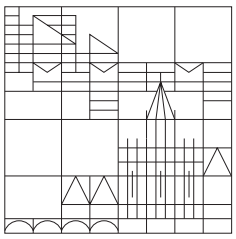
Packet forwarding using improved Bloom filters

Thomas Zink
thomas.zink@uni-konstanz.de

A Master Thesis submitted to the
Department of Computer and Information Science
University of Konstanz
in fulfillment of the
requirements for the degree of
MASTER OF SCIENCE
February 2009



Author:
Thomas Zink, 01/640539
Master candidate enrolled in master studies Information Engineering
Major Subject: Computer Science

Assessors:
Prof. Dr. Marcel Waldvogel
Distributed Systems Group
http://www.inf.uni-konstanz.de/disy/members/waldvogel/

Prof. Dr. Marc H. Scholl
Database & Information Systems Group
http://www.inf.uni-konstanz.de/∼scholl

Supervisors:
Prof. Dr. Marcel Waldvogel
Distributed Systems Group
http://www.inf.uni-konstanz.de/disy/members/waldvogel/

Konstanzer Online-Publikations-System (KOPS)
URN: http://nbn-resolving.de/urn:nbn:de:bsz:352-opus-131409
URL: http://kops.ub.uni-konstanz.de/volltexte/2011/13140/

Dedicated to my son Elaya Valerian, whose smile always inspires me.

Abstract. Efficient IPv6 packet forwarding still is a major bottleneck in todays networks. Especially in the internet core we face very large routing tables and a high number of high-speed links. In addition, economical restrictions exist in terms of manufacturing and operation costs of routers. Resources are limited and network providers reluctantly change their infrastructure.

On the other hand the number of internet hosts keeps exploding. Not only PCs and mobile computers, but all kinds of mobile devices want to connect to the internet. With low-cost end-user flatrates the number of households connecting to the net also increases. Economically emerging countries do their share. Unbalanced distribution of IPv4 addresses leads to local service shortages. The address space IPv4 provides is close to getting exhausted.

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, however, no evaluation exists for tables with millions of entries and no applications are known to make use of the proposed data structures. Furthermore, the structures are usually designed to fit generic applications. A survey and evaluation considering applicability in IPv6 routing seems appropriate. In addition we will explore new ways of exploiting the harsh conditions prevailing in the internet core to design a deployable data structure specialized for IPv6 lookup applications. Our design achieves an improvement in size by the factor of 10. Moreover, it is easily adjustable to fit different cost functions and the behavior is highly predictable.

Table of Contents

List of Figures

List of Tables

1 Introduction

In 2007, sales for mobile microprocessors first surpassed that of desktop processors. Since then the trend tends towards higher mobility in computing. This is accompanied by the demand of an everywhere internet. Today, not only PCs but mobile phones, mp3 and multimedia players want to connect to the world wide web. In addition, flourishing economy in newly industrializing countries like China and India further leads to an explosion of the number of connected hosts.

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, proposed in 1998 by the Internet Engineering Task Force, 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 is still a major problem. That holds especially in the internet core where the routing tables contain millions of entries, and packets arrive on thousand high-speed links. Identifying the correct route in a 128 bit address space is an extensive task that requires specialized hardware, efficient algorithms and optimized data structures to be able to process the packets at line speed. Satisfying these conditions is accompanied by incredibly high production costs.

1.1 Longest Prefix Matching

To forward a packet a router must search its forwarding table for the longest entry that matches the destination of the packet. This extensive task is known as *longest prefix matching* (LPM) and is also referred to as IP-lookup. It requires the router to search multiple prefixes of variable length. Common approaches for longest prefix matching include algorithms and content addressable memory (CAM), which can be queried for content and returns the addresses of its location. CAM is expensive in terms of chip size and operation. Thus, it is desirable to exchange it with algorithmic approaches. Here, the main bottleneck is the number of memory accesses needed to retrieve the prefixes. These algorithms require the prefixes to be stored in data structures that allow lookup with a maximum of one memory access to prevent the overall process to degenerate. In this thesis we concentrate on improving these data structures in terms of hardware requirements to reduce router production and operation costs. LPM algorithms are shortly reviewed in 2.1.

1.2 Hash Tables and Summaries for IP-Lookup

A natural approach is to store prefixes in hash tables that allow identification of the correct bucket in one memory access. However, due to hash collisions one can not guarantee that the correct entry can be retrieved immediately. Hashing with multiple choices can improve the lookup by allowing an item to be stored in multiple locations. While this helps to improve the distribution of items it

requires parallel lookup of all possible locations. To eliminate the need for parallelism and improve lookups, small hash table summaries - usually based on Bloom filters - can be kept in very fast but expensive on-chip memory. A variety of techniques have been proposed which will be reviewed in section 2. We propose multiple techniques to improve the efficiency of hash tables and their summaries in section 3.

1.3 Motivation

Even more than ten years after its proposal, IPv6 is still sparsely used. The main reason is the high requirement IPv6 poses to the hardware. Current solutions require vast amounts of extremely expensive but fast on-chip memory. The economic pressure on providers to deploy an IPv6 ready network does not outweigh the high costs associated with it. However, the explosion of hosts, especially in the eastern world, leads to an omnipresent need for a higher address space. An evaluation of the *status quo* and an evolution towards cost-efficient solutions that are deployable in the internet core are long overdue.

1.4 Position and Outline

This work analyzes state-of-the-art data structures, their applicability in IPv6 core routers, and suggests mechanisms to improve the efficiency of prefix-storing data structures in terms of memory requirements and router production cost. We are especially interested in aggressively reducing the number of bits needed for the summary representation to allow table sizes of millions of entries that are common in the internet core. Our design is based on four key-ideas which have been ignored so far. As we will show, these observations allow a decrease in size by a factor of 10. Section 2 discusses approaches currently in use or suggested. First, a basic overview of LPM algorithms is given to demonstrate the complexity of this task, followed by an introduction of Bloom filters and derivates. The section concludes by reviewing different hash table structures and their summaries. Section 3 starts with a hash table and summary analysis emphasizing their applicability in core routers. We then present our key-ideas and an improved and adjustable hash table design specialized for IP-lookup applications. Section 4 discusses the results of our simulations. A conclusion as well as a discussion on future work is given in 5.

1.5 Conventions

Throughout this thesis we use the following conventions.

– We strongly stick to the International System of Units (SI) [1], the SI prefixes and the approved prefixes for binary multiples. That is, we use the prefixes k,M,G ... only for multiples of 10 and the prefixes Ki,Mi,Gi ... for multiples of 2.

[1] http://physics.nist.gov/cuu/Units/

- 'Iff' with two 'f' denotes 'if and only if' throughout this document.
- Unless otherwise stated 'log' denotes the logarithm to base 2.
- 'ln' denotes the natural logarithm to base e.
- $|x|$ denotes the length of x.
- $[x]$ denotes the range of x. Unless otherwise stated it is defined as $[x] = \{0, \ldots, x-1\}$.
- == denotes equals.
- In pseudocode, $\leftarrow$ denotes assignment where = denotes equals.

2 State-of-the-art in Packet Forwarding

This section discusses data structures and algorithms that are widely used in networking applications and establishes the basis for future sections. Specifically we focus on packet forwarding mechanisms, however, many techniques have other uses in networking which are briefly mentioned where appropriate.

2.1 Longest Prefix Matching

A lot of work has been done in longest prefix matching algorithms. For an extensive overview see [1]. This work focuses on optimizing the data structures used for storing prefixes so only a brief introduction to LPM techniques will be given here to show the complexity and challenges.

The most basic data structure is a binary trie. Prefixes are stored in a search trie and nodes that represent a match are marked as terminal nodes. The destination address of an incoming packet is then used to traverse the trie until the longest match has been identified. This solution has linear complexity and thus is rather inefficient. Various improvements have been suggested to reduce complexity which will not be recounted here.

Waldvogel et al. [2] propose a "binary search on prefix lengths". The idea is threefold. First, hashing is used to identify matches of a specific prefix length. Then, binary search on the prefix length levels is performed to divide the potentially large search domain into sub-domains. Finally, to avoid backtracking in case of failure, so called markers are precomputed that can be used to identify matches in higher levels. The prefixes are sorted and grouped in hash tables by length. The binary search begins with the longest prefix length and proceeds backwards. A match in a table indicates, that the longest matching prefix is at least as long as the size of the prefixes in the queried table while a failure indicates that tables for shorter prefixes must be searched. This scheme reduces the complexity to logarithmic which is a significant improvement over previous linear search tries.

A similar approach is taken by Dharmapurikar et al. [3] who introduce a Bloom filter based approach for LPM. As in the binary search scheme the prefixes are grouped and stored in hash tables by their length. Instead of building a search tree, Bloom filters are used to represent the sets of prefixes of a specific length. A query on all filters results in a matching vector and the associated hash tables are probed to retrieve the longest prefix.

2.2 Bloom Filters and Derivates

Bloom filters have long been popular in database systems and gained a lot of attention in network applications. Only a brief introduction to the most prominent representatives is given here. For a detailed overview of Bloom filters in network applications see [4].

Bloom Filter. *Bloom filters*, first introduced by Burton H. Bloom [5], 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 return 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 a query for an item y is performed, 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. It is given as

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

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

$$k = \frac{m}{n} \ln 2. \tag{2}$$

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

$$m = c \cdot n \tag{3}$$

for a sufficiently large constant c.

Counting Bloom Filter. The problem of standard Bloom filters is that they do not support 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. [6] address this issue by introducing a *counting Bloom filter* (CBF). Instead of a bit array, the CBF maintains an array of counters $C = \{\varsigma_0, \ldots, \varsigma_{m-1}\}$ 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 dedicated

memory. With a counter-width of υ bits the amount of bits needed for the CBF is

$$\beta = m \cdot \upsilon. \tag{4}$$

Multistage Counting Bloom Filters. Estan and Varghese [7] use counting Bloom filters for traffic management and large flow identification. When a packet arrives, its flow ID is hashed into multiple independent Bloom filters. If all counters exceed a certain threshold the flow ID is added to the flow memory, dedicated for large flows. The stages can either be organized and accessed in parallel or sequentially, in which case subsequent stages need only be accessed if the flow ID passes the previous stage.

Spectral Bloom Filters. Similar to counting Bloom filters, Cohen and Matias [8] introduce a structure called *spectral Bloom filter* (SBF). The SBF also uses counters, but, whereas Bloom filters in general are used to represent a set of unique items, the SBF serves as a histogram of multi-sets keeping track of the frequency of items. Since the counters in this scenario can grow large rapidly, the authors emphasize minimizing space. The counters are compressed using elias encoding and stored consecutively. A complex data structure composed of multiple offset vectors is used to index the compressed counter string.

Bloomier Filters. *Bloomier filters* [9] generalize Bloom filters to represent arbitrary functions that map a subset $S = \{s_0, \ldots, s_n\}$ of a given domain D to a defined range $R = \{v_0, \ldots, v_{|R|-1}\}$. Items can be associated with values which are encoded in the bloomier filter such that $f(s_i) = v_i$ for $s_i \in S$ and $f(x) = 0$ for $x \notin S$. The basic construction is composed of a Bloom filter cascade. For every possible value in R one Bloom filter is used to represent the items mapping to this value. However, due to false positives, multiple filters could return the membership of an item and thus the correct value could not be retrieved. To resolve this problem, filter pairs are introduced that hold the items producing false positives. A query on the pairs can then identify filters that produced false positives.

Compressed Bloom Filters. Mitzenmacher [10] proposes arithmetic coding for Bloom filters used as messages. In his scenario the Bloom filters are used in a distributed system to exchange web cache information. Clearly, to prevent network stresses the transmission size must be small. Using arithmetic or delta encoding the Bloom filter can be compressed without sacrificing performance (in terms of false positive rate). Note, that while queries on such a compressed filter are possible, it does not support updates.

Multilayer Compressed Counting Bloom Filters. Recently, Ficara et al. [11] introduced a compression scheme for counting Bloom filters that allows

updates. It is known as ML-CCBF (MultiLayer Compressed Counting Bloom Filter) and 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 $\varphi + 1$. 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 code-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 its 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. The authors propose an index structure to lower the cost of a counter lookup. All bitmaps are split into D equal sized blocks. An index table is kept in extra memory that holds the number of '1' at the start of each block. With a bitmap size of m_i split into D blocks, $log\ \frac{m_i}{D}$ bits are needed to index the table. Thus, only the number of '1' from start of the block to u_i need to be counted.

2.3 Hash Tables

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. In this scheme, the load of the buckets can grow quite high. Azar et al. observed [12], that by allowing more possible destinations and choosing that with lowest load, the upper bound can be reduced exponentially. This effect became popular as 'the power of two choices', a term coined by Mitzenmacher in [13]. Further improvement achieved Vöcking [14] with the 'always-go-left' algorithm. Here, the items are distributed asymmetrically among the buckets. Broder and Mitzenmacher [15] suggest using multiple hash functions to improve the performance of hash tables. The n buckets of the table are split into d equal parts imagined to run from left to right. An item is hashed d times to find the 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.

d-left Counting Bloom Filters. A major problem with CBFs is the massive amount of space needed compared to a basic Bloom filter. Bonomi et al. [16] present an improved version of CBF based on d-left hashing. They make use of the fact that with d-left hashing one can achieve an "almost perfect hashing"[15]. The idea is as follows. The m buckets are split into d equal sized sub-tables imagined to run from left to right, leading to d choices to store an item. Each bucket consists of c cells holding a fingerprint of an item and a small counter. The fingerprint is twofold, the first part is the bucket index and the next part is called the remainder r. Only the remainder is stored. One hash function $H : U \rightarrow [B] \times [R]$, where [B] ([R]) is the range of buckets (remainder), produces the fingerprint. Then d random permutations are used to identify the buckets in the sub-tables. Note, that the remainder is different depending on the sub-table. Upon insertion, the d buckets are first checked whether they already hold the remainder r_i. If so, the cell counter for r_i is incremented. If not, the item is stored in the most left sub-table with the smallest load. Ties are broken by going left. Retrieving items requires d parallel lookups. A false positive can occur iff $H(x) = H(y)$ since this leads to the same permutations.

Fast Hash Table. A drawback of hashing with d choices is that it requires at least d lookups in the hash table. Though these are independent and can be done in parallel, it is inefficient to do so. In [17] Song et al. present a data structure named *fast hash table* (FHT) that eliminates the need for parallel lookups through the use of a counting Bloom filter summary. In their scheme only one bucket access is needed. Each counter corresponds to a bucket in the hash table and represents the number of items hashed into it. Thus, the k hash functions which are derived by equation 2 are used to index both the CBF and the hash table. Song et al. use a class of universal hash functions [18] to construct the Bloom filter and the hash table. These only work with multiples of 2. They use the following equation to compute the number of buckets.

$$m = 2^{\lceil \log c \ n \rceil} \tag{5}$$

Where $c = 12.8$. 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 far 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.

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. 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 are decremented. Lookup is done by hashing the item k times and computing the minimum counter value z. If $z \neq 0$, the item is retrieved from the far left bucket corresponding to z, limiting the lookup time to z. This scheme leads to high bucket loads, thus, retrieval of an item is most certainly accompanied by following multiple pointers. Figure 1 shows an example BFHT.

