Technical Report
KN–2011–DiSy–01

Distributed System Laboratory

Exploiting Degrees of Freedom for Efficient Hashing in Network Applications

Thomas Zink **Marcel Waldvogel**
Distributed Systems Laboratory
Department of Computer and Information Science
University of Konstanz – Germany

Abstract. Hashing has yet to be widely accepted as a component of hard real-time systems and hardware implementations, due to still existing prejudices concerning the unpredictability of space and time requirements resulting from collisions. While in theory perfect hashing can provide optimal mapping, in practice, finding a perfect hash function is too expensive, especially in the context of high-speed applications.
The introduction of hashing with multiple choices, *d-left hashing* and Bloom filter-based hash table summaries, has caused a shift towards guaranteed single-DRAM access. However, these guarantees come at a high price. High amounts of rare and expensive high-speed SRAM needs to be traded off for predictability. Moreover, it is infeasible for many applications to provide enough on-chip memory.
In this paper we show that previous suggestions suffer from the false precondition of full generality. To provide a workable solution, our approach exploits four individual degrees of freedom available in many practical applications, especially hardware and high-speed lookups. This reduces the requirement of on-chip memory up to an order of magnitude at the cost of only minute amounts of additional hardware. Our design makes fast hash table implementations cheaper, more predictable and above all, more practical.

Table of Contents

Abstract. . . . . a
1 Introduction 1
2 Related Work 2
3 Review of Bloom filter-based summaries 4
4 Ignore the false positive probability 7
4.1 Counter Values 7
4.2 Bucket load 8
5 Separate Update and Lookup Engines 9
5.1 Maximum Counter Value 9
5.2 Encoding 11
5.3 Updates 12
6 SRAM Memory Optimization 15
7 DRAM Memory Optimization 18
7.1 Multiple entries per word 18
7.2 Multiple words per bucket 19
8 Evaluation 20
9 Conclusion 25
References 27
List of Figures 28
List of Tables 29
List of Algorithms 30

1 Introduction

Efficient hashing in network applications is still a challenging task, because tremendously increasing line speeds, demand for low power consumption and the need for predictability pose high constraints on data structures and algorithms. At the same time, memory access speeds have almost stayed constant, especially because of the latency and waiting time between accessing the same bank repeatedly. Hashing has yet to be widely accepted as an ingredient in hard real-time systems and hardware implementations, as prejudices concerning the unpredictability of size and time requirements due to collisions still persist.

Modern approaches make use of multiple choices in hashing [1, 2] to allow compact hash tables with a small number of memory accesses. In addition, table summaries [3, 4], based on (counting) Bloom filters [5, 6] and derivates, further limit table accesses to one with high probability (w.h.p.) at the cost of fast but expensive on-chip memory (SRAM). The summaries allow set membership queries with a low false positive rate and also reveal the correct location of an item if present.

Although these improvements address space and time requirements, they come at a high price. SRAM is extremely expensive and, while external DRAM can be shared, it must be replicated for every network processor. In addition, numerous networking applications compete for their slice of this precious memory. For many - like socket lookups, Layer-2 switching, packet classification and packet forwarding - tables and their summaries tend to grow extremely large, up to the point where providing enough SRAM is not applicable.

We propose mechanisms to construct an improved data structure which we name *Efficient Hash Table (EHT)*, where efficient relates to on-chip memory usage but also to lookup performance. The design aggressively reduces the amount of bits per item needed for the on-chip summary, while still retaining a cost of one DRAM access per lookup with high probability.

Previous approaches are misguided by the need for full generality. Careful observation of network applications reveals certain degrees of freedom which can be exploited to achieve significant improvements. These observations lead to the following four key ideas:

- The summary's false positive rate can be ignored, it is irrelevant in respect to lookup performance.
- The update and lookup engines can be separated. The on-chip summary need not to be exact.
- The summary can be de/compressed in real time.
- The load of a bucket can potentially be larger than one without increasing memory accesses.

In conjunction, these concepts reduce SRAM memory size up to an order of magnitude, but they can also be applied individually for significant improvements.

2 Related Work

A hash function h maps items of a set S to an array of buckets B. Their natural application are hash tables, or dictionaries, that map keys to values. In theory, a *perfect hash function* that is injective on S [7], could map n items to n buckets. While perfect hashing for static sets is relatively easy [8], finding a suitable hash function that requires constant space and time to perform the mapping of a dynamic set is infeasible in practice. As a result, hashing has to deal with collisions, where multiple items are hashed into the same bucket. Naïve solutions anchor a linked list or an array of items to the overflown bucket or probe multiple buckets according to a predefined scheme. The need for collision resolution led to the persisting myth that hashing has unpredictable space/time requirements.

In 1994, Dietzfelbinger et al. [9] extended the scheme of Fredman et al. [8] to store dynamic sets. Their dynamic perfect hashing resolves collisions by random selection of universal hash functions [10] for a second-level hash table.

Azar et al. [11] observed, that allowing more possible destinations for items and choosing that destination with lowest load, both, the average as well as the upper bound load, can be reduced exponentially. This effect became popular as the "power of two choices", a term coined by Mitzenmacher in [12]. Vöcking [2] achieved further improvements by introducing the "always-go-left" algorithm, where the items are distributed asymmetrically among the buckets. Broder and Mitzenmacher [1] 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 (*d-left hashing*). A lookup requires examining the d locations. Since the d choices are independent, lookups can be performed in parallel or pipelined. A survey of multiple-choice hashing schemes and their applications can be found in [13].

Bloom Filters [5] 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 $\forall h \in \mathbf{N} : 0 \leq h < m$. The bits at the k corresponding positions are then set. A query for an item y just checks the k bits corresponding to y. If all of them are set, y is reported to be a member of S. 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 as shown in equation 1.

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

The major drawback of Bloom filters is that they do not allow deletions. Fan et al. [6] address this issue by introducing a *counting Bloom filter (CBF)*. Instead of a bit array, 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. Later, Bonomi et al. presented an improved version of CBF based on *d-left hashing* [14].

