A Survey of Hash Tables with Summaries for IP Lookup Applications

Thomas Zink

University of Konstanz, Germany,
thomas.zink@uni-konstanz.de

Abstract. Efficient IPv6 packet forwarding is still a major bottleneck in todays networks. Especially in the internet core we are faced with very large routing tables and a high number of high-speed links. In addition economical restrains exist in terms of manufacturing and operation costs. So demand is high for efficient IPv6 packet forwarding mechanisms. In the last few years a lot of work has been done on hash tables and summaries that allow compact representations and constant lookup time. The features sound attractive for IPv6 routing, thus a survey and evaluation of these data structures seems appropriate.

1 Introduction

The standard networking protocol used today is the Internet Protocol version 4 (IPv4). However, its address space is too small to serve the highly increased number of hosts. IPv6 promises to solve that problem by providing a virtually unlimited (in the sense of not to be expected to ever get exhausted) number of addresses. But efficient forwarding of IPv6 packets still is a major problem. That holds especially in the internet core where the routing tables contain millions of entries and packets arrive on a thousand high-speed links. Here, finding the correct route is an extensive task that requires specialized hardware, efficient algorithms and optimized data structures to be able to process the packets at line speed.

The main bottleneck is the number of memory accesses needed to successfully lookup the prefix. This number depends on the *longest prefix matching* (LPM) technique and the underlying hash table. Summaries which are kept in fast but expensive on-chip memory can be used to improve the lookup process. This document concentrates on hash tables and their summaries suitable for IP Lookup applications. We are especially interested in the number of bits per item and the total size needed for the summary representation, since the amount of on-chip memory is quite limited due to its high cost.

2 Hashing with multiple Choices

The naïve hash table can be seen as an array of linked lists. Each item to be inserted is hashed to find a bucket in the hash table and is appended to the

list of items in this bucket. The problem in this scheme is that lookup requires following multiple next pointers (and thus multiple memory accesses) to find the correct item. Furthermore, depending on the number of items and the size of the table the load of buckets can get quite high. To reduce the maximum load, Broder and Mitzenmacher [1] suggest using multiple hash functions. The n buckets of the hash table are split into d equal parts imagined to run from left to right. An item is hashed d times to find d possible locations. It is then placed in the least loaded bucket, ties are broken by going left. A lookup now requires examining the d locations. However, since the d choices are independent, the lookup can be done in parallel or pipelined. Interestingly, breaking ties by going left improves the load of the buckets as first shown by Vöking [2].

3 Bloom Filter Summaries

3.1 Bloom Filter

Bloom filters, first introduced by Burton H. Bloom [3], are used to represent set memberships of a set S from a universe U. They allow false positives, that is, they can falsely report the membership of an item not in the set, but never false negatives. Basically a *Bloom filter* is a bit array of arbitrary length m where each bit is initially cleared. For each item x inserted into the set k hash values $h_0, \ldots, h_{k-1}$ are produced while $\forall h \in \mathbb{N} : 0 \leq h < m$. The bits at the corresponding positions are then set. When querying for an item y, the k bits y hashes to are checked. If all of them are set y is reported to be a member of S. If at least one of the bits is clear y is not present in the set. A false positive occurs, if all bits corresponding to an item not in the set are 1. The probability that this happens depends on the number of items n inserted, the array length m and the number of hash functions k used. It is given as

$$\epsilon = (1 - (1 - \frac{1}{m})^{kn})^k.$$

It can be proven, that for given n and m the optimal number of hash functions is

$$k = \frac{m}{n} ln2.$$

To minimize the false positive probability m must be chosen appropriately large. To keep it constant m must grow linearly with n. Generally

$$m = \zeta * n$$

for sufficiently large constant ζ. The number of bits per item β_{bf} is equal to ζ, so the total storage requirement is equal to m.

3.2 Counting Bloom Filter

Standard Bloom filters cannot handle deletions. Since one cannot know how many items hash to specific locations in the filter the bits cannot be cleared