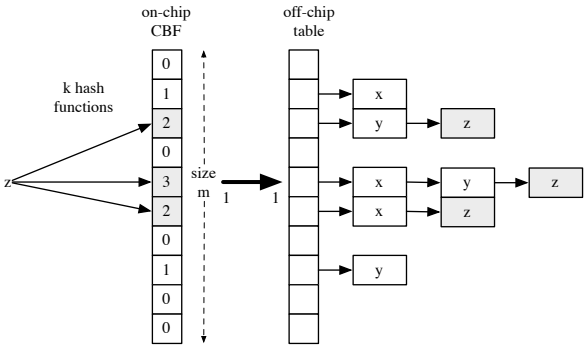


Fig. 1. Basic fast hash table

Pruned Fast Hash Table. The pruned FHT (PFHT) is an improvement on the BFHT. Items are only stored at the far left bucket with minimum counter value. Counters and lookups are handled as in the BFHT. This improves bucket load and lookup time. The authors show that given a well designed table the buckets will hold only one item with high probability. However, not storing every item in all corresponding buckets complicates updates since they influence the counters of already present items. Minimum counters of items inserted earlier might get changed during update leading to a lookup in the wrong bucket. For insertions the items 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 its items, it is not possible to identify items that have to be relocated. This can either be achieved by examining the whole PFHT and check every item (obviously this is very expensive), or by keeping an offline BFHT and examining affected buckets offline. Thus, the PFHT is only suitable for applications where updates are much rarer than queries. Figure 2 illustrates the pruned version of the BFHT depicted in figure 1.

Shared-node Fast Hash Table. The shared-node FHT (SFHT) provides support for update critical applications at the cost of slightly higher memory con-

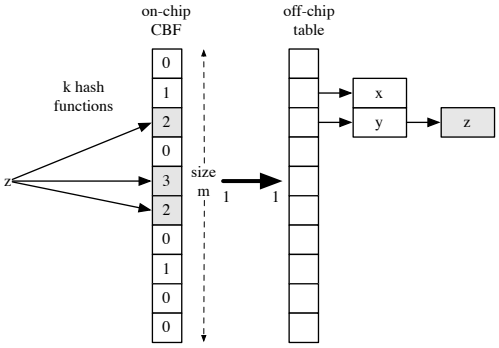


Fig. 2. Pruned fast hash table

sumption. Here the buckets only store a pointer to the first item that has been inserted. The items are stored in extra memory and carry a pointer to the next item in the list. Special care must be taken when an item is inserted that hashes to empty and non-empty buckets. Appending this item to the linked lists would lead to inconsistencies. It must be replicated and pointers set in the empty buckets and the linked lists accordingly. Again counters and lookup are treated as in the BFHT. Though updates are much easier compared to the PFHT, lookup now requires following at least one pointer.

Multilevel Hash Table. The Fast Hash Table uses one counter per bucket to keep track of the number of items stored. While this is a straightforward approach that is easy to implement, it has rather high memory requirements for the counting Bloom filter summary. Kirsch and Mitzenmacher [19] 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 [20], to store the items. The MHT consists of

$$d = \log\log n + 1 \tag{6}$$

sub-tables where each sub-table T_i has $c_1 c_2^{i-1} n$ single item buckets with $c_1 > 1$ and $c_2 < 1$. Thus $|T_i|$ is decreasing geometrically for increasing i. An occupancy bitmap is kept in on-chip memory with one bit per available bucket that allows efficient queries for empty buckets. The total number of bits needed for the occupancy bitmap is equal to the number of buckets which can be derived by

$$\beta = m = \sum_{i=1}^{d} (c1 \cdot c2^{i-1} \cdot n). \tag{7}$$

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.

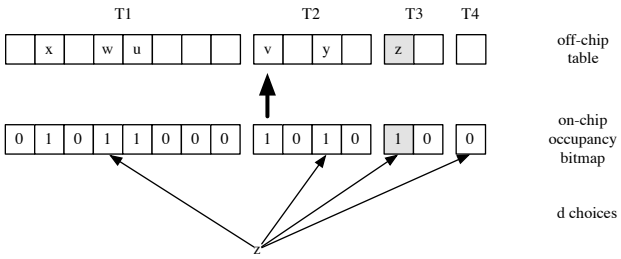


Fig. 3. Multilevel hash table

Figure 3 shows an example with four sub-tables. A so called crisis can occur when all d buckets are occupied. However, it can be proven that for any $c_1 c_2 > 1$ the crisis probability is insignificantly small. Kirsch and Mitzenmacher present three summary data structures which will now be reviewed.

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. Figure 4 illustrates the construction. Interpolation search is used to search for an item which requires

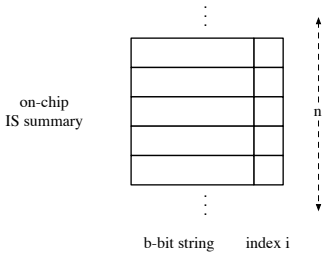


Fig. 4. Interpolation search summary

the array of strings to be ordered. Insertions and deletions requires shifting subsequent strings to keep the ordering. A failure can occur if two inserted items hash to the same b-bit string. The failure probability is

$$p_{fail}(n,b) = 1 - \prod_{k=0}^{n+1} \frac{2^b - k + 1}{2^b}. \tag{8}$$

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

$$p_{fp}(n,b) = \frac{n}{2^b}. \tag{9}$$

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

$$\beta = n\ (b + \log d). \tag{10}$$

Single Bloomier Filter. The Single Bloomier Filter summary (SF) has

$$m = n\ \log n \tag{11}$$

cells initialized to 0 and represents the type t of an item where t is the sub-table the item is stored in.

$$k = \log n \tag{12}$$

hash functions are used to access the Bloomier Filter. To insert an item, first its type is identified by inserting it into the MHT. Then it is hashed k times and the corresponding cell values are replaced with the maximum of their value and the type of the item. Insertion is depicted in Figure 5. To search for an item the

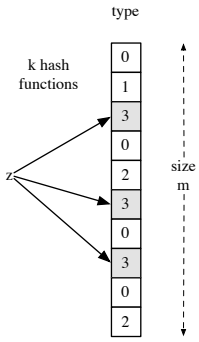


Fig. 5. Single filter summary

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 number of bits needed for the single filter summary is

$$\beta = n\ \log n\ \log\log\log n \tag{13}$$

Multiple Bloom Filters. The single filter approach introduces type failures and care must be taken during construction since false positives and type failures are competitive in respect to the number of hash functions used. 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 geometrically an array of Bloom filters $B = \{B_0, ..., B_{t-1}\}$ decreasing in 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 of 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. Unfortunately, the authors do not clarify how the number of hash functions needed can be derived, but give examples of seven hash functions for B_0 and 49 for each of the other filters with $n = \{10k, 100k\}$. However, the hash functions between Bloom filters do not need to be independent, so the same set of hash functions can be used for each filter. Figure 6 illustrates the design. With a well designed MHT the total number of

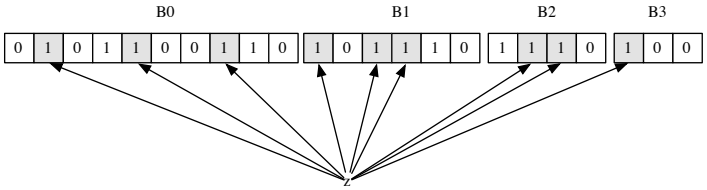


Fig. 6. Multiple Bloom filter summary

bits for the MBF is

$$\beta = n \ \log n \tag{14}$$

Deletions. The Bloom Filter based summaries only support inserts. To allow deletions significantly more effort is needed in terms of additional or modified data structures. Two deletion schemes are proposed in [19], the *lazy deletions* and *counter based deletions*

Lazy Deletions. A simple approach for adding deletion support is *lazy deletions*. Like the occupancy bitmap, a deletion bit array is kept in on-chip memory 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. Though being simply, it leads to inconsistencies in the MHT, since present and newly inserted items are placed further right, than needed. 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 schemes 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 and a counter-width υ it would require

$$\beta_{sfcounter} = \upsilon \cdot d \cdot n \ \log n \tag{15}$$

bits in total. 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 a total number of bits equal to

$$\beta_{mbfcounter} = 3.3 \ n \ \log n \tag{16}$$

3 Efficient Hash Tables

In this section we first analyze the MHT and FHT for their applicability of IPv6 core routing. We then introduce new ideas to construct space efficient hash table summaries. Our optimizations are strongly based on the scheme of Song et al [17]. However, some are also adaptable to the design of Kirsch and Mitzenmacher [19] and are mentioned where appropriate.

3.1 Hash Table Analysis

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 limited. Common reasons are

- missing evaluation for millions of entries
- need for pointer following
- high requirements of fast on-chip memory.

Considering the fast hash table types of [17], the basic and shared-node FHT need following next pointers which makes them unappealing for use in high-speed IPv6 routing. Only the PFHT seems attractive since bucket loads are much lower than in its sister structures, being one with very high probability. However, the number of bits needed for the summary is quite high. Unfortunately, Song et al. only present evaluations for tables with $n = 10,000$ items.

The MHT provides a constant lookup time of $O(1)$ which is optimal. The interpolation search (IS) summary has high memory requirements stated 64 bits per item for $n = 100k$ 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 which disqualifies this structure for IPv6 lookup applications. The single filter (SF) needs less space but does not support deletions. The smallest summary is the multiple Bloom filter (MBF), but it has similar constraints 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 needed amount of memory. Kirsch and Mitzenmacher provide evaluations for $n = \{10k, 100k\}$ and compare their results with the scheme of Song et al. However, they neglect the overhead produced by adding deletion support. The authors state, that since the PFHT needs additional offline structures to support deletions this is a fair comparison. This is true in terms of functionality, but the PFHT offline structure does not affect the needed amount of on-chip memory, while the SF and MBF summaries have much higher on-chip requirements if deletions are supported. Therefore, our analysis respects deletion support overhead in all cases.

The choice of hash functions is also important for evaluation. [17] uses a class of universal hash functions [18] to access the filter and hash tables. While they are easy to implement in hardware they always produce numbers in the range of $[2^b]$ where b denotes the address space in bits. To prevent modulo computation

Song et al. choose the filter and table size as multiples of 2 which leads to a non linear growth for bigger tables. Strictly speaking, this is not necessary. The FHT can also be constructed with arbitrary hash functions that allow linear growth. This is also the basis for the evaluations of the summaries in [19] which makes a fair comparison difficult.

Both the PFHT and MHT are analyzed for their behavior with arbitrary and universal hash functions. For the PFHT the number of buckets is calculated using equations 3 for arbitrary and 5 for universal hash functions with $c = 12.8$. The resulting amount of dedicated on-chip memory is derived from equation 4 with $\upsilon = \{3, 4\}$. The number of buckets for the MHT is calculated using equation 7 with $c1 = 3$ and $c2 = 0.5$ as suggested in [19]. This equals the number of bits needed for each of the occupancy and deleted bitmaps. Thus, in case of lazy deletions it is added twice to the size of the summaries, for the counter-based deletions it is added only once. The single filter summary size is derived using equations 13, for lazy deletions and 15 for counter-based deletions with $\upsilon = 2$. MBF summary sizes follow equations 14 (lazy) and 16 (counter-based).

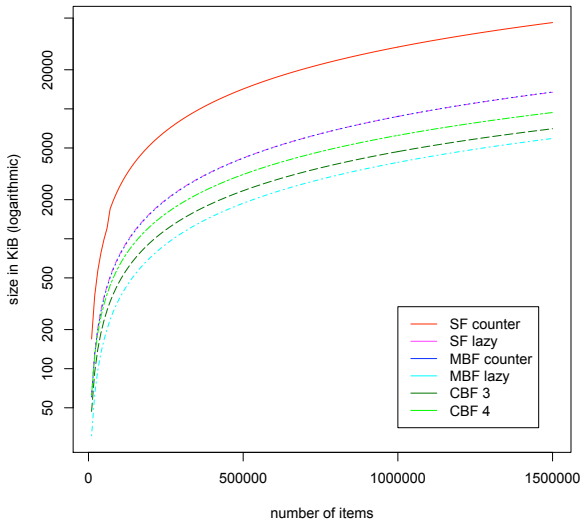


Fig. 7. Memory consumption in KiB, $n \leq 1.5m$, arbitrary hash functions

Figure 7 shows the amount of memory needed for the summaries with $n \leq 1.5m$ and arbitrary hash functions. Figure 8 is the equivalent with universal hash functions. Figures 9,10 show the sizes for $1m \leq n \leq 4m$. All summaries scale equally, with the MBF performing best followed by the CBF. In case of using universal hash functions, there are small areas where the SF lazy scheme

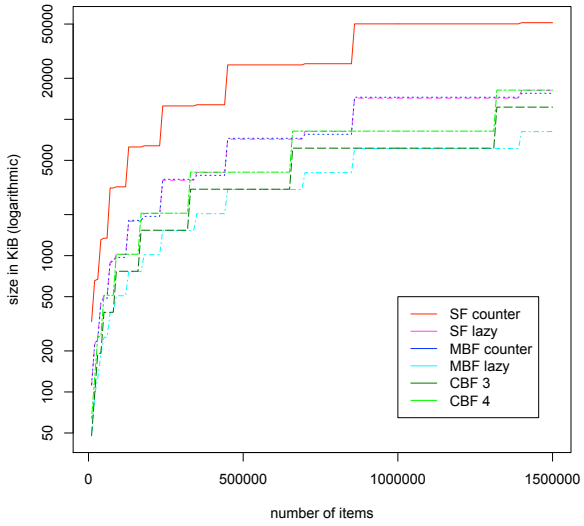


Fig. 8. Memory consumption in KiB, $n \leq 1.5m$, universal hash functions

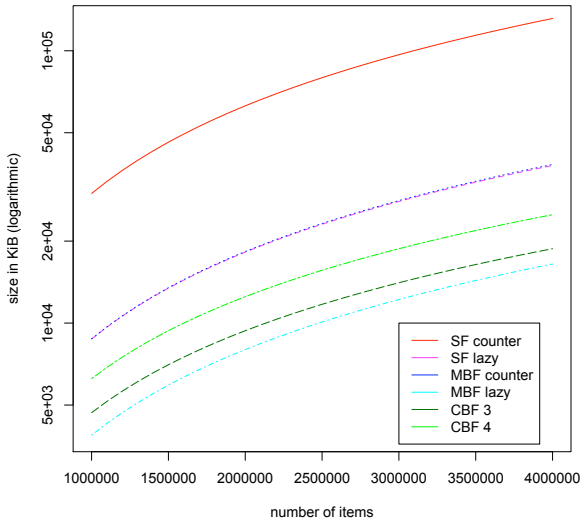


Fig. 9. Memory consumption in KiB, $1m \leq n \leq 4m$, arbitrary hash functions

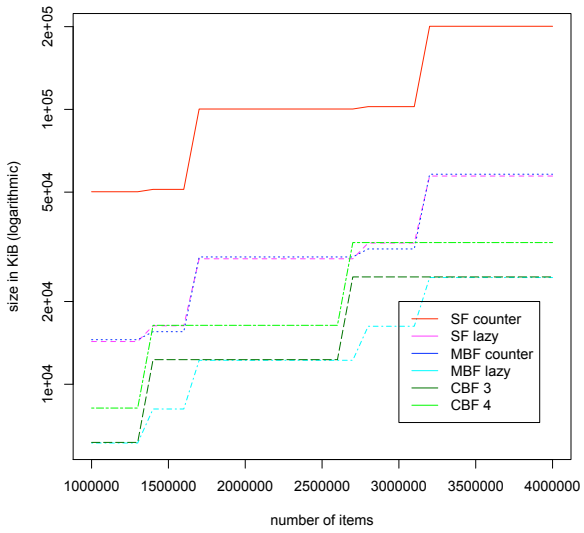


Fig. 10. Memory consumption in KiB$1m \leq n \leq 4m$, universal hash functions

is actually smaller than the CBF if a counter-width of 4 is assumed for the latter. However, all summaries grow very large with $n \geq 1m$. The needed amount of memory is around 5 MiB increasing to over 20 MiB for $n = 4m$. Though this appears small, even today the amount of on-chip memory dedicated to the lookup process is far below this limit. A number of other applications, like flow identification/control and security mechanisms also require fast on-chip memory to work at line speed. The amount of on-chip memory available for packet forwarding is in the order of tens of Mbit. Table sizes of around 4m entries are not uncommon in the core. Therefore, none of the current structures are suitable for deployment in core routers.