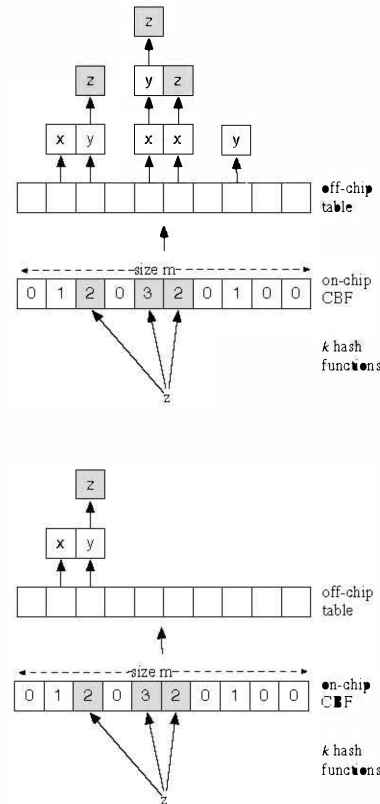

Fig. 1. The two Fast Hash Tables. The Basic FHT (top) replicates every item. The Pruned FHT (bottom) only keeps the 'left'most ('Left' refers to the table entry with the least index)

In [15] Mitzenmacher proposes arithmetic coding for Bloom filters used for exchanging messages (web cache information) in distributed systems. Recently, Ficara et al. [16] introduced a compression scheme for counting Bloom filters based on Huffman coding. They name their structure *MultiLayer Compressed Counting Bloom Filter (ML-CCBF)*. The compressed counters are stored in multiple layers of bitmaps. Indexing requires perfect hash functions since collisions must be avoided. The structure provides near optimal encoding of the counters but retrieval is extremely expensive. The authors propose splitting the bitmaps into equal sized blocks and using an index structure to lower the cost of a counter lookup.

Bloom filters have since gained a lot of attention especially in network applications [17]. Today, Bloom filters can be used as histograms [18] and represent arbitrary functions [19]. In 2005 Song et al. [3] suggested using Bloom filters as a hash table summary. This idea was later refined in [20]. Bloom filter-based summaries are also used for minimal perfect hashing [21].

3 Review of Bloom filter-based summaries

Our work is based on the schemes presented by Song et al. [3] and Kirsch and Mitzenmacher [20], which we will now review for completeness.

Song et al. [3] presented a new hash table design, named *Fast Hash Table*, based on hashing with choices and counting Bloom filter summaries that targets hardware implementations and provides fast lookups by utilizing on-chip memory to optimize performance. Their scheme eliminates the need for parallel lookups usually required by multiple-choice hashing. Each b-bit counter ($b = 3$) in CBF summary corresponds to a bucket in the hash table and represents the number of items hashed into it. Note, that if b is small, the probability of counter overflows can't be neglected. Song et al. propose using a small CAM for overflown counters. There are a total of m counters/buckets where m is calculated using equation 2.

$$m_{\mathrm{FHT}} = 2^{\lceil \log c\ n \rceil} \tag{2}$$

The constant c needs to be sufficiently large to provide low false positive and collision probabilities. It is set to 12.8 which is considered optimal. k independent hash functions, where k is derived by equation 3, are used to index both CBF and the hash table.

$$k = \lceil \frac{m}{n} \ln 2 \rceil \tag{3}$$

The *Basic Fast Hash Table (BFHT)* simply replicates all inserted items to all k locations in the table and increments the counters. As an improvement the table can be *pruned* leading to a *Pruned Fast Hash Table (PFHT)*. All replicas are removed except for the leftmost with the lowest counter value (figure 1). A lookup only requires examining the least loaded bucket, i.e., the one with the lowest counter value. While pruning improves lookup time by reducing bucket loads, updates require an additional offline BFHT since items need to be relocated when their associated counters change.

With $c = 12.8$ buckets per item and $b = 3$ bit wide counters CBF summary requires 38.4 bits per item of on-chip memory. Following equation 2 the total amount of bits β needed for the on-chip summary is

$$\beta_{\mathrm{FHT}} = 2^{\lceil \log c\ n \rceil} \cdot b \tag{4}$$

The rather high requirement of SRAM has later been addressed by Kirsch and Mitzenmacher [20, 4]. Their key idea is to separate the hash table from its summary to allow individual optimizations. They propose using a *Multilevel Hash Table (MHT)* [22] consisting of $d = \log \log n + 1$ individual tables geometrically decreasing in size. An occupancy bitmap is kept in on-chip memory that allows efficient queries for empty buckets (see figure 2).

The bitmap requires a number of bits equal to the number of buckets m which is defined as

$$\beta_{\mathrm{MHT}} = m_{\mathrm{MHT}} = \sum_{i=1}^{d} (c1 \cdot c2^{i-1} \cdot n) \tag{5}$$

with the constants c_1, c_2 chosen such that $c_1 > 1$, $c_2 < 1$, $c_1 c_2 > 1$. Following Song et al. to eliminate parallel lookup overhead, Kirsch and Mitzenmacher present three summary structures, the *interpolation search (IS)*, *single filter (SF)*