upon removal of an item. Thus when items are deleted the filter must be completely rebuilt. Fan et al [4] address this issue by introducing a *Counting Bloom Filter* (CBF). Instead of a bit array, the CBF maintains an array of counters $C = \{c_0, \ldots, c_{m-1}\}$ to represent the number of items that hash to its cells. Insertions and deletions can now be handled easily by incrementing and decrementing the corresponding counters. The counters are typically three to four bits wide, so the CBF needs about three to four times the space of a standard Bloom filter. Using a small fixed amount of bits to represent the counters introduces the problem of a possible counter overflow. If more items hash to a counter than it can represent an overflow occurs. Thus the counter-width must be chosen appropriately large for a given application. In general the counter-width is derived from the expected maximum counter value $max(C)$, which is equal to the expected maximum number of collisions per counter and can be easily computed using probabilistic methods. There are multiple approaches for dealing with overflows. One is to simply ignore counters that have reached their maximum value and stop updating them. Though this is a simple solution it leads to inaccuracies in the filter that must somehow be resolved. Another solution is to keep the exact counter value in additional dedicated memory and check this area whenever a counter at the maximum value is seen. If it is an overflown counter, only the value in the extra memory is updated and evaluated. The number of bits per item β is

$$\beta_{cbf} = log\ max(C) * \zeta.$$

3.3 Multilayer Compressed Counting Bloom Filters

Recently, Ficara et al [5] introduced a compression scheme for counting Bloom filters that also allows updates which they name ML-CCBF (MultiLayer Compressed Counting Bloom Filter). It is based on Huffman coding. They use a simple code, where the number of 1s denote the value of the counter. Each string is terminated by 0. So the number of bits needed to encode a counter value φ is $\varphi + 1$ except for the value 0. Since with an optimal Bloom filter configuration the probability of increasing counters falls exponentially this poses an optimal encoding. Increasing or decreasing a counter is also simple by just adding or removing 1. To avoid indexing and memory alignment issues the counters are not stored consecutively but each one is distributed over multiple layers of bitmaps $L_0, \ldots, L_N$, with N dynamically changing on demand. Thus L_i holds the ith bit of the string. The first layer is a standard bloom filter representing items with $\varphi \geq 1$. To index the bitmaps $k + N$ hash functions are needed. The k hash functions are random hash functions used for the Bloom filter. The other N hash functions index the bitmaps $L_1, \ldots, L_N$ and must be perfect to prevent collisions. To retrieve a counter value it's position u_0 in L_0 is first determined. If the bit at u_0 is 0 then $\varphi = 0$. Else L_1 must be examined. Let $popcount(u_i)$ be the number of ones in bitmap i before index u. $popcount(u_i)$ is then hashed using the perfect hash function H_{k+i} to find the index in L_{i+1}. If this bit is set, 1 is added to the current counter value and the next bitmap must be examined. Otherwise, the end of the code is reached. Note that N must be as large as

the maximum counter value + 1. With increasing counter values new layers of bitmaps can simply be added. This scheme provides near optimal counter storage. However, while it is easy to check the membership of an item by probing the Bloom filter L_0 retrieving all counters for an item is very expensive due to the need of computing $popcount(u_i)$ for all k counters.

4 Fast Hash Tables

The major problem with data structures with d choices is that they require at least d lookups in the hash table. Though these are usually independent and can be done in parallel if the structure is built accordingly, it is inefficient to do so. In [6] Song et al present a data structure named *Fast Hash Table* (FHT) that eliminates the need for parallel lookups by using *counting Bloom filter* summaries. In their scheme always only one bucket access is needed. They use a CBF with $m = 2^{\lceil log\ 12.8*n \rceil}$ counters each representing the number of items hashing to the corresponding bucket in the hash table. Thus the $k = \frac{m}{n} ln2$ hash functions are used to index both the CBF and the hash table.When searching for an item x it is hashed to find its k counters. The minimum z of these counters is computed. If $z = 0$ the item is not present in the hash table, else it is retrieved from the most left bucket corresponding to z. Note, that while there is only one access to a bucket, it may be necessary to follow next pointers to traverse the list of items in one bucket. Insertion and deletion of items depend on the type of FHT of which there are three. This setting requires

$$\beta_{fht} = \frac{m}{n}\beta_{cbf}$$

bits per item in the summary.

4.1 Basic Fast Hash Table.

In the Basic FHT (BFHT) items are simply inserted k times, once in every location it hashes to. The corresponding counters are incremented. In this case the counter value also equals the actual load of a bucket. Due to collisions it is possible that an item is inserted less than k times. In this case the counter experiencing the collision is incremented only once. Deletions are equally simple. The item is removed from the buckets and the counters decremented. Though simple, this scheme leads to heavily loaded buckets and thus retrieval of an item is most certainly accompanied by following multiple pointers.