3.2 Key Ideas

Based on the work of [17] and [19] we propose mechanisms to construct improved data structures specifically designed for IP-lookup applications. We call the design *Efficient Hash Table* (EHT) where efficient primarily relates to on-chip memory usage but also to lookup performance. The main focus lies on aggressively reducing the number of bits needed for the summary to allow cost efficient router production, while still retaining a lookup performance of one memory access per lookup.

We base our design on the following four observations or key ideas.

– The false positive probability can be ignored.

– A hash table bucket can hold more than one entry without the need to follow next pointers.
– The lookup engine can be separated from the update engine.
– The summary can be encoded using compression.

Lemma 1 *The false positive probability can be ignored.*

Proof. The router must provide a worst case lookup performance at link speed to prevent buffer overflows. The number of lookups needed to find the correct prefix is upper bound by the LPM technique used. The underlying data structure must have a predictable lookup performance to evaluate worst-case behavior. Whether or not the lookup is actually made has no impact on worst-case performance. Lookup performance is thus independent from the false-positive probability.

Lemma 2 *A hash table bucket can hold more than one entry without the need to follow next pointers.*

Proof. Let a bucket b equal the number of bits that can be read with one memory burst and x equal the number of bits representing the entry. If $x \ll b$, a bucket can hold up to $\lfloor \frac{b}{x} \rfloor$ entries.

Lemma 3 *The lookup engine can be separated from the update engine.*

Proof. IP-lookup, as the name implies, is a heavily lookup driven application. Updates occur infrequently and much rarer than lookups. In addition, they are not time critical and need not take effect instantly. Updates can be computed offline and changes to the online structures applied afterwards.

Lemma 4 *The summary can be encoded using compression.*

Proof. As long as the compression scheme provides real-time compression and incremental updates and is further easy to implement in hardware, the summary can be compressed without affecting the lookup performance.

Bloom filters are generally constructed prioritizing the optimization of the false-positive probability. This is indeed an important aspect in many applications, for example those related to security or flow identification and control. Considering IP-lookup, as long as the location of an item can be identified independent of the false-positive probability it is unimportant for lookup performance. The cost of a lookup of an item not in the table can be ignored, since it does not affect the worst-case performance. Ignoring the false-positive probability in designing the summary allows concentrating on optimizing the size m. Thus, the summary is explicitly used to identify the location of an item, *not* to separate items in the table from those not in the table. Of course, this is only applicable if successful lookup is not affected by the false positive probability. The MHT summaries depend on an extremely low false positive probability to prevent type failures. Thus lemma 1 can not be applied to the MHT.

Reducing the size m of the summary and the hash table also affects counter values and the average load of the buckets. The higher load can be compensated by adding additional off-chip memory, either in terms of wider memory or by using CAM or a combination of both. Thus, there exists a tradeoff between on-chip and off-chip memory requirements. We will show, that this tradeoff is reasonably small. Significant amounts of expensive on-chip memory can be saved by trading in comparatively small amounts of off-chip memory.

Examination of previous work like the MHT and FHT shows that updates - especially deletions - add significant overhead. In IP-lookup applications updates occur extremely rare compared to lookups. Core routing tables are very static by nature. Changes are scarce and rerouting is accompanied by propagation delay. Considering a core router with $1,000$ 10 Gb/s ports approximately $1,000,000$ lookups have to be performed per second. The ratio of updates to lookups is in the order of one to billionth even if we presume one update per day. Thus, one can not justify the high cost of adding update support. In conclusion, the lookup engine does not need to support updates but instead can be optimized for lookup performance. This offloads the update overhead to a designated update engine which precomputes all changes offline and applies them to the lookup engine afterwards. Of course, this increases the complexity significantly and adds additional memory requirements for the offline data structures. But, considering the high costs of on-chip memory this can be easily justified. Separating updates from lookups is applicable to the FHT and MHT structures. We will discuss both approaches in section 3.5.

To further reduce on-chip memory requirements the online summary can be compressed. While the idea of compressed Bloom filters is not new *per se*, their integration in IP-lookup applications has not yet been exercised. There are many restraints on a suitable compression algorithm. It must be easily implementable in hardware and must provide counter individual real-time de-/compression. Otherwise, the lookup process would degenerate and updates to the compressed structure would become impractical. We propose two compression schemes that fulfill these requirements in sections 3.5 and 3.6.

The following sections discuss all our ideas in great detail and incrementally introduces our design. As already mentioned, most optimizations add further complexity in terms of additional structures or offline and off-chip memory requirements. Therefore, we give multiple options in designing efficient hash tables and carefully weight the cost of each. Results for simulations are discussed in section 4.

3.3 Ignoring the false positive probability

The major reason for having relatively large Bloom filters is to minimize the false positive probability. There are various applications for which this is an important design issue. However, as proven in Lemma 1 the IP-lookup performance does not suffer from higher false positive rates as long as the summary returns the correct value independent of the false positive probability. In conclusion, counting Bloom filter summaries can potentially be much smaller. By reducing the address space

counter values and the load of buckets are expected to increase. So there exists a tradeoff between reducing on-chip memory requirements and the resulting counter values and bucket loads.

The problem is to identify a size m that optimizes this tradeoff. We will first analyze the effect of m on the counter values and then move to its impact on bucket loads.

Counter Values. Counter values follow a binomial distribution. With m possible locations and nk insertions (each insertion increments k counters) the probability p_i that a counter received is incremented exactly i times can be calculated using the following equation [17].

$$p_i = \binom{nk}{i} (\frac{1}{m})^i (1 - \frac{1}{m})^{nk-i} \tag{17}$$

This is not absolutely accurate. The probability that actually less than k counters for an item can be increased due to hash collisions is neglected. However, the estimate is close enough to allow prediction on counter values. One other problem remains, that is how to choose k. The optimal is given by equation 2. However, this leads to floating point numbers. Since we cannot have fractions of hash functions we have to normalize k. This is done by applying the ceiling function to the result of equation 2, thus, rounding up to the next integer.

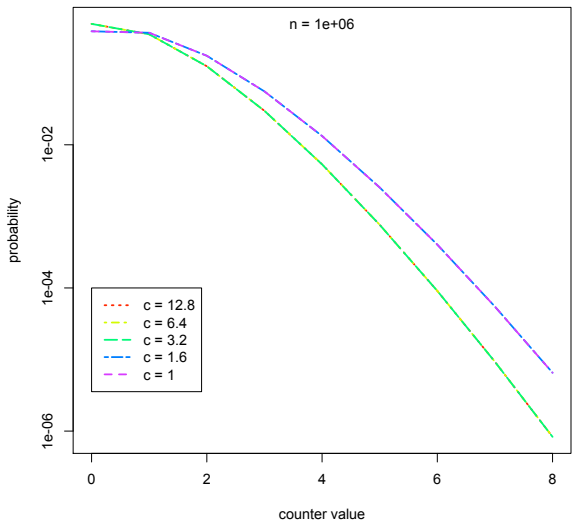


Fig. 11. Counter value probabilities for different m

Figure 11 shows the counter distribution for different m. Equation 5 is used to calculate m. The constant c is chosen such that the size is divided by multiples of 2. The result is biased. For $c > 1.6$ the length m has no effect on the counter distribution. For $c \leq 1.6$ the probability for higher counters is increased. At first thought this is against reasoning. For smaller c the number of hash functions k is also smaller, thus, less items are inserted in which case one would assume smaller counters. To clarify why the counters are actually higher, the effect of the number of hash functions on counter distribution must be examined.

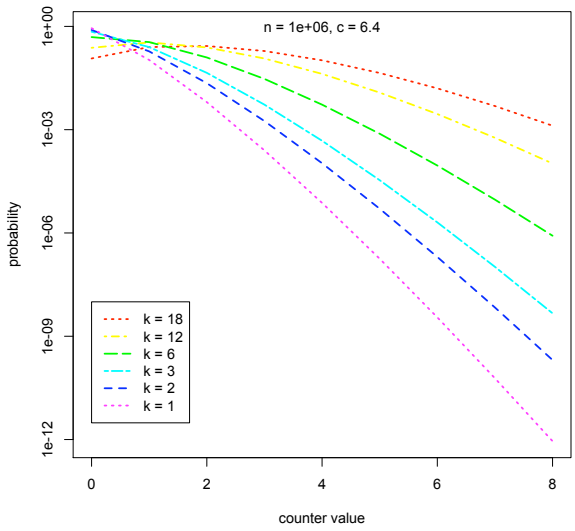


Fig. 12. Counter value probabilities for different k

Figure 12 shows the counter distributions for different k. In this example we choose $c = 6.4$ in which case the optimal value for k is 6. As can be clearly seen, the number of hash functions has great impact on counter values. The more hash functions greater than optimal, the higher the probability for higher counters. If there are less than optimal hash functions, probability of '0' is quite high with extremely low probability for counters > 2. The effect can better be explained by using a non logarithmic scale depicted in figure 13.

The number of hash functions influences the peak, the expansion and the alignment of the binomial counter distribution. This is expected, since the binomial coefficient $\binom{nk}{i}$ depends on k. In general, the less hash functions, the more the center of the distribution approaches 0 while expansion is small and the peak is high. Vice-versa, the more hash functions, the center gets moved away from 0, expansion is greater, and the peak is lower. This partially answers the question

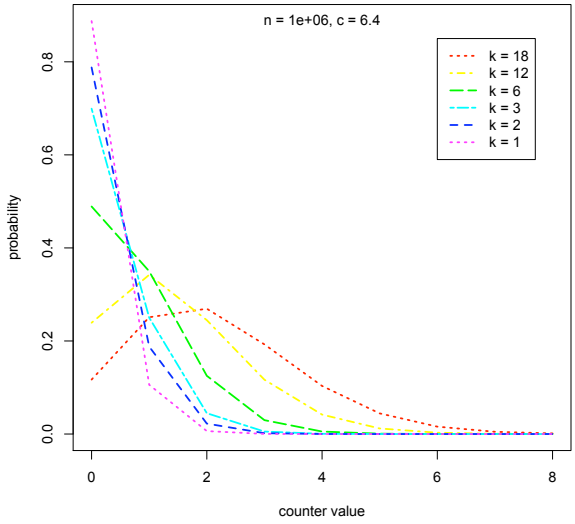


Fig. 13. Counter value probabilities for different k

from above, why for small c probability of higher counters increases. Now consider how k is derived. It depends on the ratio $\frac{m}{n}$ which can also be described as the number of counters/buckets reserved for one entry. Since n grows linearly and m by multiples of 2 (see equation 5), $\frac{m}{n}$ will not scale linearly to n. In addition the normalization of k will always lead an overestimate of the optimal number of hash functions. Thus, the counter distribution does not scale with n.

Figures 14, 15 depict the normalization for $100,000 \leq n \leq 1,000,000$ and $c = \{1.6, 3.2\}$. For $c = 1.6$ the gap between the normalized k and it's optimum is much higher than for $c = 3.2$. Paired with the low amount of counters/buckets per item this results in an increased probability of higher counters. The effect is the same for $c < 1.6$ and $c > 3.2$ and is left out for simplicity. This could be prevented by always rounding k down to the next lower integer. However, a smaller number of choices leads to higher bucket loads. Therefore, we favor more choices at the cost of higher counter values over less choices and higher bucket loads.

In conclusion, the counter distribution depends on the fraction $\frac{m}{n}$ which is the number of counters/buckets per item and on the number of hash functions k. The distribution does not scale with n. If for equal k the fraction $\frac{m}{n}$ gets too small counters grow larger. Otherwise, with k being optimal or lower, it is not expected that the counters grow larger for smaller m.

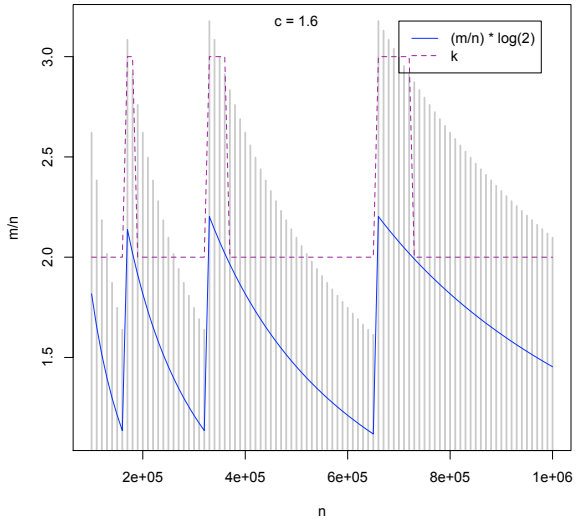


Fig. 14. Optimal k and normalization for c = 1.6

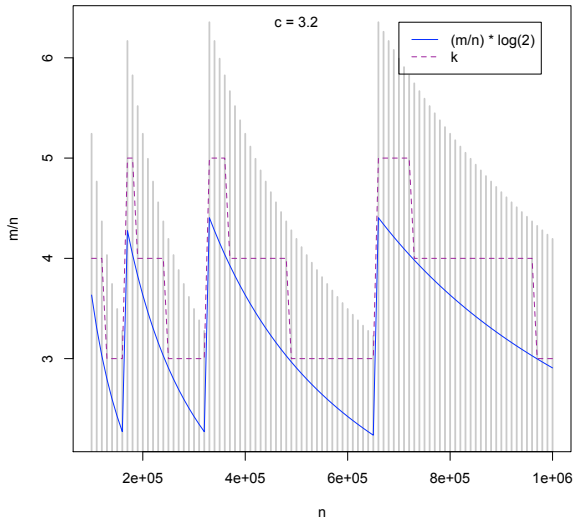


Fig. 15. Optimal k and normalization for c = 3.6

Bucket Load. Calculating the load of the buckets is not simple. Suppose we throw n items into m buckets uniformly at random. Then, the distribution would again be binomial. However, since we have k choices for every item and on each toss we favor the least loaded bucket the distribution becomes unbalanced. Probability for lower loads will be higher while for '0' and higher loads the probability will shrink. Unfortunately, to derive the distribution we would need to incrementally calculate the probabilities of every toss. With millions of items this is impractical. A better approach is to follow [12] and predict the expected maximum load that occurs with very high probability. With n items, m buckets and k choices the expected maximum load is defined as

$$E_{maxload} = \frac{\ln \ln m}{\ln k}. \tag{18}$$

The equation holds for any $m \to \infty$ with $n = m$ and $k \geq 2$. In our design, however, $m \gg n$. Thus, the result leads an overestimate of the maximum load, which in practice should be smaller. For this reason we apply the floor function to the result of equation 18 to round to the next lower integer. A special case is if $k = 1$. This happens when $\frac{n}{m} \approx 1$. The maximum load is then

$$E_{maxload} = \frac{\ln n}{\ln \ln n} \tag{19}$$

with high probability. Since n is slightly lower than m we again apply the floor function to compensate the overestimate.