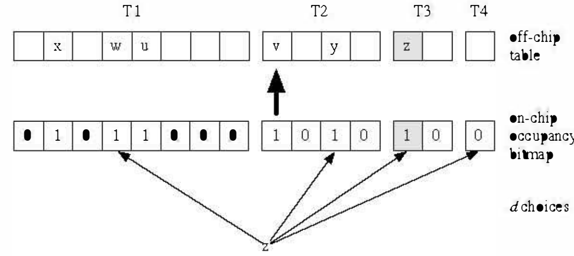

Fig. 2. Multilevel hash table with on-chip occupancy bitmap

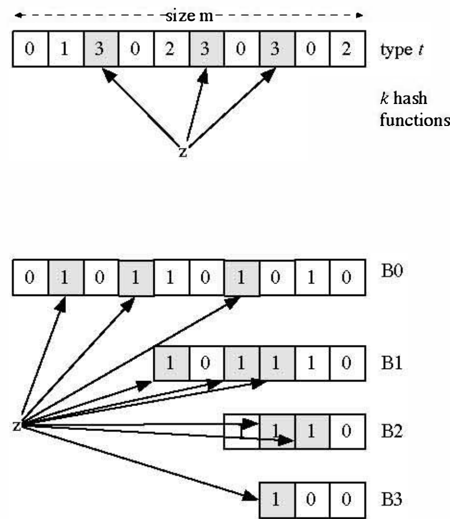

Fig. 3. Single filter (SF) and multiple Bloom filter (MBF) summaries. The SF is a single Bloomier filter representing the type of an item. The MBF is an array of Bloom filters decreasing in size

and *multiple Bloom filter (MBF)* summaries. Since IS is not applicable in our targeted environment we will cover only the latter two summaries which are based on Bloom filters. They are depicted in figure 3.

The SF summary is a single Bloomier filter[19] representing the type t of an item where t corresponds to the sub-table of the MHT where the item is located. In addition to false positives, it can also return type failure. To keep the probability small the filter must be sufficiently large. The number of cells m is defined as

$$m = n \ \log n \tag{6}$$

With $d = \log\log n + 1$ sub-tables (types) the total number of bits needed is

$$\beta_{\mathrm{SF}} = n \ \log n \ (\log\log\log n) \tag{7}$$

The MBF summary is constructed of an array of Bloom filters $B = \{B_0, ..., B_{t-1}\}$. 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. This leads to the need of extremely small false positive probabilities to guarantee successful lookup. For a well designed MHT the number of bits the MBF requires is

$$\beta_{\mathrm{MBF}} = n \, \log n \tag{8}$$

Both, the SF and MBF summaries, support only inserts. To allow deletions significantly more effort is required. Kirsch and Mitzenmacher suggest two approaches. For *lazy deletions* a deletion bitmap is kept alongside the occupancy bitmap in on-chip memory with one bit for every bucket in the MHT. On deletion, the corresponding bit is set to 1. During lookup, items in buckets that have a set deletion bit are ignored. The *counter based deletions* add counters to the SF and MBF summaries to keep track of the actual number of items. The authors do not suggest specific values for the counter width nor provide evaluation. They state however, that a counting MBF requires about 3.3 times more space than a simple MBF, that is

$$\beta_{\mathrm{MCBF}} = 3.3 \cdot n \log n \tag{9}$$

With d choices and v wide counters the modified SF requires

$$\beta_{\mathrm{SFc}} = v \cdot d \cdot n \, \log n \tag{10}$$

bits.

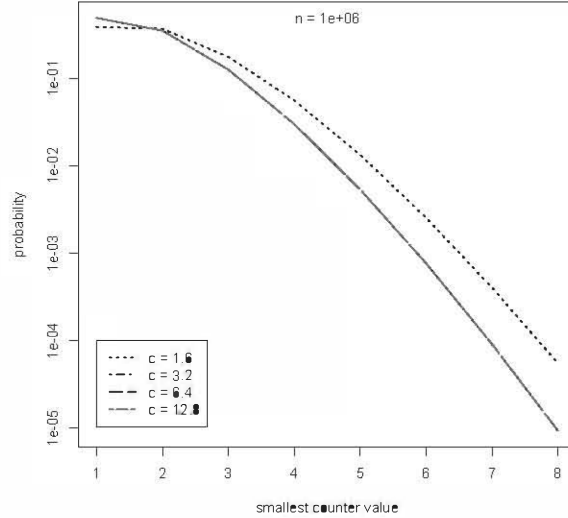

Fig. 4. Counter value probabilities for different c. For $c > 1.6$ there is no effect on the counter distribution. For $c \leq 1.6$ the probability for higher counters increases

4 Ignore the false positive probability

Bloom filters are usually constructed to optimize the false positive probability. In case of the MHT summaries having a negligible small false positive rate is essential to prevent type failure. In general, applications that require exact knowledge about set membership are dependent on minimizing false positives. This inevitably leads to relatively large filters.

We observe that applications using Bloom filter-based summaries as an index into another data structure, like the FHT, do not suffer from false positives, as long as a successful lookup independent of the false positive probability is guaranteed. The structure must provide a predictable worst-case lookup performance. A false positive returned by the summary leads to a table lookup that returns NULL. The worst-case performance is not affected. In conclusion, Bloom filter-based summaries can be potentially much smaller.

By reducing the address space of the summary while keeping the number of entries n constant, counter values and the load of buckets are expected to increase. There exists a trade-off between reducing on-chip memory requirements and the resulting counter values and bucket loads.

4.1 Counter Values

Counters 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 [3].

$$p_i = \binom{nk}{i}(\frac{1}{m})^i(1 - \frac{1}{m})^{nk-i} \quad (11)$$

This is not entirely accurate. The probability that, due to collisions, less than k counters for an item can be increased, is neglected. But the estimate is close enough to allow counter value predictions. Figure 4 shows the counter distribution for different c. The constant c is chosen to divide m into multiples of 2. As long as $c > 1.6$ the counter distribution is not affected. For $c \leq 1.6$ the probability for higher counters increases. This is the result of an overestimate of the number of choices k. Following equation 3, k depends on the number of buckets per item $\frac{m}{n}$. As $\frac{m}{n} \to 2$, k will lead an overestimate resulting in higher counter values. In conclusion, as long as $\frac{m}{n} > 2$ and k chosen optimal, the counter values are not affected by smaller sized filters. Hence the counter width in terms of bits is unaffected.