4.2 Pruned Fast Hash Table.

The Pruned FHT (PFHT) is an improvement on the BFHT by only storing the item at the most left bucket with minimum counter value. Counters are handled as in the BFHT. This improves bucket load and lookup time, but insertions and deletions are more complicated. Insertion and deletion can influence the counters

of already present items. Since an item is retrieved by examining its counters and lookup the bucket which has the lowest counter value, this counter value might have been changed leading to lookup in the wrong bucket. For insertions the items present in affected buckets must be considered for relocation. Deletions require even more effort. Decrementing a counter may result in this counter being the smallest one for items hashing to it. But since a bucket does not store all items hashing to it, it is not possible to find the items that have to be relocated. This can either be achieved by examining the whole PFHT and check every item (which obviously is very expensive), or by keeping an offline BFHT which allows examining all the items hashing to a bucket. Thus the PFHT is only suitable for applications where updates are much rarer than queries.

4.3 Shared-node Fast Hash Table.

The Shared-node FHT (SFHT) provides support for update critical application at the cost of slightly higher memory consumption than the BFHT. Here the buckets only store a pointer to the first item that has been inserted. The items themselves are stored in extra memory and carry a next pointer to the next item in the list. Special care must be taken, when an item is inserted, that hashes to empty as well as filled buckets. Appending this item to the linked lists would lead to inconsistencies, so it must be replicated and pointers set in the empty buckets and the linked lists accordingly. Counters are again treated as in the BFHT. Though updates are much easier compared to the BFHT lookup now requires following at least one pointer, resulting in more memory accesses.

5 Simple Summaries for Multilevel Hash Tables

The Fast Hash Table uses one counter per bucket to keep track of the number of items stored. While this is a straightforward approach which is also easy to implement, it has rather high memory requirements for the counting Bloom filter summary which has to grow with bigger table sizes. Kirsch and Mitzenmacher [7] observe, that the summary structure need not correspond to a bucket in the underlying data structure. This allows separation of the hash table and its summary and independent optimization. They use a Multilevel Hash Table (MHT), first introduced by Broder and Karlin [8], to store the items. The MHT consists of $d = log\ log\ n + 1$ sub-tables where each sub-table T_i has $c_2^{i-1}c_2n$ single item buckets with $c_1 > 1$ and $c_2 < 1$. Thus $|T_i|$ is decreasing for increasing i. An occupancy bitmap is kept in on-chip memory with one bit per bucket, allowing efficient queries for available empty buckets. When an item is inserted, it is hashed d times to find one possible bucket in each sub-table. The item is put in T_i with the lowest i for which the bucket is empty. A crisis can occur when all d buckets are occupied. However, it can be proven that for any $c_1c_2 > 1$ the crisis probability is insignificantly small. Kirsch and Mitzenmacher present three summary data structures which will now be reviewed.

5.1 Interpolation Search Summary

All items inserted are hashed to a b-bit string, where b must be uniformly distributed and sufficiently large. The index i of T_i where the item is placed is stored along with its string b. Interpolation search is used to search for an item, which requires the array of strings to be ordered. Insertions and deletions correspond to addition and removal of the string and index and requires shifting subsequent strings to keep the ordering. A failure can occur if two items inserted hash to the same b-bit string. The failure probability is

$$p_{fail}(n, b) = 1 - \prod_{k=0}^{n+1} \frac{2^b - k + 1}{2^b}$$

A false positive occurs when an item not inserted hashes to a string present in the summary. Supposed no failure occurred the false positive probability is

$$p_{fp}(n, b) = \frac{n}{2^b}.$$

Thus by choosing b appropriately large for given n both the failure and false positive probability can be optimized. The authors suggest $b = 61$ for $n = 100.000$. Note, that b must grow with larger n to keep the probabilities constant. $log\ d$ bits are additionally needed to represent i. With $d = 8$ the total number of bits per item needed for the summary is 64 and is derived by

$$\beta_{is} = b + log\ d.$$

5.2 Single Bloom Filter

The *Single Bloom Filter* summary (SF) has $m = n\ log\ n$ cells initialized to 0 and represents the type t of an item where t is the sub-table the item is stored in. $k = log\ n$ hash functions are used to access the Bloom filter. To insert an item first its type is identified by inserting it into the MHT. Then it is hashed k times and the k cell values are replaced with the maximum of its value and the type of the item. To search for an item the k cells are examined and the minimum z is computed. If $z = 0$ the item is not present in the MHT. Otherwise, it has a type of at most z. In addition to false positives this structure can also return type failures, iff z yields an incorrect type for an item. With $d = log\ log\ n + 1$ types the Bloom filter summary needs

$$\beta_{sf} = log\ n\ log\ log\ log\ n$$

bits per item.