The results can be seen in figure 16 and are surprisingly positive. Setting $c = 3.2$ results in a summary size of $\frac{1}{4}$ of the optimum proposed in [17]. The maximum load increases from 1 to 2 w.h.p. In other words, increasing the off-chip memory width by a factor of two allows a reduction in on-chip memory size by a factor of four. The tradeoff is even better for $c = 1.6$. With a three times wider off-chip memory, the on-chip memory size can be reduced to $\frac{1}{8}$ of the optimum. The effect of k greater than optimal is marginal and can therefore be neglected.

The next section discusses off-chip memory requirements in more detail and shows possible improvements.

3.4 Multi Entry Buckets

Lemma 1 states, that the address space, or size, m of the summary can be reduced at the cost of a higher false positive probability and higher bucket loads. These can be compensated by increasing the off-chip memory width, thus, allowing multiple entries per bucket which can be fetched in one memory cycle. To specify the width needed, a deeper look into IPv6 prefix allocation has to be made.

Theoretically, with IPv6 the prefixes can be as large as 128 bits which would equal a specific host address. However, according to [21], [22] less than 5% of the prefixes exceed 48 bits with the vast majority having up to 32 bits and no prefix being longer than 64 bits. Efficient LPM algorithms sort the prefixes by

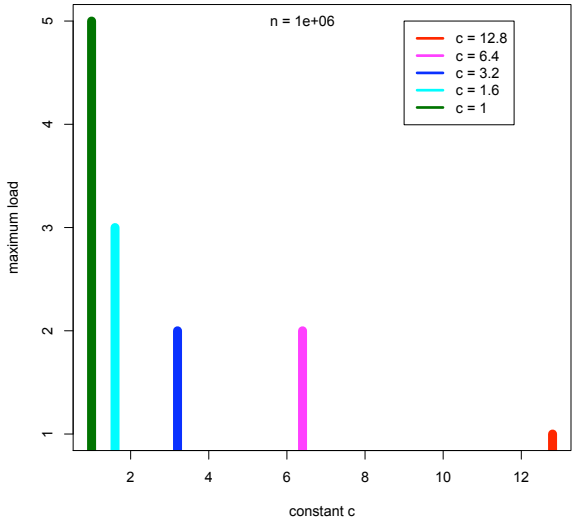


Fig. 16. Expected maximum load for different c

length and store them in multiple hash tables according to length. Only a small minority of the tables will hold prefixes with more than 48 bits, and can be treated differently. Therefore, we optimize the off-chip memory to deal with the majority of the prefixes.

Figure 16 shows the expected maximum load for different c which specifies the size of the table. Assume a router with $1,000$ possible ports and prefixes up to 48 bits. A table entry that matches prefixes to the outgoing port would then require 58 bits. Setting $c = 3.2$ results in an expected maximum load of two, thus, a 116 bit wide memory is required. Alternatively, one can use 64 bit double-data-rate memory which allows reading two words per cycle. Hence, for $c = 1.6$ either a word-size of 174 bits or a 87 bit DDR memory is required.

The size of an entry can further be decreased by using a hashing scheme similar to that in [16]. A class of hash functions can be used that do a transformation of the key, producing k digests of the same size as the key. The same size is crucial to prevent collisions and the hash function must be collision resistant. An example is CRC, which is well known and easy to implement in hardware. The digest is imagined to be composed of two parts, the index to the hash table, and the verifier of the key. Let x be the key, H the class of hash functions, $[A]$ the range of the table address space and $[V]$ the range of the remaining verifier.

$$H : U \rightarrow [A] \times [V]. \tag{20}$$

The verifier and the index are derived by bit-extraction. Let $h_{\{0,\ldots,k-1\}}$ be the k digests, then $V(h_{\{0,\ldots,k-1\}})$ produces the verifiers and $A(h_{\{0,\ldots,k-1\}})$ extracts

the bucket indexes. Instead of the prefix x only the verifier $V(h_i(x))$ is stored in bucket $A(h_i(x))$. To be able to identify which prefix corresponds to a verifier, an identifier must be kept along the verifier, that states the hash function i that produced $V(h_i(x))$. A table entry then consists of the verifier, it's identifier (which is the index of the hash function), and the associated value. Hence, $E(x) \leftarrow (V(h_i(x)), i, v)$ where v denotes the value. The total number of bits needed is $log\ k + (|H| - |A|) + |v|$ where $|y|$ is the length of y in bits.

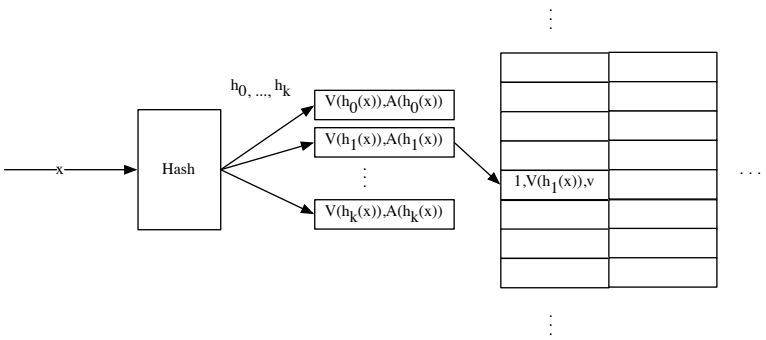


Fig. 17. Multi entry buckets and verifiers

Figure 17 depicts the process. The verifier/identifier pair allows matching entries to keys. If a key produces the same verifier with the same hash function, it is supposed to be identical. Upon lookup, the key is hashed using the k hash functions. $A(h_i(x))$ is used to access the ith bucket. The whole bucket is read and all entries examined. The examined entry's identifier j and verifier $V(e)$ are then compared to i and $V(h_i(x))$. Iff $j == i$ and $V(e) == V(h_i(x))$, then $H_i(x) == H_j(e)$ and the entry corresponds to key x. Else the examined entry can not have been produced by x. With a fixed number of entries of a fixed size, all checks can be done in parallel after the memory has been read. Note, that for k hash functions k independent transformations must be made. The transformation must be collision resistant to prevent key errors. Otherwise, different keys could lead the same verifier which would not allow a clear identification. Figure 18 shows an example of the lookup process with $k = 3$ and 3 entries per bucket.

Again consider a table of size $m = 2^{\lceil \log 3.2\ n\ \rceil}$ where $n = 1e^6$. Then 22 bits are needed to index the table. For 48 bit prefixes this leaves 26 bits to the verifier. The identifier needs an additional 2 bits to represent the $k = 3$ hash functions for a total 38 bits per entry.

Two problems have to be addressed. How to deal with the approximately 5% entries that exceed 48 bits and how to deal with overflows, in case a bucket receives more insertions than it has room for entries. If the word-size is chosen appropriately large, overflows will occur extremely rare. The easiest solution is to maintain a small CAM to hold overflown entries. However, this leads to

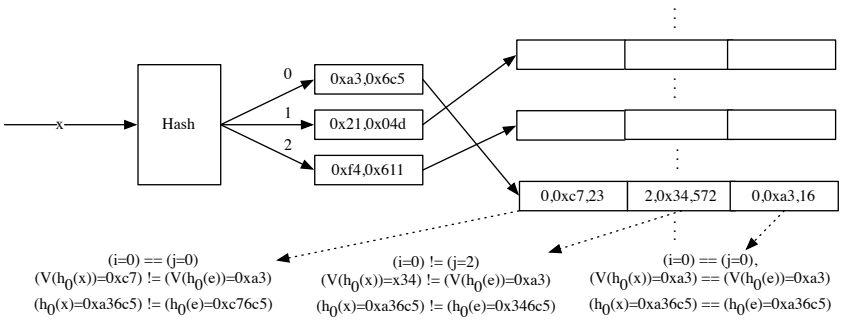


Fig. 18. Multi entry bucket lookup

more complexity to retrieve the entry. Since the summary does not distinguish between entries which are present and entries not present (remember the false positive probability is ignored), there is no possibility to know in advance if an entry should be in the table or CAM. A solution is to keep a very small on-chip memory that supports fast linear search and store the indexes of overflown buckets. In general, a bucket can only be overflown, if it's corresponding counter is equal to or exceeds the off-chip word-size. If the smallest counter for an entry is greater or equal to the off-chip word-size, the extra memory is queried for the bucket index. if the bucket has overflown a parallel lookup in off-chip memory and CAM is performed. Otherwise normal operation resumes. We will suggest a similar approach for counter overflows in section 3.6. Alternatively, to save the cost of extra memory CAM could always be queried if the smallest counter for an item leads a specific value. If a bucket overflows the corresponding counter is set to a sentinel value. All entries are removed from the affected bucket and diverted to CAM which is accessed instead of the table.

Larger prefixes are harder to deal with. Since the entries are longer, keeping them in a table with the same parameters would require more than one memory word for each bucket. Following [22] only a minority of prefixes exceed 48 bits and none are longer than 64. Thus, it is best to keep them in smaller tables with higher c such that the maximum load is lower and the entries fit into one bucket. The longest prefixes can also directly be kept in a small CAM to prevent the need for more on-chip memory. Alternatively, an additional smaller but wider off-chip memory can be provided that is dedicated to the larger prefixes. This would also allow parallel lookups for longest prefix matching and thus improve the lookup performance.

In the following we will concentrate on the majority of the tables and ignore the overhead produced by larger prefixes.

3.5 Separating the update and lookup engines

The FHT and MHT structures demonstrate that update support, especially deletes, add significant overhead to either the hash table or its summary. The PFHT needs an additional offline BFHT to identify entries that have to be relocated. The MHT summaries either need an additional deletion bitmap for lazy deletions or they must be replaced by counting filters that are much bigger in size. By separating the lookup from the update engine on-chip overhead can be avoided and the lookup summary reduced in size. The idea is to keep two summaries. One is kept online in on-chip memory and does not need to support updates but is specialized on lookup. It can be different from the offline summary which fully supports updates. When updates occur they are processed by the offline engine and changes applied to the online structures afterwards. For MHT and FHT different approaches must be taken which will now be discussed.

Multilevel Hash Table. One could argue that for an IP-lookup specialized MHT the lazy deletion scheme suffices. However, even lazy deletions require additional on-chip memory for the deletion bitmap. In fact, it is more memory efficient to use counter based deletions but to keep the counters offline and an unmodified summary online that is only updated on demand. The MBF summary is the smallest one and also provides the best room for improvements, thus only the MBF will be discussed, however, the proposed construction is equally applicable to the SF summary.

The occupancy bitmap is not needed for lookups and can be kept offline. Strictly speaking, the occupancy bitmap is not needed at all, since the table itself could be probed to identify empty buckets. However, to avoid unnecessary table accesses and not interfere with ongoing lookup processes, it is wise to identify the target location in advance and only access the table to actually store the entry. A counting MBF is stored alongside the occupancy bitmap to allow deletions. The lookup engine consists of an unmodified MBF that is kept in on-chip memory. Finally, the MHT is stored in off-chip memory. Thus, there are three storage locations, the offline storage with update engine, on-chip memory with lookup engine and off-chip memory holding the table. An overview is illustrated in figure 19.

Updates and lookups are straightforward. Upon insertion the prefix is hashed d times to find the possible storage locations and $k_{1..d}$ times to access both the offline and online MBF. The offline occupancy bitmap is probed to identify the target bucket and the type of the prefix. Then the offline MBF counters are incremented accordingly. Note, that the online MBF only needs to be updated iff at least one offline counter was 0 prior to the update and only those online cells have to be set to 1. Finally, the prefix is stored in the MHT together with its associated value. Deletions are equally simple. The prefix is again hashed d plus $k_{1..d}$ times. The offline MBF is queried to get the type of the prefix and the counters are decremented. Iff a counter value reaches 0 the corresponding online cell has to be cleared. Then the entry is removed from the table and

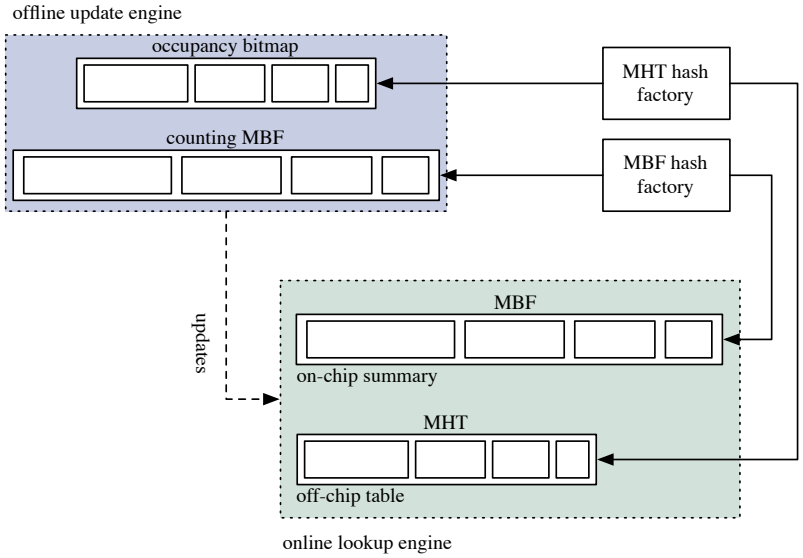


Fig. 19. Memory efficient MHT construction

the occupancy bit cleared. Lookups work just like in the original structure by querying the online MBF summary.

Clearly, this design is superior in both memory efficiency and also lookup performance. The needed amount of on-chip memory is upper bound to the number of bits needed for the MBF summary. Unfortunately, the summary still needs an extremely small false positive probability to prevent type failures, so no further space can be saved by reducing the filter sizes. On inserts, the on-chip memory has to be accessed only iff the offline counters are incremented to 1, similarly, on deletes the on-chip MBF is only updated iff counters reach 0. Thus, in addition of saving on-chip memory, it is also accessed less during updates. Though updates occur rarely and on-chip memory accesses can not be considered a bottleneck in IP-lookup applications, offloading update related work leads to better lookup performance.

Figure 20 shows a comparison of the three different MBF designs in terms of on-chip memory requirements. Graphs are shown for arbitrary and universal (base 2) hash functions. The efficient design (labelled MBF) needs only about $\frac{2}{3}$ the amount of memory than the lazy scheme. Compared to the counter based MBF the improvement is even more significant. Less than $\frac{1}{3}$ memory is needed with equal functionality. However, even with this radical improvements, a table with $4m$ entries would still need a summary of more than 10 MiB using the parameters given in [19].

