Packet Forwarding using Efficient Hash Tables

Thomas Zink Marcel Waldvogel
University of Konstanz

April 3, 2009

Abstract

This report discusses our proposed improvements to Fast Hash Tables (FHT) which we name 'Efficient Hash Table' (EHT) where 'efficient' relates to both memory efficiency and lookup performance. The mechanism we use to design the EHT lead to improvements in terms of sram memory requirements by the factor of ten over the FHT. Our results back the theoretical analysis and allow accurate predictions. A cost function is provided that allows the adjustment of EHT parameter to different user requirements.

Contents

1 Introduction

Analysis of previous work has shown that IPv6 packet forwarding is still a major bottleneck especially in the internet core. State-of-the-art data structures have high on-chip memory requirements that can not be provided for extremely big routing tables. See section 3.1 in [1] for a detailed analysis. By eliminating unnecessary restrictions these memory requirements can be reduced by an order of magnitude at reasonable costs of additional complexity and off-chip memory. The improvements are based on observations of contemporary solutions and extraction of four key-ideas which are discussed in the following sections. The resulting Efficient Hash Table (EHT) is an evolution of the Fast Hash Table approach introduced in [2] with our key ideas applied. It provides better performance and requires much less on-chip memory.

2 State-of-the-art

This section is an excerpt of section 2 in [1]. We concentrate on a short review of the FHT and give a brief introduction to other solutions. For more information, please refer to [1].

2.1 Fast Hash Table

In [2] Song et al. present a data structure named *fast hash table* (FHT) that uses a counting Bloom filter (CBF) summary in on-chip memory to identify the target bucket of an item. Each counter corresponds to a bucket in the hash table and represents the number of items hashed into it. They use k universal hash functions to access both the CBF and the hash table. With n items the number of buckets /counters m is derived using the following equation.

$$m = 2^{\lceil \log c\ n \rceil} \tag{1}$$

The constant c is the number of items per hash entries per item to be stored (which also equals the number of SRAM counters per item) and is chosen optimally as $c = 12.8$ as follows.

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

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.

2.1.1 Basic Fast Hash Table

In the basic FHT (BFHT) items are simply inserted k times, once in every location it hashes to. The corresponding counters are incremented. 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.

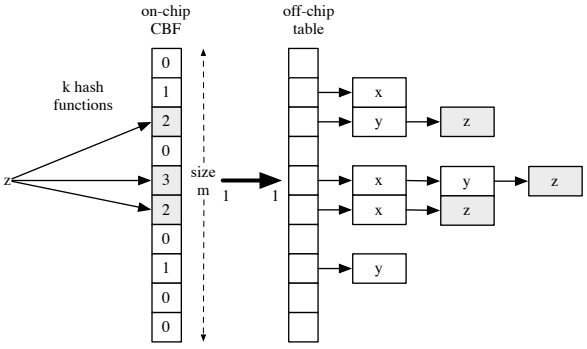

Figure 1: Basic fast hash table

2.1.2 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.

2.2 Other Solutions

Kirsch and Mitzenmacher [3] 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 [4], to store the items. Three summary structures are introduced. The first is an interpolation search summary using a bit string to represent each item. The second summary is a single Bloomier filter which encodes the type of each item to

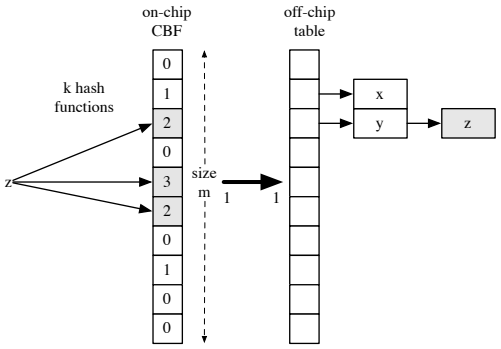

Figure 2: Pruned fast hash table

allow identification of the sub-table it is stored in. The last presented summary is a multiple Bloom filter summary where there is one Bloom filter to represent the set of items which are stored at least in the corresponding sub-table. The Multilayer Hash Table and all the summaries are deeply discussed in section 2.3 in [1].

3 Efficient Hash Tables

Two conclusions can be made by observing modern hash tables and their summaries.

- Big summaries are used to optimize the false positive probability.
- Update support adds significant overhead.