5.3 Multiple Bloom Filters

The single filter approach introduces type failures and care must be taken during construction since false positives and type failures are not independent. The *Multiple Bloom Filter* summary (MBF) eliminates this additional effort by making

use of the skew of the items in the MHT. Since the number of items in subsequent sub-tables decreases rapidly an array of Bloom filters $B = \{B_0, ..., B_{t-1}\}$ with decreasing size can easily be used to represent the set of items of a specific type. Each filter B_i represents the set of items with type at least $i + 1$. Thus a false positive on B_i is equal to a type i failure. Obviously the false positive probability must be extremely small for successful lookup. This leads to the need of significantly more hashing. The authors give examples of 15 hash functions for B_0 and 49 for each of the other filters. However, the hash functions between Bloom filters need not to be independent, so the same set of hash functions can be used for each filter and the result modulo the size of each filter used for access. With a well designed MHT the total number of bits for the MBF is $n\ log\ n$, leading to

$$\beta_{mbf} = log\ n.$$

5.4 Deletions

The Bloom filter based summaries only support inserts. To allow deletions significantly more effort is needed in terms of additional data structures. Two deletion schemes are proposed in [7], the *lazy deletions* and the *counter based deletions*

Lazy Deletions. A simple approach for adding deletion support is *lazy deletions*. A deletion bit array is kept with one bit for every bucket in the MHT. When an item is deleted the corresponding bit is simply set to 1. During lookup, items in buckets with set deletion bit are simply ignored. However, problems can occur if too many newly inserted items collide with deleted buckets. Thus after a certain threshold the whole MHT must be rebuilt, that is, all items must be examined for relocation.

Counter Based Deletions. As with counting Bloom filters this scheme adds counters to the Bloom filter based summaries to keep track of the number of items inserted. The single filter summary must now contain one counter for each possible type in each of its cells. In the multiple Bloom filter summary the Bloom filters are replaced by counting Bloom filters. Since the number of items decreases throughout the sub-tables the counter-width can also decrease. No evaluation is given by the authors for the modified single filter summary but given d choices it would require

$$\beta_{sfm} = dc \log n$$

bits per item. Generalizing the amount of bits needed for the modified multiple Bloom filter summary is not as straightforward since the choice of how many bits per counter and filter should be used depends on the type of application and also personal taste. However, the authors give some examples and state that the modified version occupies 3.3 times more space than the simple multiple Bloom filter summary. This leads to

$$\beta_{mbfm} = 3.3log\ n$$

bits per item.

6 Discussion

Though the provided data structures show a lot of improvements over naïve hash tables and sound appealing for IPv6 lookup applications, their usability in the internet core is quite limited. The reasons are 1) missing evaluation for millions of entries, 2) need for pointer following and 3) high requirements of fast on-chip memory.

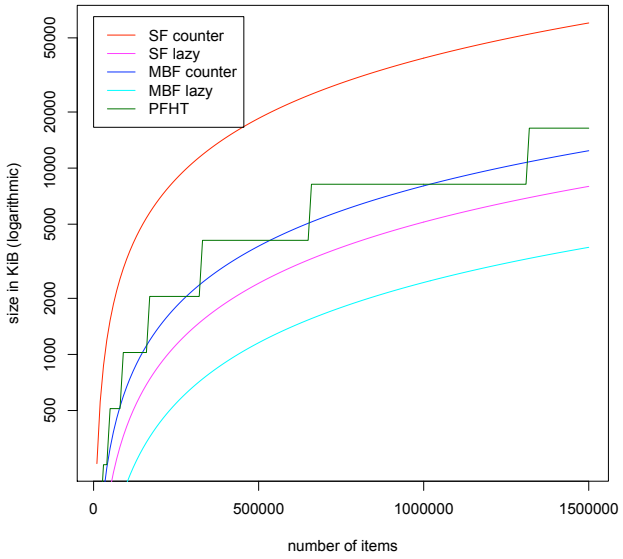
All the Fast Hash Table types need following next pointers during lookup. Only the Pruned FHT seems appealing, since bucket loads are lower than in its sister structures. However, the number of bits per item needed for the summary is quite high. With millions of entries in the lookup table the expected summary size exceeds the available lookup-dedicated on-chip memory by far. Compressing the counting Bloom filter summary by using a ML-CCBF is not an option because of its need for multiple perfect hash functions and the expensive counter computation. The simple summaries for MHTs provide a constant lookup of $O(1)$, which is the optimum. However, the Interpolation Search (IS) summary has high memory requirements, stated 64 bits for $n = 100,000$ and expected to be much higher for millions of entries. In addition, interpolation search is an expensive task and at the time of this writing cannot be done in hardware. The Single Filter (SF) summary needs less space but does not support deletions. No evaluation is given for deletion support, but can easily be computed. The smallest summary is the multiple Bloom filter summary (MBF), but it has similar constraints as the SF regarding deletions. In addition to the summaries, the occupancy and optional deletion bitmaps are also kept in on-chip memory for efficiency reasons, which further increases the the needed amount of memory. However, this is not a requirement for the functionality of the summaries.