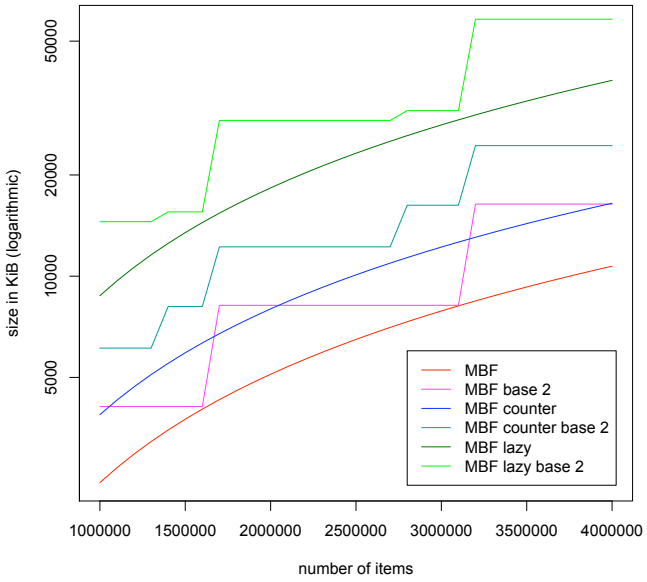


Fig. 20. Memory requirements for different MBF configurations

Fast Hash Table. Decoupling the update and lookup engines for the FHT is not as easy as for the MHT. An entry can only successfully be retrieved by computing the minimum counter value. On first glance, there is little room for optimizing the CBF. One possibility is to limit the allowed maximum counter value to a value smaller than the expected maximum thus specifically allowing more overflown counters. Limiting the counter values allows for better encoding of the summary either in reduction of the counter-width or by using compression. Successful lookup is guaranteed as long as not all counters corresponding to a prefix are overflown, which would not allow to identify the correct bucket. The problem is to find a counter value where the probability of this event is appropriately small.

[17] gives an analysis of the probability that in any $k' < k$ chosen buckets the counter value has a specific height. The derivation of the equation is quite complex and for simplicity left out at this point. Interested readers are referred to the actual paper. We analyze the probability of this event in respect to m and k.

Figure 21 shows the expected smallest counter value in k' chosen counters depending on the size m. Again, the constant c is chosen to divide m by multiples of 2. The table size has significant impact on the smallest counter value. At first glance this may contradict figure 11 which states, that the size has no impact

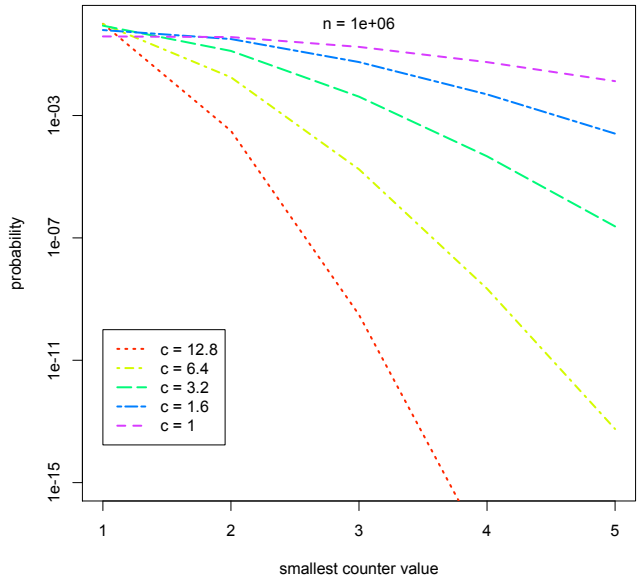


Fig. 21. Probability of smallest counter value in k' counters for different m

on counter values. However, with smaller table sizes, the number of counters to choose from is smaller, hence the higher probability to select higher counters.

The effect of the number of hash functions on the smallest counter value can be seen in figure 22. The result corresponds to figure 12. With a smaller number of hash functions, the choice is limited and thus the probability to select a higher counter is higher. For more than optimal hash functions, the counter distribution is stronger in the range $[1, 4]$ and thus the probability to draw such a counter as smallest value is higher.

Choosing an appropriate value for χ is a tradeoff between storage saved and number of counter overflows. To be able to retrieve all entries the event that all chosen k' counters equal χ must be dealt with. The easiest solution is to move entries which can not be retrieved by calculating the counters to CAM. A small CAM must already be maintained for overflown buckets. If χ is chosen appropriately large the overhead is minimal.

The expected number of entries that are diverted to CAM can easily be calculated. Let $Pr\{C = s\}$ be the probability, that of k' chosen counters the smallest counter has value s and let l be the highest counter value to be expected in the offline summary.

$$E_{CAM} = \sum_{i=\chi}^{l} Pr\{C = i\} \times n \tag{21}$$

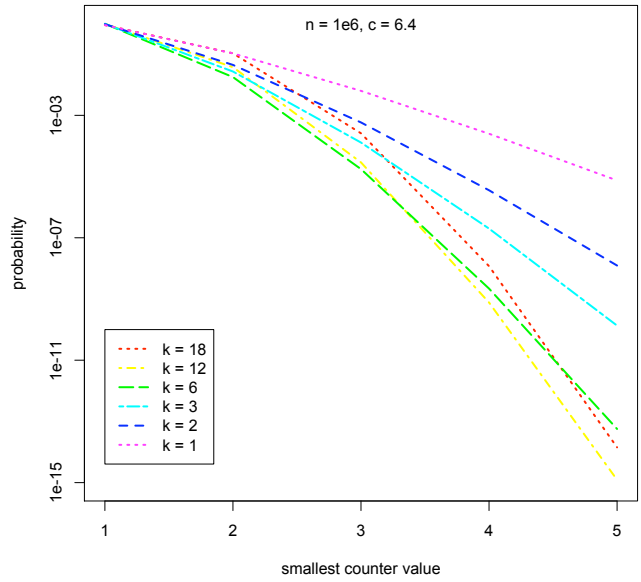


Fig. 22. Probability of smallest counter value in k' counters for different k

The expected number of CAM entries for $c = \{12.8, 6.4, 3.2, 1.6, 1\}$ and $\chi = \{3, 4, 5\}$ can be seen in figure 23. The numbers can be used as a guideline for choosing χ. For example, with $c = 12.8$ and $\chi = 3$, the expected number of CAM entries is still 0. Without any additional cost, the counter-width of the summary can be reduced to 2 bits, achieving a reduction in size of 30%. By further providing a small CAM for few entries, c can be halved, leading to a summary only $\frac{1}{3}$ of the optimum in size. The tradeoff gets better for increasing χ. Consulting the graphs, each time χ is incremented once, c can be reduced by the factor of two, at the cost of few additional CAM entries.

As mentioned, limiting the counter range allows for better optimized encoding or compression of the summary. This is especially useful if the size of the table is reduced to $c < 6.4$. In this case χ must be higher and the counter-width would again be 3 bits. A simple and well known compression scheme is to pack a number of values limited to a certain range into one memory word. For instance, if the counters are limited to a maximum value of $\chi = 5$ and thus a range of $[0, 5]$ then with a 128 bit word-size 49 counters can be encoded, saving 19 bits. To represent 49 counters of said range $6^{49} \ll 2^{128}$ different memory words are needed, leaving the rest of the words unused. In general

$$\gamma_p = \lfloor \frac{log\ 2^b}{log\ |[\chi + 1]|} \rfloor \tag{22}$$

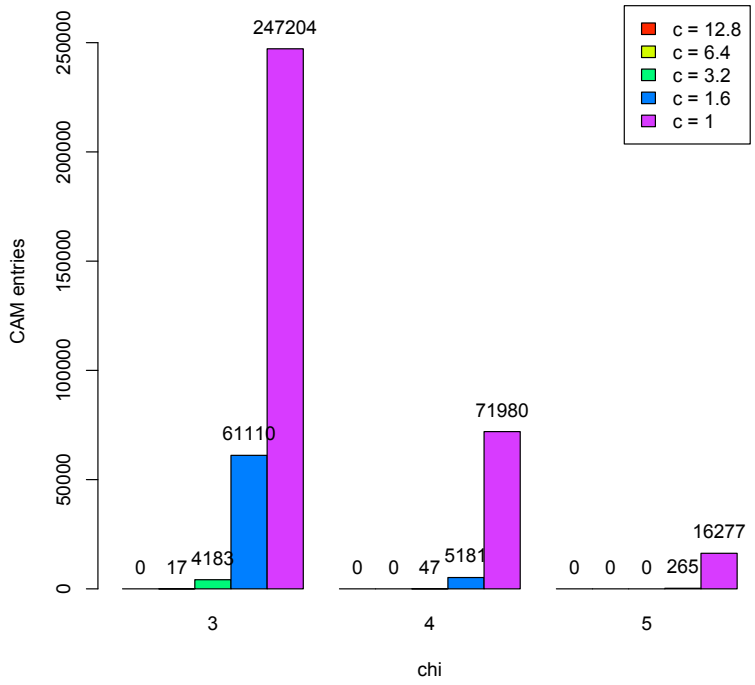


Fig. 23. Expected number of CAM entries for different c and χ

counters can be encoded in a word of b bit size. Compression and decompression is then straightforward. Let ω be the compressed representation of γ_p counters (in the following we will also refer to γ_p as compression rate).

$$\omega = \sum_{i=0}^{\gamma_p - 1} \varsigma_i \cdot |[\chi + 1]|^i \tag{23}$$

A word can be decompressed using pseudocode 1.

Algorithm 1: Word decompression

Data: ω,χ,γ_p
Result: decompressed counters

```
C = ∅ ;
for ∀i ∈ [γ_p] do
    C ← C ∪ ω % χ ;
    ω ← ω/χ ;
end
return C ;
```

The only drawback is the expensive modulo computation to calculate the counters. However, implemented in hardware, all counter values can be decoded in parallel. To prevent confusion with other compression schemes we will refer to this as word packing and name the summary packed counting Bloom filter (PCBF).

The table construction is now four-fold. It is composed of the offline CBF and BFHT, an online on-chip compressed CBF, the online hash table in off-chip memory and a small CAM for overflow entries. The design is depicted in figure 24.

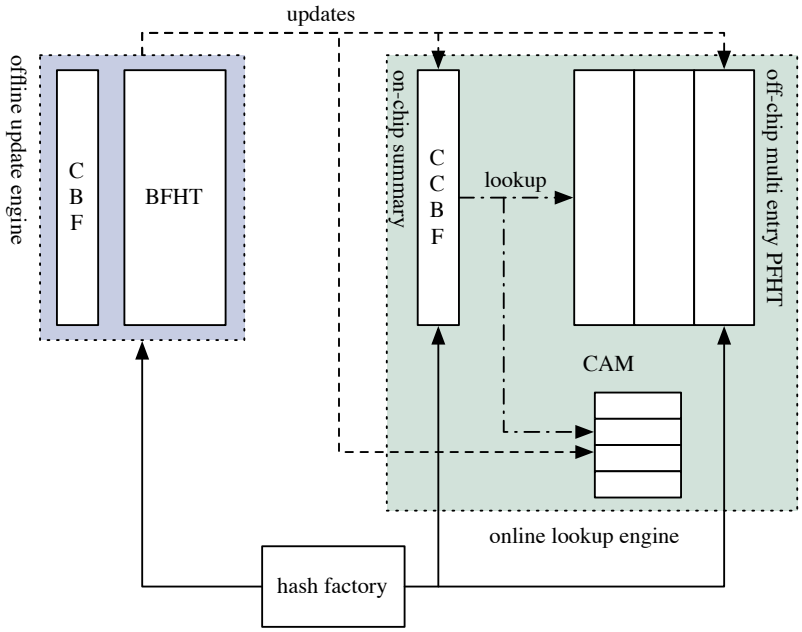


Fig. 24. Memory efficient FHT construction

Strictly, the offline CBF is not needed, the counter values could also be computed by examining the length of the linked list of the affected bucket. However, this would lead to significant overhead especially when entries are considered for relocation. In this case one is only interested in the bucket indexes and the associated counter values of affected entries. Computing the counter values by traversing all buckets to which the entry hashes is too expensive. Thus, we keep an offline CBF for performance reasons.

In [17] the offline BFHT is needed to add deletion support. Inserts are done online by utilizing the PFHT. In our design we want to completely separate updates from lookups to keep interference with the lookup process as small as possible. In the following, when discussing entries, one must distinguish between offline entries, stored in the offline BFHT, and online entries, which are stored in the actual lookup table. The latter has been discussed in section 3.4. When performing updates, the offline table pre-computes all changes and applies them to the online CCBF, table and CAM. The update engine must be able to identify which offline entries in an affected bucket are also online entries, and which are in CAM. Otherwise, it would not be possible to compute relocations. Therefore, for offline entries we must maintain an additional locator, which is simply the index of the hash function used to store the corresponding prefix in the online table. If the entry is not present in the online table but moved to the CAM, the locator is set to ∞. An offline entry thus is defined as $E_{offline}(x) \leftarrow (x, loc, v)$, where loc is the locator for prefix x. Note that we do not keep the verifier and identifier. This would require to keep k offline entries for every stored prefix. Instead we keep only one offline entry for each prefix x and set pointers in the offline table accordingly.

The design and update process is now discussed in more detail. The offline BFHT and CBF are built exactly as in [17]. All verifiers are stored with their ID and associated value and loc is initialized to ∞. Pruning now includes building the online CCBF and PFHT. Algorithm 2 shows the pseudocode of the pruning process. Entries are not removed from the BFHT as in [17], instead a new PFHT is built. All entries in the offline BFHT are examined and their target location in the online PFHT computed. Depending on the maximum counter value χ and the load of target online buckets some entries may be directly put into CAM. Note, that in this case, the offline entry's associated locator is not modified. This is done to identify entries that have been moved to CAM and is evaluated during update processing.

As already mentioned, all updates are preprocessed offline and modifications to the online structures made afterwards. Algorithm 3 shows the pseudocode for insertions. First, two relocation lists are initialized, one to collect affected online entries (R^0) and the second to collect entries from CAM (R^1). R^1 is later appended to R^0 (line 3). That is because online entries must be relocated prior to CAM entries since it is possible that these buckets were filled to the maximum and space becomes available to hold the entries from CAM. The hash values for x are computed, counters retrieved and the target location identified. If all counters are equal to or exceed the maximum allowed value χ, the new entry

Function `prune`

Data: B: offline BFHT, C: offline CBF, χ: maximum counter value, γ: compression rate / counters per compressed word;
Result: P: online PFHT, Z: online CCBF, Y: CAM ;

```
1  begin
2      P, Y ← ∅ ;
3      Z ← Compress(C, χ, γ) ;
4      for b ∈ B do
5          H ← Unique(H(x)) ;
6          ζ ← {C_{h(x)} ∀h ∈ H} ;
7          i ← SmallestIndex(Min(ζ), H(b)) ;
8          if c not ≥ χ ∀c ∈ ζ & SpaceLeft(P_{A(H_i)}) then
9              SetLoc(b, i) ;
10             P_{A(H_i)} ← P_{A(H_i)} ∪ NewEntry(V(H_i), i, Value(b))
11         else
12             Y ← Y ∪ NewEntry(Key(b), Value(b)) ;
13         end
14     end
15 end
```

must be placed into CAM and the locator is set to ∞. Otherwise the entry's locator is set to the index of the hash function used to store x (lines 3 - 3). For all affected buckets those entries, which have a locator equal to the considered bucket's index or ∞ are collected, $V_i(x)$ is inserted and the corresponding counters incremented (lines 3 - 3). After the entries have been collected, they are considered for relocation. Those entries for which the locator does not change are removed from the list. The other entries are modified accordingly (lines 3 - 3). Note, that this modification is assumed to be done by reference and affects all offline representations of these entries. After all changes have been computed, x is stored in it's target (PFHT or CAM), the online structures are updated with the remaining relocation entries and the online counters are incremented. Line 3 performs the update. Entries that need to be relocated are removed from their former position and reinserted in their target location. The online PFHT needs only be accessed to store x and for removal and reinsertion, thus keeping

online access to a minimum. Similarly, the online CCBF is only accessed for those counters that have not yet reached χ.