The following sections show how the conditions of IP Lookup applications can be exploited to optimize hash tables and summaries.

3.1 Key Ideas

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. □

The key ideas and the Efficient Hash Table design are discussed deeply in section 3 of [1]. The following sections give a short introduction into our research.

3.2 Ignoring the false positive probability

The major reason for having relatively large Bloom filters is to minimize the false positive probability. 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.

Analysis has shown that as long as the number of hash functions k is near optimal and the constant c is chosen such that $\frac{m}{n} > 2$ the counter values are not affected by reducing the size m. However, since the optimal number of hash functions is a floating point number the practical k usually leads an overestimate and there are some construction for which the counter distribution does not scale. However, this has only a slight effect on the overall performance.

Reducing the size m affects the bucket loads of the hash table buckets. This can be compensated by providing a wider off-chip memory to allow multiple items per bucket. In general 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 $\frac{1}{8}$th of the optimum. Figure 3 shows the expected maximum load for different table sizes that will occur with high probability.

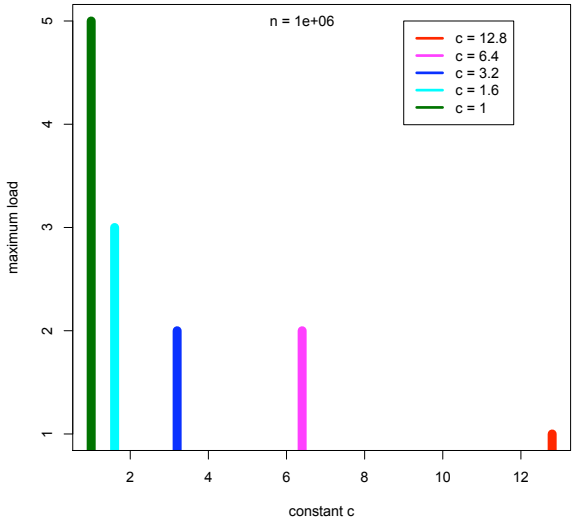

Figure 3: Expected maximum load for different c

An extensive analysis of the effect of reducing the size m is given in section 3.3 of the master thesis.

3.3 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. The expected bucket load and the size of an entry specifies the number of bits needed for the off-chip memory width.

According to [5], [6] 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. 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 prefixes. Longer prefixes can be stored in tables with larger c and thus smaller load, while the very few prefixes > 64 bits can be directly kept in CAM. By using a longest prefix matching algorithm that works on trees of hash tables, like that described in [7], it is also possible to have fixed size prefixes.

The size of an entry can further be decreased by using a hashing scheme similar to that in [8]. 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. The verifier and the index are derived

by bit-extraction. Instead of the prefix only it's verifier is stored in the bucket. To be able to identify which prefix corresponds to a verifier, an identifier must be kept along the verifier, that states the hash function that produced the verifier. Thus with fewer bits it is possible to identify which prefix corresponds to a stored verifier.

One problem remains, that is 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 but still need to be handled. To hold overflown entries a small CAM is reserved. In General, a bucket can only be overflown, if the corresponding counter value at least exceeds the off-chip word-size. In case an overflow occurs, all entries are moved to CAM. On lookup a sentinel value in the CBF summary can be used to identify overflown buckets and the entries be retrieved from CAM. The process is discussed in sections 3.4 and 3.5 of [1].

3.4 Separating the update and lookup engines

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.

An entry can only successfully be retrieved by computing the minimum counter value. The counters can be limited to a value χ 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. 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 CAM entries for $n = 10^6$, $c = \{12.8, 6.4, 3.2, 1.6, 1\}$ and $\chi = \{3, 4, 5\}$ can be seen in figure 4. 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. We will present two compression schemes in the following sections. The table construction is now four-fold. It is composed of an offline update engine which includes a 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 5. In our design we want to completely separate updates from lookups to keep interference with the lookup