4.2 Bucket load

c	k	E
12.8	12	1
6.4	6	2
3.2	3	2
1.6	2	3
1	1	5

Table 1. Expected maximum load for different c

We follow [11] to predict the expected maximum load that occurs with high probability. With n items, m buckets and k choices the expected maximum load is defined as

$$E_{\text{maxload}} = \frac{\ln \ln m}{\ln k} \tag{12}$$

The equation holds for any $m \to \infty$ with $n = m$ and $k \geq 2$. In our design, however, $m \gg n$. The result leads an overestimate of the maximum load, which in practice should be smaller. To compensate we apply the *floor* function to round to the next lower integer. A special case arises for $k = 1$. This happens when $\frac{n}{m} \to 1$. Then the expected maximum load is defined as

$$E1_{\text{maxload}} = \frac{\ln n}{\ln \ln n} \tag{13}$$

Table 1 shows the expected maximum load in respect to different c. The results are surprisingly positive. Setting $c = 3.2$ results in a summary size $\frac{1}{4}$ of the optimum proposed in [3]. The maximum load increases from 1 to 2 w.h.p.. In other words, allowing two entries per bucket leads to a reduction in on-chip memory size by a factor of four. The trade-off even improves for $c = 1.6$. With three entries per bucket, the on-chip memory size can be reduced to $\frac{1}{8}$ of the optimum.

The problem arising is how to deal with more than one entry per bucket. A naïve solution is to use E memory backs, one for each possible entry, and query them in parallel. The additional cost is acceptable compared to the saved SRAM. In section 7 we will discuss this issue in more detail and present techniques that allow multiple entries per bucket and do not require parallel or sequential memory accesses.

5 Separate Update and Lookup Engines

Previous suggestions have shown that support for updates is accompanied by enormous overhead to the tables and their summaries. The PFHT needs an additional offline BFHT to identify entries that have to be relocated. The MHT requires an occupancy bitmap and the summaries require either a deletion bitmap for lazy deletions or counting filters.

In most real-world applications, especially those that require fast lookups, updates are much rarer than lookups. By completely separating update and lookup engines, on-chip requirements can be reduced. The idea is to keep two separate summaries. One is kept online in on-chip memory and is optimized for lookups. It does not need to be exact and can be different from the update summary which is kept offline. Keeping only an approximate online summary allows individual optimization and more efficient encoding. The update engine precomputes all changes and sends modifications to the online structures.

Although, some of the techniques we describe are applicable to different table and summary structures such as the FHT and MHT, we concentrate on optimizing the scheme of Song et al. [3], which we argue has most room for improvement. Figure 5 shows a simplified overview of our design. It is composed of the offline CBF and BFHT, an online on-chip compressed CBF (CCBF), the online PFHT in off-chip memory and a small extra memory (CAM, registers) for overflown entries (we will refer to the overflow memory as CAM in the following). Strictly, the offline CBF is not needed, the counter values could also be computed by examining the length of the linked list. However, this would lead to significant overhead when querying counters, so we keep the offline CBF for performance reasons.

5.1 Maximum Counter Value

A lookup requires retrieving the leftmost smallest counter in the CBF summary. Successful lookup is guaranteed as long as not all counters corresponding to a key are overflown. If all the counters are overflown, it is not be possible to identify the correct bucket. The goal is to identify a maximum allowed counter value χ where the probability that all $k' < k$ chosen counters for an item equal χ is appropriately small. In essence, choosing an appropriate value for χ is a trade-off between storage saved, the number of counter overflows, and the number of expected lookup failures.

[3] gives an analysis of the probability that in any $k' < k$ chosen buckets the counter value has a specific height s. The derivation of the equation is quite complex and for simplicity left out at this point. Interested readers are referred to the actual paper. Figure 6 shows the expected smallest counter value in k' chosen counters depending on the size m, or to be specific, the buckets per item constant c. The constant c is chosen to divide m by multiples of 2. As expected the table size has significant impact on the smallest counter value. That is, for smaller c the probability of choosing a higher counter is higher. When reducing c the maximum counter value χ must be higher.

To be able to retrieve all entries the event that all chosen $k' < k$ counters equal χ must be dealt with. The easiest solution is to move entries which cannot be retrieved by calculating the counters to CAM. A small CAM must already be

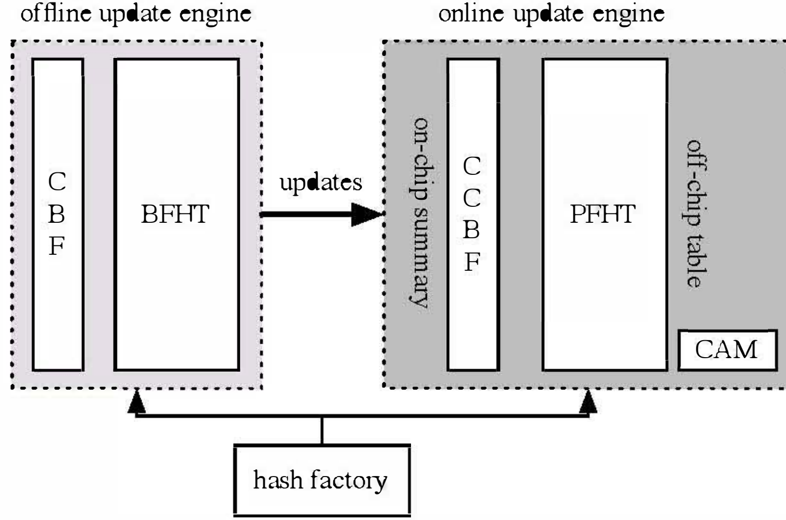

Fig. 5. Efficient Hash Table design overview. The offline update engine precomputes all updates. The online lookup engine is optimized for time/space efficient lookups. The online summary is not exact and compressed. The table is pruned. A small extra memory is used for counter and entry overflows

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.

$$E_{\mathrm{CAM}} = \sum_{i=\chi}^{l} Pr\{C = i\} \times n \qquad (14)$$

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 table 2. 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 trade-off improves for increasing χ. Consulting the numbers, each time χ is incremented once, c can be halved, at the cost of few additional CAM entries.