Function `insert(`x`)`

Data: B: offline BFHT, C: offline CBF, χ: maximum counter value, P: online PFHT, Z: online CCBF, Y: CAM ;
Result: updated tables and summaries, such that they include x;

```
1  begin
2      R^0, R^1, I ← ∅ ;
3      H ← Unique(H(x)) ;
4      ζ ← {C_A(h) ∀h ∈ H} ;
5      i ← SmallestIndex(Min(ζ), H) ;
7      if c not ≥ χ ∀c ∈ ζ & SpaceLeft(P_A(H_i)) then
       e = NewEntry(Key(x), i, Value(x)) ;
9      else e = NewEntry(Key(x), ∞, Value(x)) ;
11     for ∀h ∈ H do
           // collect entries
12         for ∀b ∈ B_A(h) do
13             if Loc(b) = A(h) then R^0 ← R^0 ∪ b ;
14             else if Loc(b) = ∞ then R^1 ← R^1 ∪ b ;
15         end
16         B_A(h) ← B_A(h) ∪ e ;
17         if C_A(h) < χ then I ← I ∪ h ;
19         C_A(h) + + ;
20     end
       // check entries for relocation
22     R ← Append(R^0, R^1) ;
24     for ∀r ∈ R do
25         H' ← Unique(H(x)) ;
26         ζ' ← {C_A(h) ∀h ∈ H'} ;
27         i' ← SmallestIndex(Min(ζ), H') ;
28         if i' ≠ Loc(r) then
29             if SpaceLeft(P_A(H_i')) then SetLoc(r, i') ;
30             else SetLoc(r, ∞) ;
31         else
33             R ← R − r ;
34         end
35     end
       // insert entry into online table
36     if Loc(e) ≠ ∞ then P_A(H_i) ← P_A(H_i) ∪ NewEntry(V(H_i), i, Value(x)) ;
37     else Y ← Y ∪ x ;
39     UpdateOnline(R) ;
40     Increment(Z, I) ;
41 end
```

Deletions work similar to insertions with minor differences. The deleted entry x is removed from the offline BFHT prior to collecting entries. Then all entries in affected buckets buckets are collected and the relocation computed. Afterwards, x is removed from PFHT/CAM, the structures updated and the CCBF counters decremented. Note, that the CCBF counters need only be modified, if they get decremented below χ.

3.6 Huffman Compressed Counting Bloom Filter

Section 3.5 introduced a simple word packing scheme for counting Bloom filters where the counters are packed in memory words. Another form of compressed counting Bloom filters has been proposed by Ficara et al in [11]. Using Huffman compression the CBF is distributed among multiple bitmap layers. Unfortunately, computing the counter values is expensive due to the fact that all preceding cells must be evaluated and the bitmaps must be accessed using perfect hash functions. Though the applicability of the ML-CCBF as a CBF replacement for the FHT is not evaluated we assume it is unable to provide the required performance.

In this section we propose another design for compressed counting Bloom filters also based on Huffman compression, which we name Huffman compressed counting Bloom filter (HCCBF). Given a binomial distribution like the CBF counters, Huffman compression produces an optimal encoding. In addition, each symbol is mapped to a prefix free code, allowing individual de-/compression. Huffman codes are easily calculated using a binary tree. The probability of each counter value is computed and the list is sorted by probability. On each iteration the two items with highest probability are aggregated to a parent node with the left child being the higher weighted counter, and the right child the lower. This is repeated until the list contains only one root node. The resulting Huffman tree for counters of range $[0, 4]$ is depicted in figure 25.

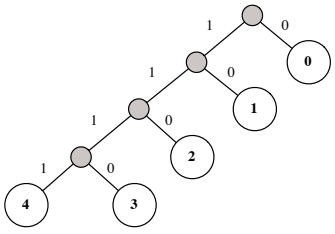


Fig. 25. Example Huffman tree

As in section 3.5 the counters are limited in range. This is done for two reasons. First, the resulting Huffman tree is finite and very small in size. Second, the code bit-length is upper bound to the maximum allowed value χ. This makes

upper bound storage prediction easy and allows further tuning. In addition, it reduces the probability of word overflows, which we will discuss shortly.

The tree can be stored in small dedicated hardware, like a hardware lookup table, or decompression could take place inside the data path. We will also refer to the the Huffman tree as codebook henceforth. To achieve real-time de-/compression the counters must be easily addressable. Storing the compressed counters consecutively is not feasible. Without the help of complex indexing structures one could not retrieve a specific value. Therefore, when compressing the offline CBF we calculate the maximum number of counters γ_h that can be compressed in one memory word, such that each word encodes exactly γ_h counters. The pseudocode for compression is shown in algorithm 4.

Function `compress`

```
   Data: C: offline CBF, χ: maximum counter value, b: word size in bits ;
   Result: Z: online HCCBF, γ_h: compression rate ;
 1 begin
 2     γ_h ← ∞ ;
 3     Z ← ∅ ;
 4     ω ← 0 ;
 5     while i ← 0 < |C| do
 6         if C_i ≥ χ then z ← Code(χ) ;
 7         else z ← Code(C_i) ;
 8         i ← i + 1 ;
 9         if NumberCounters(ω) < γ_h then
10             if (|ω| + |z|) ≤ b then
11                 ω ← Append(ω, z) ;
12             else
13                 γ ← NumberCounters(ω) ;
14                 Z ← ∅ ;
15                 ω ← 0 ;
16                 i ← 0 ;
17             end
18         else
19             Z ← Z ∪ ω ;
20             ω ← z ;
21         end
22     end
23     Z ← Z ∪ ω ;
24 end
```

The algorithm runs as long as not all counters have been processed. It iteratively tries to fit as many counters into a word ω as allowed by the compression rate γ_h which is initialized to ∞. If the bit-length of ω would exceed the word-size, everything is reset and restarted with γ_h set to the last number of counters

in ω. This ensures, that every word (except the last) has exactly γ_h counters encoded and allows easy indexing.

This algorithm has an obvious flaw. It heavily depends on the sequence of counters leading to an unpredictable compression rate γ_h. In addition, the compression is wasteful in storage. Since γ_h depends on the sequence of counter values, it is upper bound to the longest code sequence it can compress in one word. Assume no compression is used, then every counter will occupy three bits, which equals the length of the Huffman code for $c = 2$. Thus, if during compression a long sequence of counters ≥ 2 is found, the compression rate γ_h will degenerate.

A better approach is to define γ_h in advance such that a desired compression is achieved. In general, the Huffman compression only achieves improvement over the word packed compression if $\gamma_h > \gamma_p$. Thus, γ_p can be used as a guideline for choosing γ_h. In the following we will refer to this compression scheme as *hard compression*. This can lead to word overflows, if the compressed γ_h counters do not fit into a word. These overflows could also occur during insertions. If values $< \chi - 1$ are incremented and the compressed word already occupies all the available bits then incrementing the counter will shift one bit out of the word. Thus, the last counter value will not be retrievable. Note, how the counter limit χ affects the probability of word overflows.

There are different approaches of how to address word overflows. One is to simply ignore the overflown counters and assume they have value χ. As long as these counters are not the smallest for any entry, the lookup process is not affected. If, however, the actual counter value is crucial to the lookup, the correct location of an entry can not be computed. This would lead to significant overhead to retrieve the entry and is is obviously an inferior solution. Alternatively, the longest code in the word could be replaced with a shorter overflow code, indicating, that an overflow occurred. However, this would increase the length of nearly all counter codes and in return the probability of word overflows. Probably the best solution is to keep a small extra CAM, or other memory, to store the overflown bits. If counters that are completely or partially overflown must be retrieved, the remaining bits are read from the extra memory. We will show in section 4, that depending on γ_h and χ the cost of additional memory is reasonably small.

With m counters, a compression rate of γ counters per word and an on-chip word-size of $|\omega|$ bits, the summary needs

$$\beta_{eht} = \lceil \frac{m}{\gamma} \rceil \cdot |\omega| \tag{24}$$

bits in total.

3.7 Building Efficient Hash Tables: A Guideline

The previous sections covered various techniques to improve on-chip memory requirements of hash table summaries. The improvements are usually bought

at the cost of additional complexity and off-chip/offline memory. The tradeoff can be optimized by a careful choice of parameters. This section combines the lessons learned to guide the construction of an *efficient hash table*.

Primary point for improvement is to reduce the size m of the summary and the table at the cost of a higher false-positive probability. The size m depends on the factor c which influences the number of buckets and counters reserved for any item. Reducing the size results in higher bucket loads which can be compensated by increasing the off-chip word-size, thus allowing multiple entries per bucket. This number depends on the expected maximum load that appears with high probability. With the use of transformations and bit-extraction, the entry size can be reduced and off-chip memory saved. Bucket overflows are handled by keeping a small amount of CAM to store the overflown entries. To off-load update overhead, we keep separate online lookup and offline update data structures. The online counting Bloom filter's counters are limited in range, given by parameter χ, which depends on the probability of the smallest counter in k' chosen counters. In case all chosen k' counters for an item are χ, the item is stored in the overflow CAM. The range limit allows better encoding by either using word packed filters, or Huffman compression. In case of Huffman compressed filters, the compression factor γ_h, which is the amount of counters compressed in one on-chip memory word, must be chosen such that $\gamma_h > \gamma_p$, which can easily be calculated. The amount of memory that can be saved depends on the on-chip word-size. In case of word overflows, that is, the compressed counters do not fit in one on-chip memory word, the overflow bits are stored in a small dedicated CAM and are extracted on demand. Figure 26 recapitulates the design including a HCCBF.

As a concrete example consider a core router with 1.000 ports and a forwarding table of $n = 1,000,000$ prefixes. 95% of the prefixes have length ≤ 48 bits while the rest has between 49 and 64 bits. We would use Waldvogel's "binary search on prefix lengths" algorithm to do the actual longest prefix matching. The prefixes would then be sorted by length and stored in tables accordingly. For simplicity assume now that all prefixes are stored in one table. If any longer prefixes are inserted they are stored in CAM. We suggest the following configuration.

We choose $c = 1.6$. Using equations 5 and 2 this results in a table size of $m = 2^{21}$ buckets and $k = 2$ hash functions. With prefixes up to 64 bits, the verifier needs up to 43 bits. An additional bit is needed for the identifier and further 10 bits for the outgoing port for a total of 54 bits. Following equation 18 the maximum load is 3. Thus, we use a 96 bit wide DDR off-chip memory to store the entries. We limit the online counters to $\chi = 5$. This gives a probability for selecting χ as the smallest counter for any item of $< 2.6e^{-4}$. Thus, about 260 entries will be stored in CAM. On-chip memory can theoretically be arbitrarily wide, we choose a word-size of $|\omega| = 128$ bits to achieve high compression. Using a PCBF this allows packing $\gamma = 49$ counters into one word. Following equation 24 the summary has a size of 668.75 KiB. Using a HCCBF this size can further be reduced at the cost of a small counter overflow CAM. An evaluation is given in section 4.

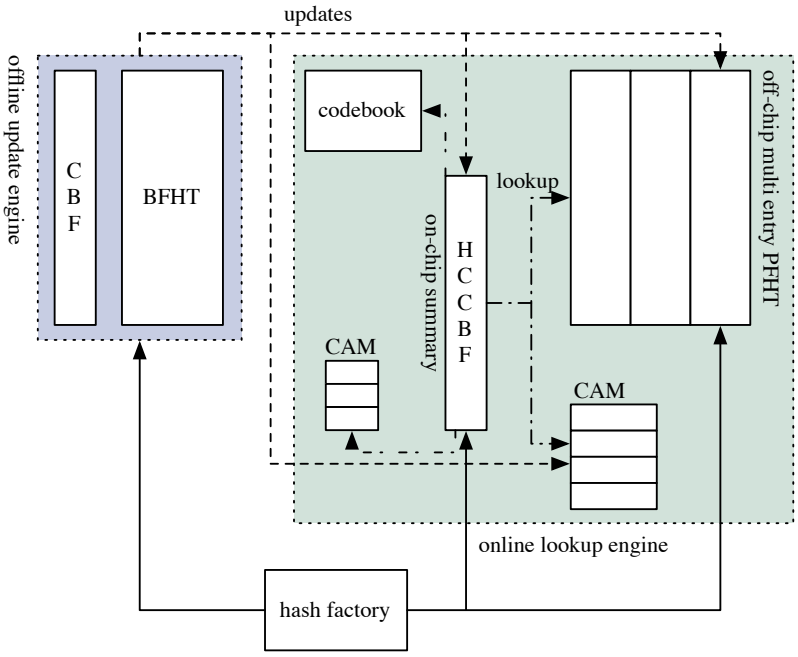


Fig. 26. Huffman compressed Bloom filter as hash table summary

In comparison, the optimal FHT configuration proposed in [17] would use $c_{opt} = 12.8$ for $m_{opt} = 2^{24}$ buckets and $k_{opt} = 12$ hash functions. Assuming counters with 3 bits each the summary needs 6 MiB, nearly 10 times the amount of memory our approach needs. Our *efficient hash table* design achieves improvement over the scheme of Song et al by the factor of 10.

A practical implementation would store the prefixes in different tables according to length. Depending on the number of prefixes in the tables and the size of the entries different configurations may be necessary. It might be impractical to provide an off-chip memory of 96 bits width. Instead a 64 bit wide DDR memory can be used and prefixes with length > 48 bits stored in tables with $c = 3.2$ which would result in a maximum load of 2 entries. The sum of the summaries will then be higher. This is neglected in all previous work, therefore, we ignore the overhead of multiple tables in our evaluations.