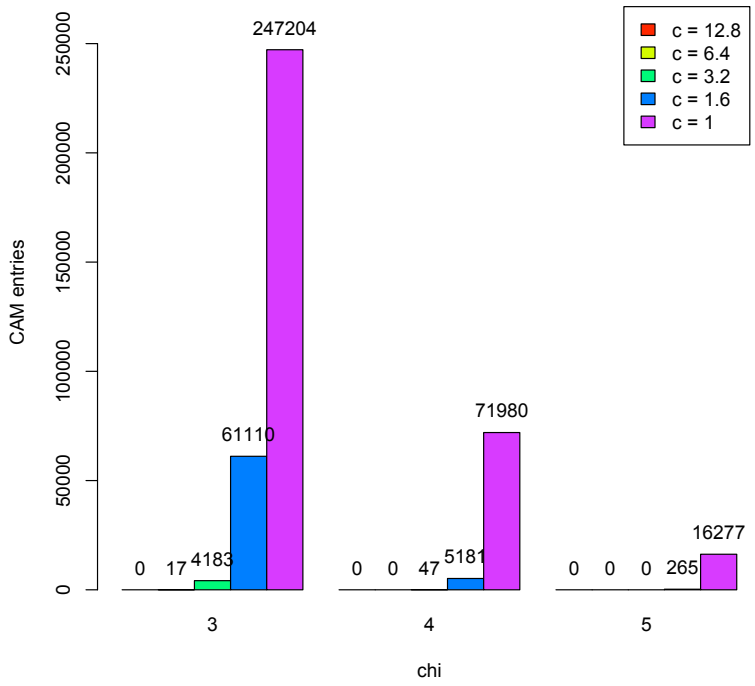

Figure 4: Expected number of CAM entries for different c and χ

process as small as possible. The offline update engine precomputes all changes that occur during updates and generates update vectors for the online CCBF and PFHT. Thus interference with the lookup process is kept to the minimum.

See an extensive analysis and explanation of the update process in section 3.5 in [1].

3.5 Packed Counting Bloom Filter (PCBF)

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. In general

$$\gamma_p = \lfloor \frac{log\ 2^b}{log\ |[\chi + 1]|} \rfloor \tag{3}$$

counters can be encoded in a word of b bit size. In the following γ is referred to as compression rate, that is the number of counters encoded into one word. Let ω be the compressed representation of γ_p counters.

$$\omega = \sum_{i=0}^{\gamma_p - 1} \varsigma_i \cdot |[\chi + 1]|^i \tag{4}$$

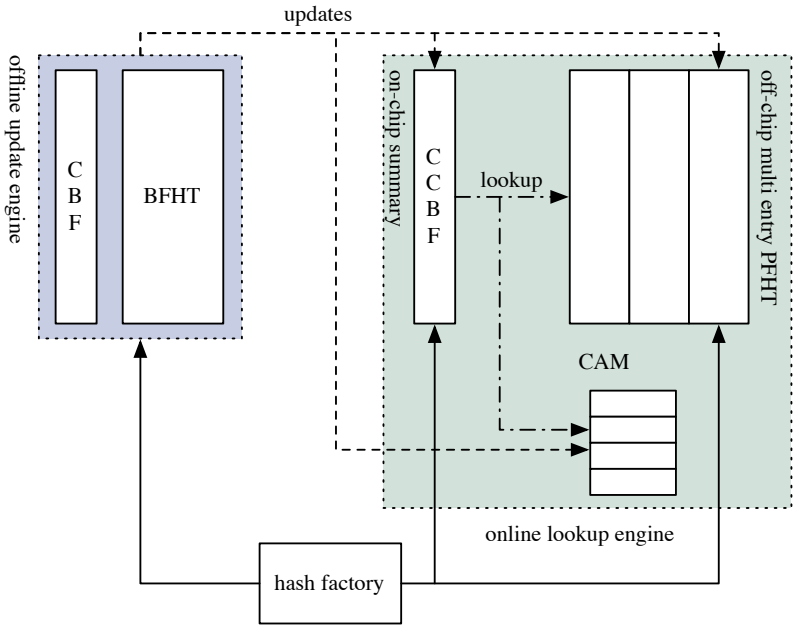

Figure 5: Memory efficient FHT construction

A word can be decompressed using pseudocode 1.

Algorithm 1: Word decompression

Data: ω,χ,γ_p
Result: decompressed counters
$C = \emptyset$;
for $\forall i \in [\gamma_p]$ **do**
  $C \leftarrow C \cup \omega \ \% \ \chi$;
  $\omega \leftarrow \omega / \chi$;
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).

3.6 Huffman Compressed Counting Bloom Filter (HCCBF)

We propose another design for compressed counting Bloom filters 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 tree can be stored in small dedicated hardware, like a hardware lookup table. Decompression could also be done 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. We iteratively try 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. 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*.