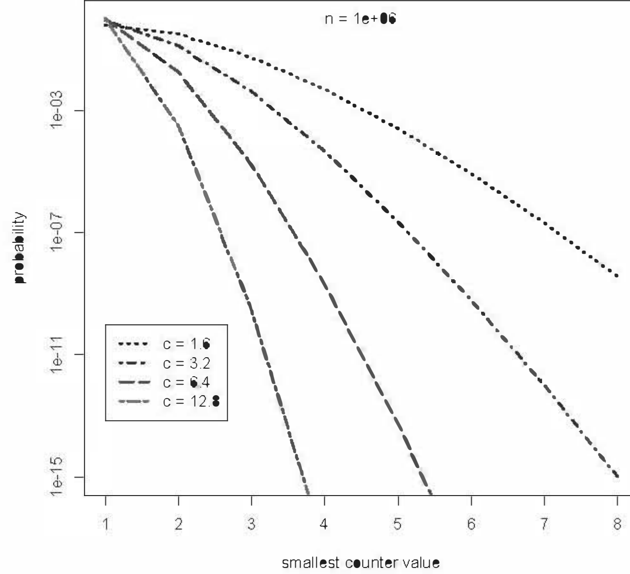

Fig. 6. Probability of smallest counter value in k' counters for different c

	χ		
c	5	4	3
12.8	0	0	0
6.4	0	0	17
3.2	0	47	4183
1.6	285	5181	61110

Table 2. Expected number of CAM entries for different c and χ with $n = 10^6$ inserted items.

5.2 Encoding

Limiting the counter range allows for better optimized encoding of the summary. We follow a simple and well known approach that is also used in [4] to pack few counters into one Byte. The difference is that we extend the scheme to an arbitrary word size to achieve higher compression rates. We argue, that SRAM, being implemented on-chip, can potentially have an arbitrary word size. Basically, the wider the memory, the more counters can be packed into one word and the more bits can be saved. In reality, one will not find memory widths > 128.

Counters that are limited in range can easily be encoded in a specified number of bits. Let ω be a memory word, $|\omega|$ its width in bits, and counters limited to the range $[0, \chi]$, then the number of counters that can be packed into ω is defined as

$$\gamma_p = \lfloor \frac{log\ 2^b}{log\ |[\chi + 1]|} \rfloor \qquad (15)$$

We will also refer to γ as the compression rate. Compression (equation 16) and decompression is trivial. Implemented in hardware, all counters can be unpacked in parallel.

$$\omega = \sum_{i=0}^{\gamma_o - 1} \varsigma_i \cdot ||[\chi + 1]|^i \tag{16}$$

We will introduce a more sophisticated Huffman compressed summary in section 6.

5.3 Updates

In our design we want to completely separate updates from lookups to keep interference with the lookup process as small as possible. When performing updates, the offline table pre-computes all changes and applies them to the online CCBF, PFHT and CAM.

There are three types of entries that must be distinguished. *Offline entries* are kept in the offline BFHT. Due to overflows, each offline entry has a corresponding online entry either in the online PFHT (*table entry*) or in extra memory (*cam entry*). The update engine must be able to identify which of the *offline entries* in affected buckets are *table entries*, and which are *cam entries*. Else, it would not be possible to compute relocations without examining all possible locations in the online structure. Since we want to minimize online table access all *offline entries* are paired with a *locator*. In case the corresponding entry is a *table entry*, the *locator* is simply the index of the hash function used to store the *table entry*. If it is a *cam entry*, the locator is set to ∞. An offline entry of item x thus is defined as $E_{\text{offline}}(x) \leftarrow (k, v, i)$, where k denotes the key, v the associated value, and i the locator.

Algorithm 1 shows the pseudocode for insertions. First we initialize a relocation list R, a counter increment list L and an update map M. The list is used to collect all entries that are considered for relocation while the update map maps online buckets to their new value. 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 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. Note, that in any case we create a new offline entry with a locator set to ∞ since we cannot yet know where the item is placed. Only after relocation we can be sure, whether the item is put to the table or to CAM. We then collect all entries in affected buckets that are also either table entries or cam entries, add the new offline entry and increment the counters. Note, that the table entries are inserted at the head of the list, while the cam entries are appended to the end. This is for balancing reasons. Online entries must be relocated prior to CAM entries since it is possible that space becomes available to hold the entries from CAM. Next the collected entries must be considered for relocation. For each collected entry we compute the hash values, the new locator and the new bucket address. We also collect all online entries for the target bucket. If the new address is different from the old address the entry r might be relocated. There are 3 possible events:

1. The entry is moved inside the table. M is updated with an empty entry at the old bucket. If the new bucket has enough space left, M is updated with

Algorithm 1: Insert

Data: k: number of choices, B: offline BFHT, C: offline CBF, χ: maximum counter value

Input: x: the item to insert

Output: updated tables and summaries such that they include x

1 **procedure:** insert (x) **begin**
2 $R, L \leftarrow 0$;
3 $M \leftarrow$ map: $\{$bucket, content$\}$;
4 $H \leftarrow \{h_i(x)$ **for** $i \leftarrow 0$ **to** $k\}$;
5 $\zeta \leftarrow \{C_h \forall h \in H\}$;
6 $l, a \leftarrow \infty$;
7 **if** $!(c \geq \chi)\ \forall c \in \zeta$ **then**
8   $l \leftarrow$ `SmallestIndexOf(`$\min(\zeta), \zeta$`)`;
9   $a \leftarrow H_l$;
10 `// collect and insert`
11 $e \leftarrow$ new offline entry(x, ∞);
12 **for** $\forall h \in H$ **do**
13   $R \leftarrow$ insert `TableEntries(`B_h`)`;
14   $R \leftarrow$ append `CamEntries(`B_h`)`;
15   $B_h \leftarrow B_h \cup e$;
16   **if** $C_h < \chi$ **then** $L \leftarrow L \cup C_h$;
17   $C_h + +$;
18 `// compute relocations`
19 **for** $\forall r \in R$ **do**
20   compute new Hash values H_n, counters $zeta_n$, locator l_n, old and new bucket address a_o, a_n;
21   **if** $a_n \neq a_o \&\& a_o != \infty$ **then**
22   `// entry moved within table`
23   **if** `!SpaceLeft(`B_{a_n}`)` **then**
24     $a_n = \infty$;
25   **else**
26   M`.Update(`$\{a_o, 0\}, \{a_n, r\}$`)`;
27   **if** $a_n \neq a_o \&\& a_o == \infty$ **then**
28   `// entry moved from cam to table`
29   **if** `SpaceLeft(`B_{a_n}`)` **then**
30   M`.Update(`$\{\infty, r\}, \{a_n, r\}$`)`;
31   **if** $a_n \neq a_o \&\& a_n == \infty$ **then**
32   `// entry moved from table to cam`
33   M`.Update(`$\{a_o, 0\}, \{\infty, r\}$`)`;
34 `// calculate position of new item`
35 **if** $l \neq \infty$ **then**
36   **if** `SpaceLeft(`B_{H_l}`)` **then**
37    $e.l = l$;
38   **if** $!H_l \in M$ **then**
39   M`.Update(TableEntries(`B_{H_l}`))`;
40   M`. Update(`B_{H_l}, e`)`;
41 `UpdateOnline(`M, L`)`;

