 14, no. 2, pp. 57–79, 2009.

[16] A. Z. Broder and M. Mitzenmacher, "Network applications of Bloom filters: A survey," *Internet Math.*, vol. 1, no. 4, pp. 485–509, 2004.

[17] F. Chang, W.-C. Feng, and K. Li, "Approximate caches for packet classification," in *Proc. IEEE INFOCOM*, Mar. 2004, pp. 2196–2207.

[18] B. Chazelle, J. Kilian, R. Rubinfeld, and A. Tal, "The Bloomier filter: An efficient data structure for static support lookup tables," in *Proc. ACM-SIAM SODA*, 2004, pp. 30–39.

[19] A. Z. Broder and A. R. Karlin, "Multilevel adaptive hashing," in *Proc. ACM-SIAM Symp. Discrete Algorithms (SODA)*, 1990, pp. 43–53.

[20] Y. Kanizo, D. Hay, and I. Keslassy, "Optimal fast hashing," in *Proc. IEEE INFOCOM*, Apr. 2009, pp. 2500–2508.

[21] F. Bonomi, M. Mitzenmacher, R. Panigrah, S. Singh, and G. Varghese, "Beyond Bloom filters: From approximate membership checks to approximate state machines," *ACM SIGCOMM Comput. Commun. Rev.*, vol. 36, no. 4, pp. 315–326, 2006.

[22] A. Kirsch and M. Mitzenmacher, "Simple summaries for hashing with choices," *IEEE/ACM Trans. Netw.*, vol. 16, no. 1, pp. 218–231, Feb. 2008.

[23] A. Z. Broder and M. Mitzenmacher, "Using multiple hash functions to improve IP lookups," in *Proc. IEEE INFOCOM*, Apr. 2001, pp. 1454–1463.

[24] H. Noda *et al.*, "A cost-efficient high-performance dynamic TCAM with pipelined hierarchical searching and shift redundancy architecture," *IEEE J. Solid-State Circuits*, vol. 40, no. 1, pp. 245–253, Jan. 2005.

[25] F. Putze, P. Sanders, and J. Singler, "Cache-, hash-, and space-efficient Bloom filters," *J. Experim. Algorithmics*, vol. 14, Dec. 2009, Art. no. 4.

[26] Y. Qiao, T. Li, and S. Chen, "Fast Bloom filters and their generalization," *IEEE Trans. Parallel Distrib. Syst.*, vol. 25, no. 1, pp. 93–103, Jan. 2014.

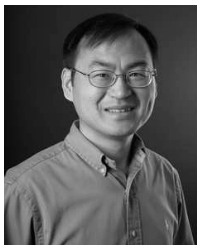
[27] L. Fan, P. Cao, J. Almeida, and A. Z. Broder, "Summary cache: A scalable wide-area Web cache sharing protocol," *IEEE/ACM Trans. Netw.*, vol. 8, no. 3, pp. 281–293, Jun. 2000.

[28] O. Rottenstreich, Y. Kanizo, and I. Keslassy, "The variable-increment counting Bloom filter," in *Proc. IEEE INFOCOM*, Mar. 2012, pp. 1880–1888.

[29] H. Wang, H. Zhao, B. Lin, and J. Xu, "DRAM-based statistics counter array architecture with performance guarantee," *IEEE/ACM Trans. Netw.*, vol. 20, no. 4, pp. 1040–1053, Aug. 2012.

[30] Y. Qiao, T. Li, and S. Chen, "One memory access Bloom filters and their generalization," in *Proc. IEEE INFOCOM*, Apr. 2011, pp. 1745–1753.

Yan Qiao received the B.S. degree in computer science and technology from Shanghai Jiao Tong University, China, in 2009, and the Ph.D. degree from the University of Florida in 2014. Her advisor is Dr. S. Chen. Her research interests include network measurement, algorithms, and RFID protocols.



Shigang Chen received the B.S. degree from the University of Science and Technology of China in 1993, and the M.S. and Ph.D. degrees from the University of Illinois at Urbana–Champaign in 1996 and 1999, respectively, all in computer science. He spent three years with Cisco Systems. He joined the University of Florida in 2002, where he is currently a Professor with the Department of Computer and Information Science and Engineering. His research interests include computer networks, Internet security, wireless communications, and distributed computing.

Zhen Mo received the B.E. degree in information security engineering and the M.E. degree in theory and new technology of electrical engineering from Shanghai Jiao Tong University in 2007 and 2010, respectively. He is currently pursuing the Ph.D. degree with the Department of Computer and Information Science Engineering, University of Florida. His research interests include cyber security and cloud computing security.

Myungkeun Yoon received the B.S. and M.S. degrees in computer science from Yonsei University, South Korea, in 1996 and 1998, respectively, and the Ph.D. degree in computer engineering from the University of Florida in 2008. He was with the Korea Financial Telecommunications and Clearings Institute from 1998 to 2010. He is currently an Assistant Professor with the Department of Computer Engineering, Kookmin University, South Korea. His research interests include computer and network security, network algorithm, and mobile network.