Compressing a fixed number of counters into a word can lead to word overflows if the compressed counters do not fit into the word. This can happen during hard compression or updates. There are multiple solutions to deal with word overflows.

(1) Set the overflown word to illegal and keep all counters in extra memory.

(2) Replace longest codes in word with smaller overflow code.

(3) Ignore counter value and assume value χ.

(4) Keep the overflown bits in extra memory and retrieve on demand.

While (1) is a straightforward and easy solution it is wasteful to keep the values of all counters of the affected word in extra memory. (2) Requires relatively short overflow codes to replace the longest counter codes. The counter values can be kept in extra memory. However, since the overflow code must be pretty small the codes for higher counter values will be longer and in effect increase the probability of word overflows. (3) As long as the overflown counter is not the smallest for any item ignoring the value will not affect the lookup process. However, if the affected counter is crucial for the lookup computing the correct location requires additional effort. The best solution to deal with word overflows is (4) and 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

$$\beta_{eht} = \lceil \frac{m}{\gamma} \rceil \cdot |\omega| \tag{5}$$

bits in total.

See section 3.6 of [1] for a complete description of the HCCBF.

3.7 Building Efficient Hash Tables

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 entries of overflown buckets. 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 6 shows a size comparison of modern summaries and our improved EHT. We use the optimal parameters suggested in the original papers to calculate the sizes for summaries for $n = 1.000.000$. For details for the FHT and parameters see [2], for MHT see [3]. A detailed explanation of the chosen parameters can also be found in section 3.7 of [1]. As can be clearly seen our design achieves an improvement over the original FHT by the factor of 10. It also performs much better than any other designs by at least a factor of 5.

4 Results and Discussion

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*

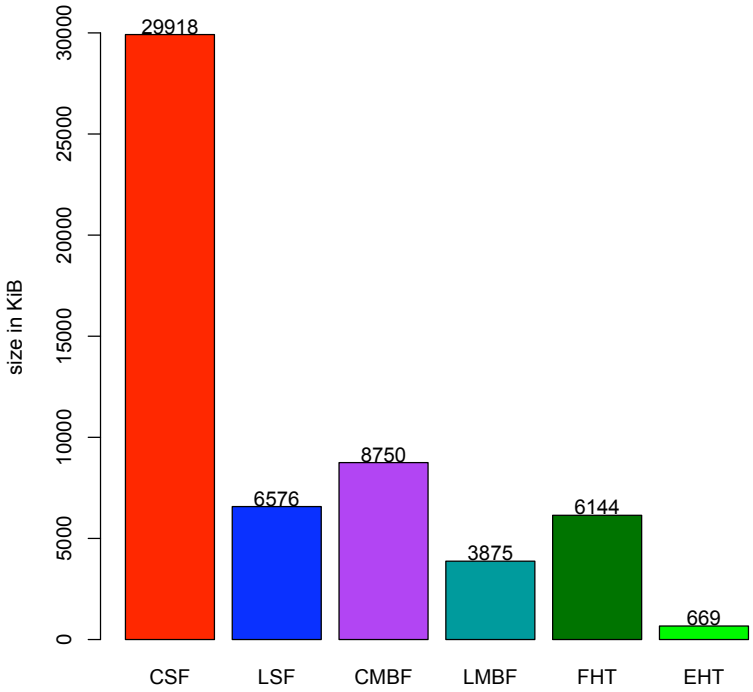

Figure 6: Summary size comparison for n = 1,000,000

fully resemble the complex structure of the EHT but simulates it's behavior appropriately.

For simulations we use the following parameters:

$$n = \{100,000; 1,000,000\}; c = \{6.4; 3.2; 1.6\}; \chi = \{4; 5\}; |\omega| = \{64; 128\}$$

for a total of 32 different simulations. The number of hash functions k is always chosen optimal. 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 and queried for all n keys. As summary a HCCBF is used. The compression rate γ_h is automatically calculated to be optimal. 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.

Here we will only present and discuss a subset of the results. For an extensive discussion see section 4 of [1].