the new bucket and r, else r must be moved to cam and M is updated with an ∞ bucket (indicating overflow memory) and r.
2. The entry is moved from cam to table. If the new bucket has enough space left, M is updated with $\{\text{new bucket}, r\}$ and $\{\infty, r\}$. Else r can't be moved to table and M is not updated.
3. The entry is moved from table to cam. M is updated with $\{\text{new bucket}, \emptyset\}$ and $\{\infty, r\}$.

In any case, the locator of a relocated offline entry must be updated.

The actual update of the online structure is performed by the procedure "UpdateOnline". The update map M contains bucket addresses and their associated content. The buckets in M are simply replaced with their new value. A special case is if bucket address is ∞, which indicates overflow memory. In this case the overflow memory is probed for the associated entries. If the entry is present, it is removed, else it is inserted. The list L contains a list of counter addresses that must be incremented.

The PFHT needs only be accessed to write changed buckets. Hence, the complexity is optimal and upper bound by the number of changed buckets. With n items stored in m buckets and $k = \frac{m}{n}\log 2$ choices, the upper bound is $O(1 + \frac{m}{n}k) = O(1 + \log 2)$. Similarly, the online CCBF needs only be accessed for counters that actually change, i.e. those that have not yet reached χ.

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 relocation computed. Afterwards, the bucket from which the item is removed is added to M if not already present. Then the online updates are performed. Deletions have the same complexity as insertions.

6 SRAM Memory Optimization

Section 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 [16]. Computing counter values in the ML-CCBF 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.

We propose another design for compressed counting Bloom filters also based on Huffman compression, which we name *Huffman compressed counting Bloom filter (HC-CBF)*. Huffman compression is used for multiple reasons. It yields optimal and prefix free codes with the distribution of counter values. Compressed counters can be easily and individually decompressed. As mentioned in section 5, counters are limited in range, 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 $\chi + 1$. Figure 7 shows an example Huffman tree for $\chi = 4$. The tree, or codebook, can be stored in very small dedicated hardware.

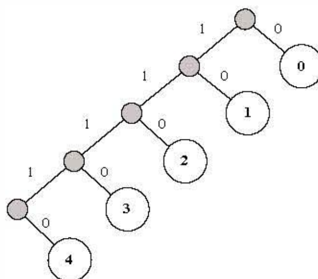

Fig. 7. Example Huffman tree for $\chi = 4$.

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. 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. A first approach to compress the counters is shown in algorithm 2.

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 depends heavily 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

Algorithm 2: Compress

Input: C: offline CBF, H: Huffman tree, χ: maximum counter value, b: word size in bits

Output: Z: online HCCBF, γ_{h}: compression rate

1 **function:** compress (C, χ, b) **begin**
2 `// Initialize compression rate, CCBF, helpers`
3 $\gamma_{\mathrm{h}} \leftarrow \infty$;
4 $Z \leftarrow \emptyset$;
5 $\omega, z, n \leftarrow 0$;
6 `// While there still are counters`
7 **while** $i \leftarrow 0 < |C|$ **do**
8 `// Get the Huffman code of the counter`
9 $z \leftarrow H[\min(C[i], \chi)]$;
10 $i \leftarrow i + 1$;
11 `// Check if we have less counters than the compression rate`
12 **if** $n < \gamma_h$ **then**
13 `// If there still is space in the word add the compressed counter, else reset everything and start with new, lower compression rate`
14 **if** $(|\omega| + |z|) \leq b$ **then**
15 $\omega \leftarrow \omega \cup z)$;
16 $n \leftarrow n + 1$;
17 **else**
18 $\gamma \leftarrow n$;
19 $Z \leftarrow \emptyset$;
20 $\omega, i \leftarrow 0$;
21 **else**
22 `// there are more counters, write the word to CCBF and create a new one`
23 $Z \leftarrow Z \cup \omega$;
24 $\omega \leftarrow z$;
25 `// append last word and return`
26 $Z \leftarrow Z \cup \omega$;
27 **return** Z, γ_h ;

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 rate is achieved. In general, Huffman compression only achieves improvement over word packed compression if $\gamma_{\mathrm{h}} > \gamma_{\mathrm{p}}$. Thus, γ_{p} can be used as a guideline for choosing γ_{h}. Since we force γ_{h} in advance, it can lead to word overflows, if the compressed γ_{h} counters do not fit into a word (in the following we will refer to this scheme as *hard compression*).

Overflows can also occur during insertions. If a counter $c < \chi - 1$ is incremented and the compressed word already occupies all the available bits,

then incrementing the counter will shift one bit out of the word. As a result the last counter value will not be retrievable.

There are different approaches of how to address word overflows. One is to simply ignore the affected 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 bucket of an entry can not be computed.

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 memory, CAM or registers, 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 8, 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

$$\beta_{\mathrm{EHT}} = \lceil \frac{m}{\gamma} \rceil \cdot |\omega| \tag{17}$$

bits in total.