Structure	Bits per item	total size in KiB
PFHT	83.89	1,024
IS	64	781.25
SF lazy	33,54	410.43
SF counter	265,75	3,244.07
MBF lazy	16.61	202.75
MBF counter	54.81	699.09

Table 1. Number of bits for n = 100,000

Table 1 summarizes the expected number of bits per item and total size in KiB needed for $n = 100,000$. For the MHT summaries the occupancy and deletion bitmaps are neglected. We assume a counter-width of 4 for the PFHT and 2 for the counter-based SF. Figure 1 shows a size comparison for $n \leq 1.5 \cdot 10^6$. Since no information is given on how to appropriately choose b, the interpolation search summary is neglected in this figure.

Fig. 1. Total summary sizes



7 Conclusion

IP lookup, as the name implies, is a heavily lookup driven application where updates occur rarely and need not take effect instantly. Under these conditions the lookup engine can be separated from the update engine. Applying this observation to the available MHT summaries allows us to resign the occupancy bitmap completely and check every bucket during inserts. With deletions the lazy deletion scheme suffices. Thus, the lazy multiple Bloom filter summary performs best in terms of memory requirement and false-positive probability, if only the filters are kept in on-chip memory. However, it requires significantly more hashing than other structures. Though the same set of hash functions can be used for all the filters, computing modulo is pretty expensive and should be avoided. Apart from this, the MHT summaries do not seem to leave room for further improvements. The PFHT can be further tuned to better suit IP lookup applications. First, the counting Bloom filter could be compressed using easily implementable encoding schemes like Huffman encoding. The counters could be encoded sequentially in on-chip memory and the code-book stored in registers. This allows easy encoding/decoding while still allowing updates. However, to allow direct indexing, a fixed number of counters must be encoded per memory word. To prevent overflows, the word cannot be completely filled which limits the achievable compression factor. The length of the CBF and the number of buckets is optimized for a vanishing false positive probability. By reducing the length m

at the cost of a higher false positive rate the filter size can also be significantly reduced. This is not an option for the bloom filter based MHT summaries since here the false positive probability is coupled to the type failure probability which must be kept extremely small. Optimizing the PFHT is subject of [9] and can not be reviewed here in detail. However, reducing the memory requirements of the PFHT by supporting a constant lookup time of $O(1)$ can shift the tuned PFHT in favor over the MBF summary.

References

1. Andrei Broder and Michael Mitzenmacher. Using multiple hash functions to improve IP lookups. In *IEEE INFOCOM*, 2001. 2
2. Berthold Vöcking. How asymmetry helps load balancing. *J. ACM*, 50(4):568–589, 2003. 2
3. Burton H. Bloom. Space/time trade-offs in hash coding with allowable errors. *Commun. ACM*, 13(7):422–426, 1970. 3.1
4. Li Fan, Pei Cao, Jussara Almeida, and Andrei Broder. Summary cache: A scalable wide-area web cache sharing protocol. In *Proceedings of ACM SIGCOMM*, pages 254–265, September 1998. 3.2
5. D. Ficara, S. Giordano, G. Procissi, and F. Vitucci. Multilayer compressed counting bloom filters. *INFOCOM 2008. The 27th Conference on Computer Communications. IEEE*, pages 311–315, April 2008. 3.3
6. Haoyu Song, Sarang Dharmapurikar, Jonathan Turner, and John Lockwood. Fast hash table lookup using extended Bloom filter: An aid to network processing. In *SIGCOMM '05*, pages 181–192, New York, NY, USA, 2005. ACM Press. 4
7. Adam Kirsch and Michael Mitzenmacher. Simple summaries for hashing with choices. *IEEE/ACM Trans. Netw.*, 16(1):218–231, 2008. 5, 5.4
8. Andrei Z. Broder and Anna R. Karlin. Multilevel adaptive hashing. In *SODA '90: Proceedings of the first annual ACM-SIAM symposium on Discrete algorithms*, pages 43–53, Philadelphia, PA, USA, 1990. Society for Industrial and Applied Mathematics. 5
9. Thomas Zink. Packet forwarding using improved bloom filters. Master's thesis, University of Konstanz, 2009. 7