4.1 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 and then take the maximum of the frequencies to evaluate the worst-case behavior. The results are shown in table 1.

		load				
configuration	$E_{maxload}$	0	1	2	3	4
$n = 10^6, c = 1.6$	3	1184464	837562	80950	684	1
$n = 10^6, c = 3.2$	2	3204894	980039	10438	1	0
$n = 10^6, c = 6.4$	2	7388934	999621	217	0	0
$n = 10^5, c = 1.6$	3	167662	89728	5327	24	0
$n = 10^5, c = 3.2$	2	424659	99411	369	0	0
$n = 10^5, c = 6.4$	2	948583	100000	5	0	0

Table 1: Entry distribution.

For all the tables with $n = 1e+6$ there was one bucket overflow in the worst-case. That is, only one bucket must be diverted to CAM. None of the buckets for tables with $n = 1e+5$ experienced an overflow. Column $E_{maxload}$ shows the expected maximum load. As can be seen, the EHT performs as expected.

4.2 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. We also calculate the expected number of CAM entries. Table 2 shows the results.

n	c	χ	min	max	avg	expected
10^6	4	1.6	5017	5446	5194.05	5181
10^6	5	1.6	236	287	258.20	265
10^6	4	3.2	40	61	47.00	47
10^6	5	3.2	0	0	0.00	0
10^6	4	6.4	0	0	0.00	0
10^6	5	6.4	0	0	0.00	0
10^5	4	1.6	144	209	177.95	178
10^5	5	1.6	2	11	6.05	6
10^5	4	3.2	0	1	0.15	0
10^5	5	3.2	0	0	0.00	0
10^5	4	6.4	0	0	0.00	0
10^5	5	6.4	0	0	0.00	0

Table 2: Number of CAM entries.

Once again, the results closely resemble the expectations. One interesting fact is, that 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. 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.

4.3 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 3 shows the results.

n	c	χ	$\|\omega\|$	min γ_h	max γ_h	avg γ_h	γ_p	γ_0	max bits
10^6	1.6	4	64	22	24	22.8	27	21.3	63.3
10^6	1.6	5	64	21	22	21.5	24	21.3	63.3
10^6	1.6	4	128	50	53	51.0	55	42.6	126.4
10^6	1.6	5	128	47	51	49.5	49	42.6	125.1
10^6	3.2	4	64	23	26	24.6	27	21.3	62.7
10^6	3.2	5	64	24	25	24.9	24	21.3	63.2
10^6	3.2	4	128	56	59	57.7	55	42.6	126.3
10^6	3.2	5	128	55	58	56.9	49	42.6	126.3
10^6	6.4	4	64	24	26	24.8	27	21.3	63.3
10^6	6.4	5	64	23	25	24.3	24	21.3	63.3
10^6	6.4	4	128	56	58	57.3	55	42.6	127.2
10^6	6.4	5	128	55	58	56.0	49	42.6	126.1
10^5	1.6	4	64	25	27	26.0	27	21.3	62.6
10^5	1.6	5	64	24	26	25.4	24	21.3	62.5
10^5	1.6	4	128	57	60	58.8	55	42.6	126.6
10^5	1.6	5	128	55	60	57.8	49	42.6	125.7
10^5	3.2	4	64	23	26	25.5	27	21.3	63.0
10^5	3.2	5	64	23	26	24.6	24	21.3	62.1
10^5	3.2	4	128	57	60	58.3	55	42.6	126.9
10^5	3.2	5	128	56	59	57.0	49	42.6	125.8
10^5	6.4	4	64	24	26	25.0	27	21.3	62.6
10^5	6.4	5	64	22	26	24.2	24	21.3	62.5
10^5	6.4	4	128	56	59	57.2	55	42.6	125.8
10^5	6.4	5	128	55	58	56.5	49	42.6	126.3

Table 3: 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.

Better compression can be achieved by reducing the word-size $|\omega|$ while retaining γ_h. Of course this leads to more word overflows which have to be compensated by additional memory. For example, for $n = 1.000.000$, reducing $|\omega|$ to 118 bits saves 10 bits per word at the cost of additional 160 overflown words.

4.4 Comparing sizes

This section presents 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.