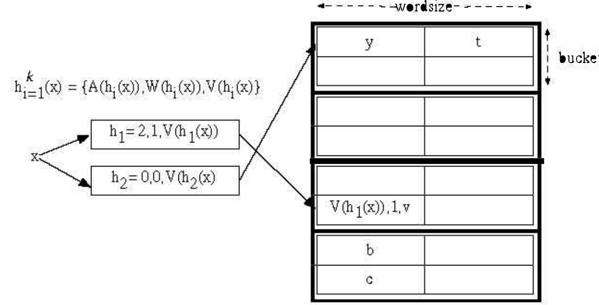

Fig. 8. Verifier hashing and buckets with multiple entries

7 DRAM Memory Optimization

A hash table bucket usually holds a single entry or a reference to a collection of entries. If more than one entry is placed in a bucket, lookup might require multiple memory reads by following pointers. This leads to more sophisticated hash table constructions that try to limit the bucket load to one with high probability.

We argue that by using intelligent hashing and wider memory a bucket can hold more than a single entry without the need of sequential or parallel memory accesses. As a preliminary, we define that a bucket will never hold reference to a collection of entries with variable size. A bucket is defined as an array of entries of fixed size, where every entry can be directly accessed.

7.1 Multiple entries per word

One solution is to allow more entries per memory word. Let $|\omega_D|$ be the word size in bits and $|e|$ be the size of an entry in bits. If $|e| \ll |\omega_D|$, a bucket can hold up to $\lfloor \frac{|\omega_D|}{|e|} \rfloor$ entries which can be read in one cycle. This holds for applications, like QoS/CoS classification, flow-based Server Load balancing or socket lookups, that store only small entries. But many application require larger entries (e.g. IPv6 lookup). While SRAM width is highly flexible, the word size of DRAM is usually fixed, wider memory might not be possible.

By using a hashing scheme similar to that proposed in [14] the size of an entry can be decreased. A class of hash functions can be used that perform transformations of the key, producing k digests of a fixed size, greater or equal to the size of the key. This is crucial to prevent collisions and the hash function must be collision resistant. 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.

$$H : U \to [A] \times [V] \tag{18}$$

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, or addresses. Instead of the key x only its verifier $V(h_i(x))$

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

7.2 Multiple words per bucket

An extension to the former scheme is to allow a bucket to span multiple words. For simplicity, we assume the words are consecutive, although this is not a precondition, as long as there is a fixed offset between the words. A bucket can now be seen as a matrix of r entries per word and w words.

In addition to the address and verifier, the hash function must also lead the correct word, or row, of the bucket. Let $[W]$ the range of words for each bucket.

$$H : U \rightarrow [A] \times [W] \times [V] \tag{19}$$

Note, that in practice $[W]$ will be very small, needing only $1 - 2$ bits. Figure 8 shows the design and an example.

bit	3	2	1	0
parameter	n	c	χ	$\lvert\omega\rvert$
value 0	10^5	1.6	4	64
value 1	10^6	3.2	5	128

Table 3. Parameter configurations p of the software simulations

8 Evaluation

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:
$n = \{10^5; 10^6\}; c = \{3.2; 1.6\}; \chi = \{4; 5\}; |\omega| = \{64; 128\}$
This leads to a total of 16 different parameter configurations. The number of hash functions k is always chosen optimal. In the following, when referencing the parameter configurations, we will use a single hexadecimal digit $p = [0, F]$ representing the encoding depicted in table 3.

On each simulation we perform ten trials, that is we instantiate the EHT and fill it with n random keys and values. 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. As summary we use HC-CBF. The compression rate γ_h is calculated using algorithm 2. 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 HC-CBF and is included in the analysis.

For each try, we calculate the size of the offline CBF, the size of a CCBF and the size of the online HC-CBF. We count the frequency of all counter values in the offline summary and derive the number of overflown counters in the online summary. Every compressed word in the HC-CBF 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.

Counter distribution Since the parameters χ and $|\omega|$ have no effect on the counter distribution, we count the counter frequencies for $n = \{1E+6, 1E+5\}$ with sizes of $c = \{1.6, 3.2\}$ and also calculate the expected frequency for each counter value. The results are shown in figure 9. The real frequencies resemble the expected frequencies almost exactly.

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.

In the worst-case there was only a single unexpected bucket overflow, for tables with $n = 10^6$.

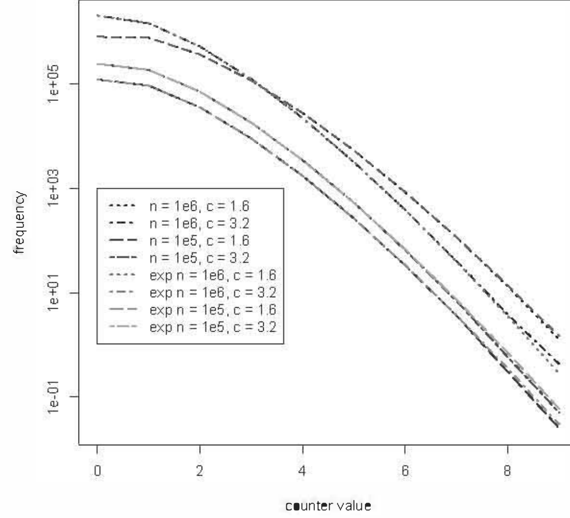

Fig. 9. Real and expected counter frequencies

p	E	load 0	1	2	3	4
$0-3$	3	167662	89728	5327	24	0
$4-7$	2	424659	99411	369	0	0
$8-B$	3	1184464	837562	80950	684	1
$C-F$	2	3204894	980039	10438	1	0

Table 4. Entry distribution and expected maximum load

p	min	max	avg	E
$0-1$	144	209	177.95	178
$2-3$	2	11	6.05	6
$4-5$	0	1	0.15	0
$6-7$	0	0	0.00	0
$8-9$	5017	5446	5194.05	5181
$A-B$	236	287	258.20	265
$C-D$	40	61	47.00	47
$E-F$	0	0	0.00	0

Table 5. Real and expected number of CAM entries

Overflow entries We aggregate the results for χ according to n and c, calculate the average, and take the minimum/maximum values encountered. Following equation 14 we also calculate the expected number of CAM entries. Table 5 shows the results. Once again, the results closely resemble the expectations.