Figure 27 shows a size comparison of the summaries discussed in this thesis and our improvements. We use the optimal parameters suggested in the original papers to calculate the sizes. That is, for the MHT summaries, equation 6 is used to define the number of sub-tables. Equation 7 provides the number of bits needed for each of the occupancy and deletion bitmaps. In case of lazy deletions, the bitmap size is added twice to the summary size. In case of counter-based deletions the bitmap size is added only once. For the lazy deletions single filter

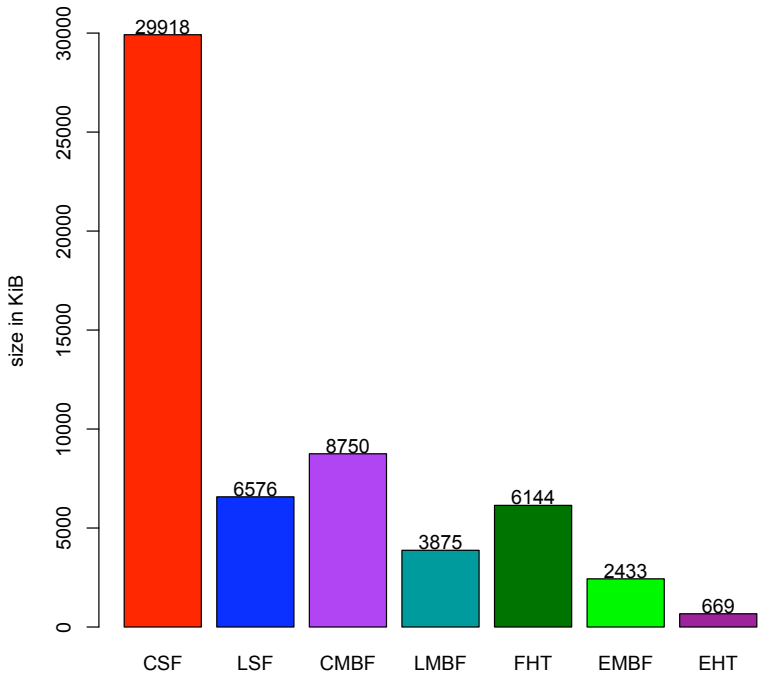


Fig. 27. Summary size comparison for n = 1,000,000

summary (LSF) we use equation 13 to calculate the filter size. The counter-based deletions single filter (CSF) size is derived using equation 15. A counter-width of 2 bits is assumed. For multiple Bloom filter summaries, we use equation 14 for lazy deletions (LMBF). For counter-based deletions (CMBF) equation 16 is used. The bar labelled EMBF refers to the summary suggested in section 3.5 and is simply the size of a MBF summary without update overhead. The number of counters needed for the FHT is calculated using equation 5 with $c = 12.8$. The result is multiplied with a counter-width of 3. EHT refers to our efficient hash table design using the parameters stated above.

A cost function can now be defined as follows. Let α_S be a constant cost factor of on-chip memory, α_D the equivalent for off-chip memory, w the width of off-chip memory in bits, E_o the expected number of bucket overflows and α_C the cost of CAM cells.

$$f_{EHT} = \alpha_S \times \beta_{eht} + \alpha_D \times (m \cdot w) + \alpha_C \times (E_{CAM} + E_o). \tag{25}$$

Depending on the costs of the components the parameters for the EHT can be chosen such that the total cost is minimized.

4 Discussion and Results

In this section we present and discuss results of a conceptual implementation of the EHT. The implementation is conceptual in the sense that it does *not* fully resemble the complex structure of the EHT but simulates it's behavior appropriately.

For simulations we use the following parameters:

$$n = \{100,000; 1,000,000\}; c = \{3.2; 1.6\}; \chi = \{4; 5\}; |\omega| = \{64; 128\}$$

for a total of 16 different simulations. The number of hash functions k is always chosen optimal, using normalization as mentioned in section 3.3. When referencing the different parameter configurations we use the following table.

ref	n	c	k	χ	$\lvert\omega\rvert$
0	1,000,000	1.6	2	4	128
1	1,000,000	1.6	2	4	64
2	1,000,000	1.6	2	5	128
3	1,000,000	1.6	2	5	64
4	1,000,000	3.2	3	4	128
5	1,000,000	3.2	3	4	64
6	1,000,000	3.2	3	5	128
7	1,000,000	3.2	3	5	64
8	100,000	1.6	2	4	128
9	100,000	1.6	2	4	64
A	100,000	1.6	2	5	128
B	100,000	1.6	2	5	64
C	100,000	3.2	4	4	128
D	100,000	3.2	4	4	64
E	100,000	3.2	4	5	128
F	100,000	3.2	4	5	64

Table 1. Parameter configurations.

On each simulation we do ten trials, that is we instantiate the EHT and fill it with n random keys and values. The structure is then pruned using algorithm 2. No updates are performed but the EHT is queried for all n and additional $2n$ random keys to verify that every key can be retrieved and to analyze the false-positive probability. Simulations are very expensive in software, that is why only ten trials are performed. This may seem few but confidence is pretty high and it is assumed that the ten trials are representative for a larger set of tables. As summary a HCCBF is used. The compression rate γ_h is calculated using algorithm 4. No hard compression is used, since we want to evaluate the quality of the compression algorithm. The cost of using hard compression can be derived by examining the resulting HCCBF and is included in the analysis.

For each try we calculate the size of the offline CBF, the size of a PCBF and the size of the online HCCBF. We count the frequency of all counters in the offline summary which allows to derive the number of overflown counters in the online summary. Every compressed word in the HCCBF is analyzed for the number of bits that are actually used to encode counters, resulting in a histogram of code-lengths per word. In addition the load of all online buckets is calculated and the number of CAM entries counted. We are especially interested in the effect of the on-chip word-size $|\omega|$ and the counter limit χ on compression quality and CAM size to evaluate the tradeoff. Not all the results will be presented since this would go beyond the scope of this thesis. Instead we concentrate on expected and obscure behavior and present these with selected examples.

The analysis follows the outline in section 3. We begin by evaluating the counter distribution and bucket loads. We proceed with an analysis of the effect of χ and the number of CAM entries. Then the achieved compression rate is discussed and a tradeoff for hard compression elaborated. Last, we evaluate the sizes of the summaries and weigh improvements and overhead.

4.1 Counter distribution

Since the parameters χ and $|\omega|$ have no effect on the counter distribution we count the frequencies of the counters for $n = \{1E+6, 1E+5\}$ with sizes of $c = \{1.6, 3.2\}$. Using equation 17 presented in section 3.3 we also calculate the expected frequency for each counter value. The probability of the counter values is multiplied with the number of counters m to derive the expected frequency.

The results are shown in figure 28. The actual frequencies resemble the expected frequencies exactly. Thus, the structure behaves as predicted. The number of counter overflows in the online summary depends on χ and can be seen in table 2.

	configuration			
χ	[0]-[3]	[4]-[7]	[8]-[B]	[C]-[F]
4	34106.85	25868.12	2033.25	4068.850
5	6277.50	3594.20	301.15	602.225

Table 2. Number of counter overflows.

It is interesting that for $n = 1E+6$ the number of overflows is smaller for the larger table while for $n = 1E+5$ the frequencies are reversed. However, against first assumptions, this is expected and can easily be explained. Recall from section 3.3 that the counter distribution depends on the fraction $\frac{m}{n}$ and the number of hash functions k. To explain this behavior, an analysis of these factors seems appropriate. Table 3 shows the number of choices k and counters/buckets per item for $n = \{1E+6, 1E+5\}$ and $c = \{1.6, 3.2\}$.

As can be seen tables with $n = 1E+5$ have an advantage in both the number of choices and the number of counters/buckets per item. Especially for

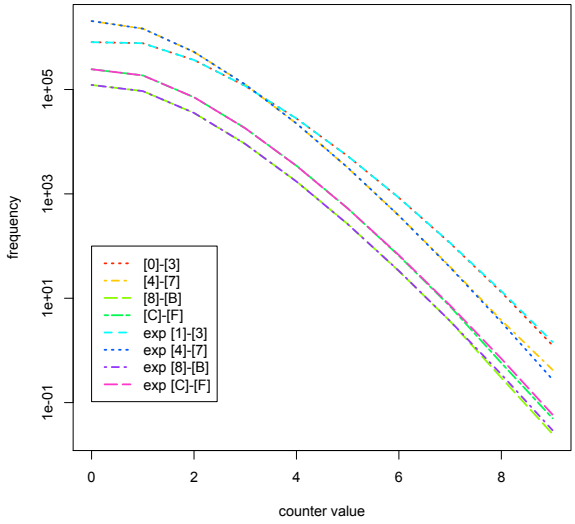


Fig. 28. Real and expected counter frequencies

n	c	k	m/n
$1E+06$	1.6	2	2.097152
$1E+05$	1.6	2	2.621440
$1E+06$	3.2	3	4.194304
$1E+05$	3.2	4	5.242880

Table 3. Number of choices and buckets per item.

$c = 1.6$ the number of hash functions k is equal, but the fraction $\frac{m}{n}$ is higher for $n = 1E+5$. Hence, the probability that a counter receives more insertions is higher for $n = 1E+6$, explaining the numbers in table 2.

We will study the impact of the number of overflows on the number of CAM entries in section 4.3.

4.2 Bucket Load

The maximum load depends on the number of choices k and the number of items n. We aggregate the results of the combinations for n and c and count the number of entries in every online bucket. We then take the maximum of the frequencies to evaluate the worst-case behavior. The results are shown in table 4.

For all the tables with $n = 1E+6$ there was one bucket overflow in the worst-case. That is, only one entry will be diverted to CAM. None of the buckets for

configuration	$E_{maxload}$	load 0	1	2	3	4
$[0]-[3]$	3	1184464	837562	80950	684	1
$[4]-[7]$	2	3204894	980039	10438	1	0
$[8]-[B]$	3	167662	89728	5327	24	0
$[C]-[F]$	2	424659	99411	369	0	0

Table 4. Entry distribution.

tables with $n = 1E + 5$ experienced an overflow. With these results the best solution to deal with bucket overflows is to artificially increase the counter of the affected bucket to χ and move the entries to CAM. This can be performed during pruning and by the update engine which adjusts the counters accordingly.

According to the numbers, the bucket loads do not scale with n. Compared to tables with $n = 1E+5$ the number of entries per bucket for $n = 1E+6$ are lower for 0 entries, and much higher for > 1 entries. Like the counter distribution, bucket load distribution also depends on the fraction $\frac{m}{n}$ and the number of choices k. In the previous section we have shown that tables with $1E + 5$ have an advantage in both factors over tables with $1E + 6$ (see table 3). This also explains the behavior in table 4.

4.3 The power of χ

The counter limit χ bears a variety of effects. It affects the number of counter overflows in the online summary which have been evaluated in section 4.1. Depending on the probability of the smallest counter value in k' chosen counters it also influences the number of entries that must be stored in CAM. Finally, the achievable compression rate γ directly depends on χ.

We will start by evaluating the number of CAM entries and then move onward to inspecting the achieved compression.

CAM requirements. For CAM entries we aggregate the results for χ according to n and c, calculate the average and take the minimum/maximum values encountered. Following equation 21 we also calculate the expected number of CAM entries. Table 5 shows the results.

Once again, the results closely resemble the expectations. It can be observed, that the numbers do not scale with n. We have explained the reasons in previous sections and a recapitulation is omitted. However, one interesting fact can be extracted by analyzing table 5. That is, the quality of χ also depends on the fraction $\frac{m}{n}$. This does not come as a surprise, since with higher counters in general the probability to choose a higher counter as smallest counter value for any item is also increased. Still, it is worth mentioning. For tables with a sufficiently large number of counters/buckets per entry (like $n = 1E+5$,$c = 1.6$) χ can be set to 4 without much overhead. Consulting table 3 allows a rough estimate of the lower bound somewhere in $2 > \frac{m}{n} > 2.6$.

n	c	χ	min	max	avg	expected
$1E+6$	4	1.6	5017	5446	5194.05	5181
$1E+6$	5	1.6	236	287	258.20	265
$1E+6$	4	3.2	40	61	47.00	47
$1E+6$	5	3.2	0	0	0.00	0
$1E+5$	4	1.6	144	209	177.95	178
$1E+5$	5	1.6	2	11	6.05	6
$1E+5$	4	3.2	0	1	0.15	0
$1E+5$	5	3.2	0	0	0.00	0

Table 5. Number of CAM entries.

It can be expected at this point, that the achieved compression is more effective for tables with higher $\frac{m}{n}$. The next section analyses compression quality in detail.

Compression. To analyze the achieved compression we take the minimum, maximum and average γ_h and compare that to γ_p and the number of counters if no compression is used (denoted γ_0). We also include the maximum number of bits used to compress the counters. Table 6 shows the results.

n	c	χ	$\lvert\omega\rvert$	min γ_h	max γ_h	avg γ_h	γ_p	γ_0	max bits
$1E+6$	1.6	4	64	22	24	22.8	27	21.3	63.3
$1E+6$	1.6	5	64	21	22	21.5	24	21.3	63.3
$1E+6$	1.6	4	128	50	53	51.0	55	42.6	126.4
$1E+6$	1.6	5	128	47	51	49.5	49	42.6	125.1
$1E+6$	3.2	4	64	23	26	24.6	27	21.3	62.7
$1E+6$	3.2	5	64	24	25	24.9	24	21.3	63.2
$1E+6$	3.2	4	128	56	59	57.7	55	42.6	126.3
$1E+6$	3.2	5	128	55	58	56.9	49	42.6	126.3
$1E+5$	1.6	4	64	25	27	26.0	27	21.3	62.6
$1E+5$	1.6	5	64	24	26	25.4	24	21.3	62.5
$1E+5$	1.6	4	128	57	60	58.8	55	42.6	126.6
$1E+5$	1.6	5	128	55	60	57.8	49	42.6	125.7
$1E+5$	3.2	4	64	23	26	25.5	27	21.3	63.0
$1E+5$	3.2	5	64	23	26	24.6	24	21.3	62.1
$1E+5$	3.2	4	128	57	60	58.3	55	42.6	126.9
$1E+5$	3.2	5	128	56	59	57.0	49	42.6	125.8

Table 6. Compression rate.

The numbers provide a lot of useful information. With sufficiently large $|\omega|$ or larger χ, Huffman compression always performs better than word packing, even without using *hard compression*. If $|\omega|$ is small and χ is also small, word

packing is the better choice. The only exception to this rule is for tables with $n = 1E + 6$ and $c = 1.6$. However, we have already seen that these have to be treated differently and we will ignore them for now. In all cases, compression yields an improvement over not using compression. The counter limit χ only slightly influences the compression rate γ_h. It's impact on γ_p is greater by far.

We will now analyze the impact of a small $\frac{m}{n}$ on the HCCBF and show possible improvements at the cost of additional CAM. As examples we choose tables [2] for $n = 1E+6$ and [A] for $n = 1E+5$, because $|\omega|$ and χ are sufficiently large and Huffman compression does not scale for [2].

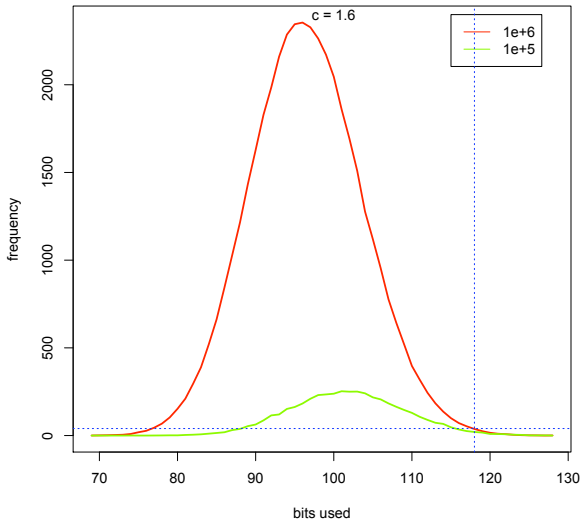


Fig. 29. Frequency of used bits for n = 1e6, c = 1.6

Figure 29 shows the frequencies of the number of bits per word used to compress the counters. The distribution again follows a binomial distribution, which is to be expected. The two graphs can not be directly compared. However, since all parameters except n are equal, it can be concluded, that for smaller $\frac{m}{n}$ and equal k, the center of the distribution moves to the left, leading to a smaller average bit usage. Considering the counter distribution, which also leads to higher counters for smaller $\frac{m}{n}$ and equal k, this is a probable explanation.