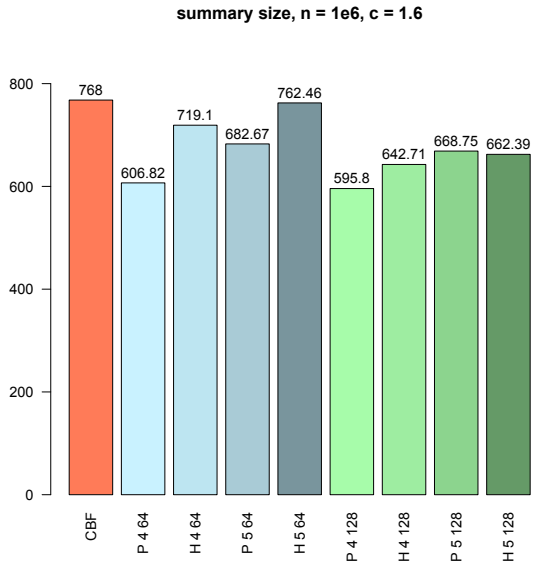

Figure 7: Summary sizes for n = 1e6, c = 1.6

5 Conclusion

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.

- 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.

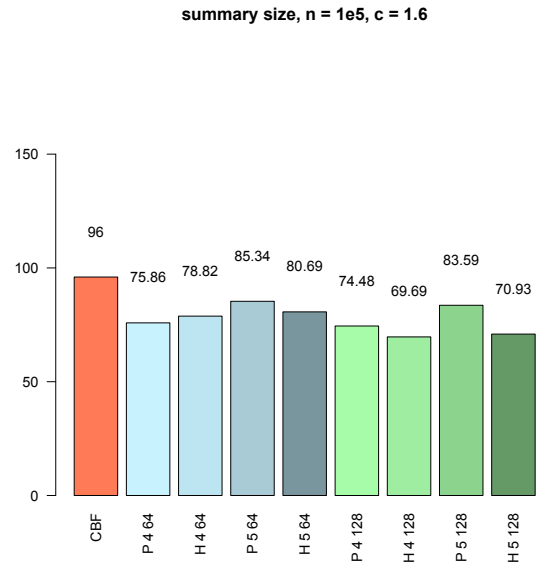

Figure 8: Summary sizes for n = 1e5, c = 1.6

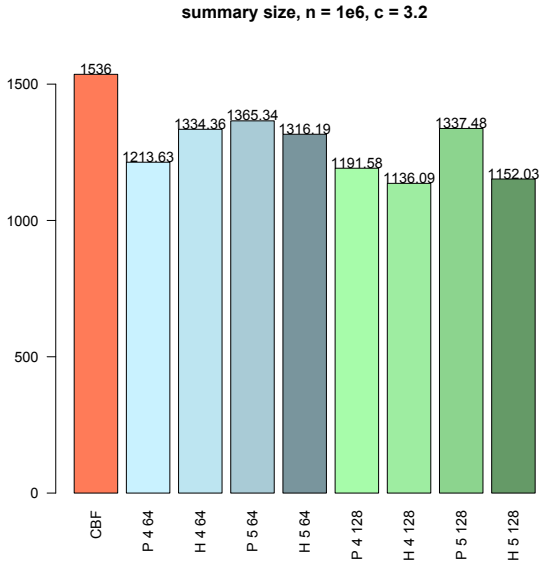

Figure 9: Summary sizes for n = 1e6, c = 3.2

- 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.