The compression can be improved by reducing $|\omega|$ while keeping the same γ_h, thus resembling hard compression. For example, by reducing $|\omega|$ to 118 bits, 10 bits per word can be saved (see the blue lines in figure 29). Of course, this leads to a number of word overflows. In this case approximately 160 words will overflow

with up to 10 bits. Thus by providing CAM for an additional 160 overflown words, 10 bits per on-chip memory word can be saved.

4.4 Comparing sizes

So far we have analyzed and explained the effect of the different parameters on the performance of the EHT. It is now time to present the resulting sizes of the summary. Since we already proved that the results perform as expected no open questions should remain. Therefore we omit a discussion of the summary sizes.

The figures present the average sizes for all simulations made. They include the size of uncompressed filters (CBF), the packed filters (denoted P) and Huffman compressed filters (denoted H) for each χ and word-size $|\omega|$ grouped by the number of items n and the size of the table.

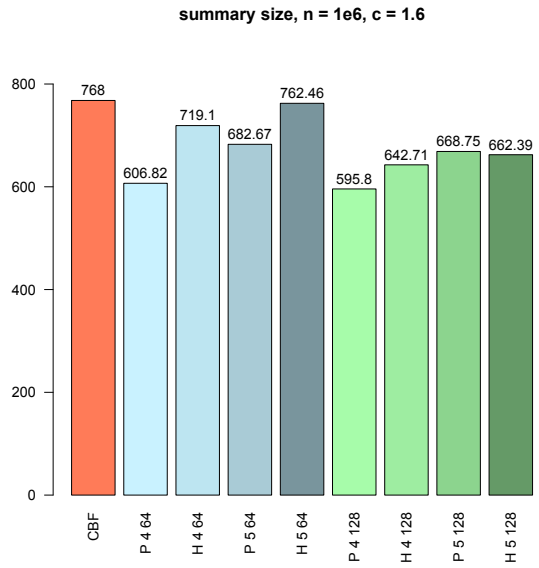


Fig. 30. Summary sizes for n = 1e6, c = 1.6

4.5 Summary

The results fully meet the expectations and backup our theoretical analysis. We have shown that our initial assumptions allow fundamental improvements over previous work. In conclusion, when constructing an EHT, the following aspects must be considered.

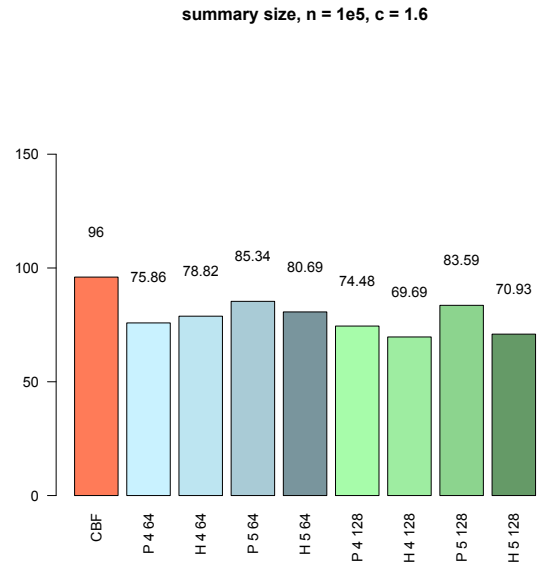


Fig. 31. Summary sizes for n = 1e5, c = 1.6

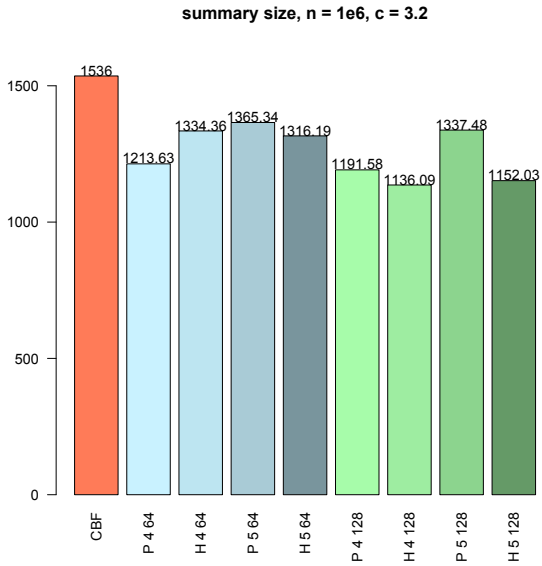


Fig. 32. Summary sizes for n = 1e6, c = 3.2

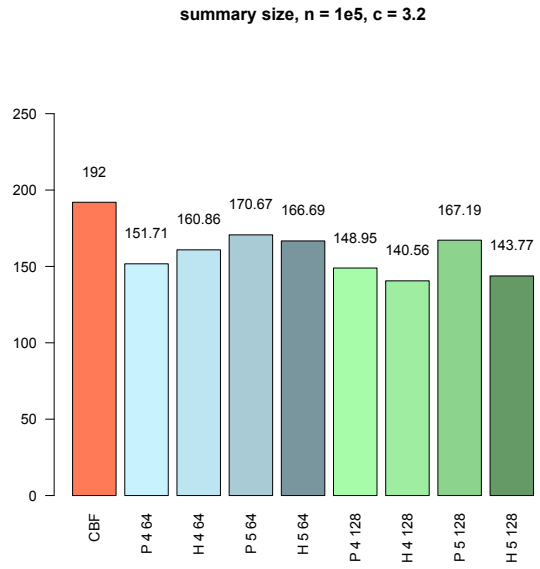


Fig. 33. Summary sizes for n = 1e5, c = 3.2

- Reducing the size m is achieved by increasing the off-chip memory width. Analysis has shown, that the expected maximum load will not exceed 3 as long as $\frac{m}{n} > 2$. Bucket overflows are extremely rare, even for a large set of items. The off-chip memory width can be reduced at the cost of additional CAM.
- Performance does not scale with n. With equal k but smaller $\frac{m}{n}$, performance will be worse. This holds especially if table sizes are very small such that $\frac{m}{n} \to 2$.
- Choosing χ depends on the fraction $\frac{m}{n}$. Starting with $\chi = 5$ for $2 < \frac{m}{n} < 2.5$, χ can be decremented by one each time $\frac{m}{n}$ is doubled for a small overhead in terms of CAM.
- Huffman compression is favorable over word packed compression, unless the word-size $|\omega|$ and the counter limit χ are small.
- At the cost of few additional CAM cells, the performance of Huffman compression can be improved.

5 Conclusion and Future Outline

In the present thesis we have proven that by eliminating unnecessary overhead on-chip memory requirements can be significantly reduced and the lookup performance improved. Based on four key ideas we have introduced new techniques to design an efficient hash table specialized on lookup applications. The results backup our theoretical analysis and allows accurate predictions. A cost function has been provided that enables manufacturers to individually adjust the design and optimize production cost.

5.1 Contributions

Despite being ten years old, IPv6 is still rarely in use due to its high requirements on the hardware. We have proposed several key ideas that lead to a relaxation of these requirements. The cost in terms of complexity and additional components is minimal compared to the achievable gain.

By ignoring the false positive probability which has no impact on lookup performance the hash table summaries can be optimized for size. We have shown, that high amounts of on-chip memory can be traded in for comparatively small amounts of off-chip memory and additional CAM. Clever chosen hash functions allow the reduction of off-chip memory size. Offloading update overhead to offline structures leads to a more optimized lookup engine and allows better encoding. We proposed two compression schemes for the summary that provide real-time performance and are easy to implement. Combined, the presented design achieves an improvement over previous designs by an order of magnitude.

5.2 Future Outline

Some issues have not been addressed in this thesis which will now be discussed.

Updates. We did not evaluate the behavior of our data structure during updates. Thus, no evidence exists to prove that the performance is unaffected by incremental updates. Based on the fact, that the presented results fit the theoretical analysis nearly perfectly we believe that updates pose no problem. The strict separation of update and lookup engine further allows efficient restructuring if necessary. Still, evaluation of update performance is a pending issue.

Predicting compression. The analysis has shown that the number of choices and counters/buckets per item influence the counter distribution and in return affects the compression quality. Though we discussed that fact in detail, we still lack a lower bound for $\frac{m}{n}$ where quality degenerates. The results are still impressive even for small $\frac{m}{n}$, but a lower bound is desirable. We also omitted probability prediction for γ_h. Being able to predict the expected compression rate would allow better optimization of parameters and eliminate the need for empirical analysis.

ML-CCBF. The multilayer compressed counting Bloom filter was proposed shortly before we concluded our research. Therefore, we were not able to analyze whether it could be adopted to our design. The ML-CCBF performs better than our proposed PCBF/HCCBF in terms of compression. However, calculating the counters is expensive or requires index structures for improvement. Combining the simplicity of our summaries with the compression performance of the ML-CCBF does seem compelling and deserves further work.

Hardware implementation. The efficient hash table is destined to be implemented in hardware. By utilizing FPGAs to implement a working prototype its performance under real-world conditions could be evaluated. Open projects like the NetFPGA pose a suitable platform.

5.3 Final thoughts

This thesis has made efficient IPv6 core routing more probable. Hopefully the ideas posed herein will be adapted in future products and not collect dust in the university archives. Networks have always been my passion and being able to shape the future of networking is thrilling. I would be pleased to receive feedback or requests regarding my work.

A Acknowledgement.

At this point I would like to thank all the people who allowed me to finish my studies.

Special thanks go to my supervisor and advisor Prof. Dr. Marcel Waldvogel, who not only allowed me to write this thesis at his chair but also granted me invaluable insight into his vast knowledge. Knowing that I am a single parent he supported me on numerous occasions. Tribute also to Prof. Dr. Marc H. Scholl, who accepted to assess my work and who is always willing to listen to problems.

My fellow student Sebastian Kay Belle provided support and did prove-reading. Moreover, he took responsibility for some of my duties in the final steps of my writing. All without hesitation and despite the fact that he himself was busy writing his Master Thesis. The world needs more people like him.

I would also like to express my deepest gratitude to my father Rolf Zink and his significant other Martina Kriebel. They took care of my son when time was precious. Without them, I would not have been able to finish my studies. Many thanks also to my mother Eva-Maria Zink, without her I would not roam the world.

Last, but not least, I want to thank my son Elaya Valerian Zink. He is the reason for my being in Konstanz and the force that drives me onwards. He accepted times of my absence with great endurance. Now that the thesis is finished I will make amends for the times lost.

References

[1] M. A. Ruiz-Sanchez, E. W. Biersack, and W. Dabbous, "Survey and taxonomy of ip address lookup algorithms," *Network, IEEE*, vol. 15, no. 2, pp. 8–23, 2001. 2.1

[2] M. Waldvogel, G. Varghese, J. Turner, and B. Plattner, "Scalable high speed IP routing table lookups," in *Proceedings of ACM SIGCOMM*, pp. 25–36, Sept. 1997. 2.1

[3] S. Dharmapurikar, P. Krishnamurthy, and D. E. Taylor, "Longest prefix matching using bloom filters," in *SIGCOMM '03: Proceedings of the 2003 conference on Applications, technologies, architectures, and protocols for computer communications*, (New York, NY, USA), pp. 201–212, ACM, 2003. 2.1

[4] A. Broder and M. Mitzenmacher, "Network applications of bloom filters: A survey," in *Internet Mathematics*, vol. 1, pp. 485–509, 2002. 2.2

[5] B. H. Bloom, "Space/time trade-offs in hash coding with allowable errors," *Communications of the ACM*, vol. 13, no. 7, pp. 422–426, 1970. 2.2

[6] L. Fan, P. Cao, J. Almeida, and A. Broder, "Summary cache: A scalable wide-area web cache sharing protocol," in *Proceedings of ACM SIGCOMM*, pp. 254–265, Sept. 1998. 2.2

[7] C. Estan and G. Varghese, "New directions in traffic measurement and accounting: Focusing on the elephants, ignoring the mice," *ACM Transactions on Computer Systems*, vol. 21, pp. 270–313, Aug. 2003. 2.2

[8] S. Cohen and Y. Matias, "Spectral bloom filters," in *SIGMOD '03: Proceedings of the 2003 ACM SIGMOD international conference on Management of data*, (New York, NY, USA), pp. 241–252, ACM, 2003. 2.2

[9] B. Chazelle, J. Kilian, R. Rubinfeld, and A. Tal, "The bloomier filter: an efficient data structure for static support lookup tables," in *SODA '04: Proceedings of the fifteenth annual ACM-SIAM symposium on Discrete algorithms*, (Philadelphia, PA, USA), pp. 30–39, Society for Industrial and Applied Mathematics, 2004. 2.2

[10] M. Mitzenmacher, "Compressed bloom filters," in *Proc. of the 20th Annual ACM Symposium on Principles of Distributed Computing*, IEEE/ACM Trans. on Networking, pp. 144–150, 2001. 2.2

[11] D. Ficara, S. Giordano, G. Procissi, and F. Vitucci, "Multilayer compressed counting bloom filters," *INFOCOM 2008. The 27th Conference on Computer Communications. IEEE*, pp. 311–315, April 2008. 2.2, 3.6

[12] Y. Azar, A. Z. Broder, A. R. Karlin, and E. Upfal, "Balanced allocations," in *SIAM Journal on Computing*, pp. 593–602, 1994. 2.3, 3.3

[13] M. D. Mitzenmacher, *The Power of Two Choices in Randomized Load Balancing*. PhD thesis, Harvard University, 1996. 2.3

[14] B. Vöcking, "How asymmetry helps load balancing," in *In Proceedings of the 40 th IEEE-FOCS*, pp. 131–140, 1999. 2.3

[15] A. Broder and M. Mitzenmacher, "Using multiple hash functions to improve IP lookups," in *IEEE INFOCOM*, 2001. 2.3, 2.3

[16] F. Bonomi, M. Mitzenmacher, R. Panigrahy, S. Singh, and G. Varghese, "An improved construction for counting bloom filters," 2006. 2.3, 3.4

[17] H. Song, S. Dharmapurikar, J. Turner, and J. Lockwood, "Fast hash table lookup using extended bloom filter: An aid to network processing," in *SIGCOMM '05*, (New York, NY, USA), pp. 181–192, ACM Press, 2005. 2.3, 3, 3.1, 3.2, 3.3, 3.3, 3.5, 3.5, 3.7

[18] L. J. Carter and M. N. Wegman, "Universal classes of hash functions," in *Proceedings of the ninth annual ACM symposium on Theory of computing*, 1977. 2.3, 3.1

[19] A. Kirsch and M. Mitzenmacher, "Simple summaries for hashing with choices," *IEEE/ACM Trans. Netw.*, vol. 16, no. 1, pp. 218–231, 2008. 2.3, 2.3, 3, 3.1, 3.2, 3.5

[20] A. Z. Broder and A. R. Karlin, "Multilevel adaptive hashing," in *SODA '90: Proceedings of the first annual ACM-SIAM symposium on Discrete algorithms*, (Philadelphia, PA, USA), pp. 43–53, Society for Industrial and Applied Mathematics, 1990. 2.3

[21] X. Hu, X. Tang, and B. Hua, "High-performance ipv6 forwarding algorithm for multi-core and multithreaded network processor," in *PPoPP '06: Proceedings of the eleventh ACM SIGPLAN symposium on Principles and practice of parallel programming*, (New York, NY, USA), pp. 168–177, ACM, 2006. 3.4

[22] "Ipv6 report." http://bgp.potaroo.net/index-v6.html. 3.4, 3.4