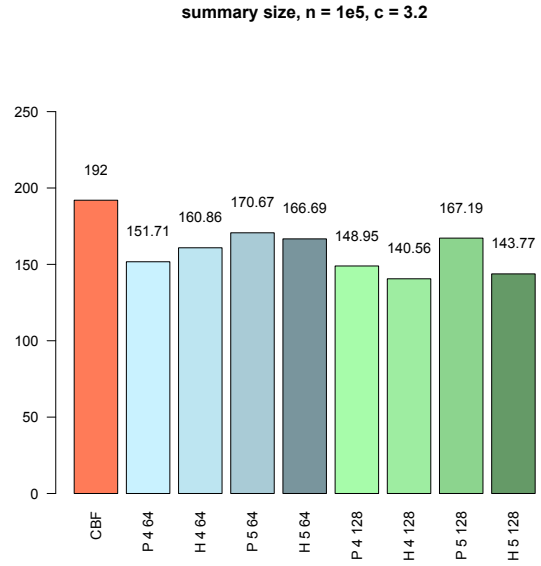

Figure 10: Summary sizes for n = 1e5, c = 3.2

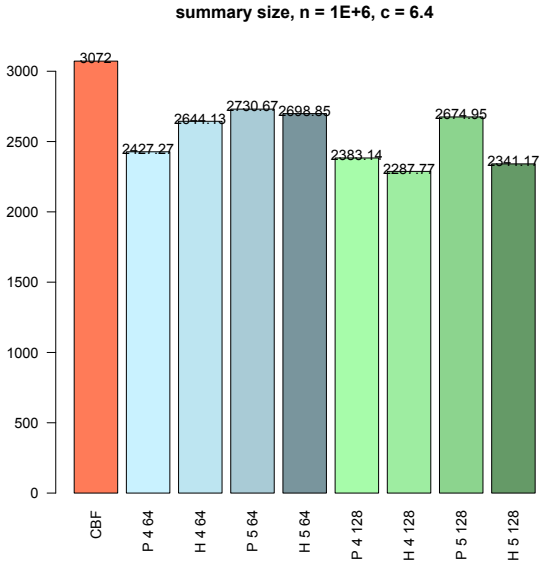

Figure 11: Summary sizes for n = 1e6, c = 6.4

5.1 Recommendations

Following the theory and results an EHT implementation could be as follows.

- Prefixes are sorted and stored in tables according to length. The maximum prefix length is 32 bits resulting in hash table trees for prefixes up to 64

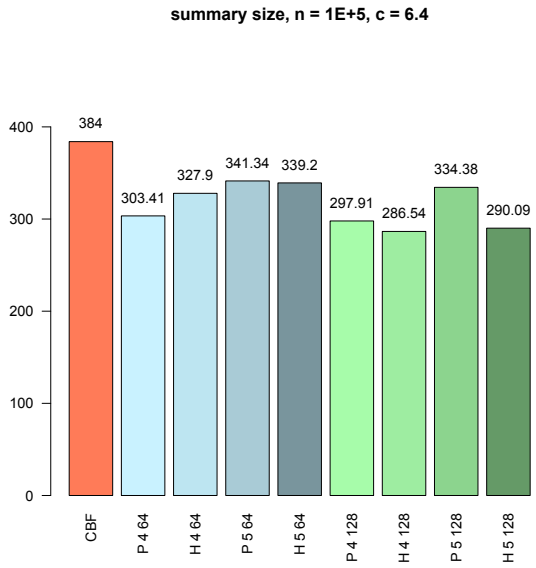

Figure 12: Summary sizes for n = 1e5, c = 6.4

bits. The very few prefixes larger than 64 bits are kept in CAM.

- The prefixes are stored in EHTs with $c = 1.6$ buckets/counters per prefix. This leads to an expected maximum load of 3 which is exceeded only very rarely. With $n = 4 \cdot 10^6$ entries a single table would have 2^23 buckets. Using the hashing scheme of section **??** a prefix needs only 11 bits leaving much room for an associated value. A 64 bit wide off-chip DDR memory is used. This provides provides enough space to make bucket overflows vanish even for millions of entries. Alternatively, longer prefixes can be allowed (approximately up to 48 bits, depending on n and the value size).

- The maximum online counter value is $\chi = 5$. With millions of entries the expected number of entries which have to be stored in CAM is still only in the tens.

- A 128 bit wide on-chip memory is used. This allows compressing about 50 counters per word using a HCCBF. To keep overflown bits a small CAM with a few cells suffices. The memory width can be reduced at the cost of additional CAM cells. Since the number of entries grows exponentially with every bit saved, it's probably best not to go below 110.

Our implementation includes a simulator which can be freely configured and simulates the construction and behavior of an EHT including updates. Thus sample configurations can be tested and the results used for a practical implementation.

References

[1] T. Zink, "Packet forwarding using improved bloom filters," Master's thesis, University of Konstanz, 2009. 1, 2, 2.2, 3.1, 3.3, 3.4, 3.6, 3.7, 4

[2] 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. 1, 2.1, 3.7

[3] A. Kirsch and M. Mitzenmacher, "Simple summaries for hashing with choices," *IEEE/ACM Trans. Netw.*, vol. 16, no. 1, pp. 218–231, 2008. 2.2, 3.7

[4] 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. 2.2

[5] 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. 3.3

[6] "IPv6 report." http://bgp.potaroo.net/index-v6.html. 3.3

[7] 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. 3.3

[8] F. Bonomi, M. Mitzenmacher, R. Panigrahy, S. Singh, and G. Varghese, "An improved construction for counting bloom filters," 2006. 3.3