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

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

Table 6. Compression rate

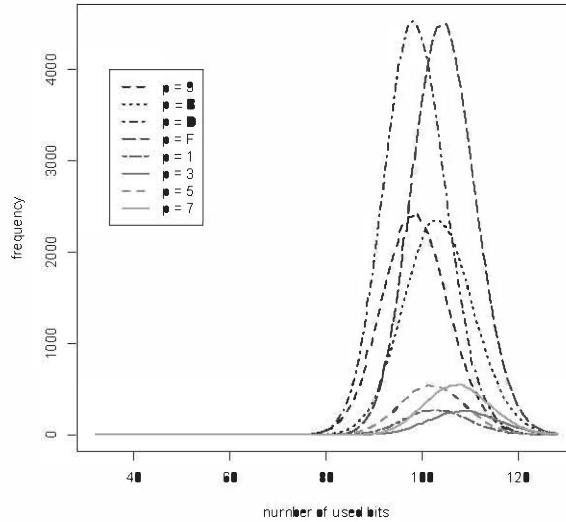

Fig. 10. Frequencies of used bits per compressed word for $|\omega| = 128$

if no compression is used (denoted γ_0). We also include the maximum number of bits actually used to compress the counters.

The numbers in table 6 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 usually the better choice. 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. This reason probably is that the values for χ differ only by 1. It is expected that for higher differences γ_h is more affected.

summary	Size in KiB
FHT	6144
SF	6576
MBF	3875
counting SF	29918
counting MBF	8750
EHT 3.2	1136
EHT 1.6	643

Table 7. Comparison of on-chip requirements of different Bloom filter-based summaries

Another interesting aspect is the frequency of used bits per word (figure 10). The distribution follows a Poisson binomial distribution, which is to be expected. The graphs show a shift of the center depending on χ, which is a result of nearly equal γ_p with different code lengths. The graphs reveal potential to further reduce SRAM requirements. The compression can be improved by reducing $|\omega|$ while keeping the same γ_h, thus, effectively resembling *hard compression*. For example, by reducing $|\omega|$ from 128 to 118 bits, 10 bits per word can be saved. Of course, this leads to a higher number of word overflows. However, making use of the frequency distribution the number of expected overflows can be kept small. By providing CAM for an additional few overflown words, smee bits per on-chip memory word can be saved.

On-chip requirements We will now compare the on-chip requirements of different EHT configurations to the FHT and MHT summaries. In the comparison we include all update overhead. The authors of [3, 20] present evaluation and results for a table with $n = 10^4$ entries. We are interested in much larger tables with $n = 10^6$. Thus, for the FHT and MHT summaries, we calculate the expected summary sizes. Equation 4 is used for the FHT summary. Equation 5 provides the number of bits needed for each of the MHTs occupancy and deletion bitmaps. In case of lazy deletions, the bitmap size is added twice to the summary size. For the MHT summaries we use eq. 7 for SF, eq. 8 for MBF, eq. 9 for counting MBF and eq. 10 for counting SF. The number of items n is set to 10^6, all other parameters are chosen optimal as suggested by the authors. For the EHT we choose $\chi = 5, c = 1.6, |\omega| = 64$ (EHT 1.6) and $\chi = 4, c = 3.2, |\omega| = 128$ (EHT 3.2) and use the HC-CBF as on-chip summary.

Table 7 shows the resulting sizes. Our EHT summaries always perform better than previous solutions. The EHT 1.6 needs less than 6 bits per inserted item and is almost an order of magnitude smaller than the comparable FHT summary. Compared to the smallest MBF summary that only supports lazy deletions out approach still requires about 6 times less the space.

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

– Reducing the size m is achieved by ignoring the false positive probability. As a result, bucket loads will increase which can be compensated by parallel

banks, increasing the off-chip memory width or by better hashing. 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. So only a very small extra overflow memory is needed.
- By separating updates from lookups the lookup summary can be optimized for smaller size and performance. The lookup summary is not exact and limited in counter range $[\chi]$.
- 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. Performance will degrade when $\frac{m}{n} \to 2$.
- Huffman compression is favorable over word packing, 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, saving SRAM.

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.

$$f_{EHT} = \alpha_S \times \beta_{eht} + \alpha_D \times (m \cdot w) + \alpha_C \times (E_{CAM} + E_o). \quad (20)$$

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

9 Conclusion

We have proven that by eliminating unnecessary preconditions and exploiting degrees of freedom present in many applications, on-chip memory requirements can be significantly reduced and lookup performance improved at the cost of minimal additional hardware. Based on four key ideas we have introduced new techniques to design an efficient hash table. The simulation results fully meet the expectations, backup our theoretical analysis and allow accurate predictions.

High amounts of on-chip memory can be traded in for comparatively small amounts of off-chip memory and additional CAM. Cleverly 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 improved 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 solutions up to an order of magnitude.

References

[1] A. Broder and M. Mitzenmacher, "Using multiple hash functions to improve IP lookups," in *IEEE INFOCOM*, 2001. 1, 2

[2] B. Vöcking, "How asymmetry helps load balancing," *J. ACM*, vol. 50, no. 4, pp. 568–589, 2003. 1, 2

[3] 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: ACM Press, 2005, pp. 181–192. 1, 2, 3, 4.1, 4.2, 5, 5.1, 8

[4] A. Kirsch and M. Mitzenmacher, "Simple summaries for hashing with choices," *IEEE/ACM Trans. Netw.*, vol. 16, no. 1, pp. 218–231, 2008. 1, 3, 5.2

[5] B. H. Bloom, "Space/time trade-offs in hash coding with allowable errors," *Commun. ACM*, vol. 13, no. 7, pp. 422–426, 1970. 1, 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*, Sep. 1998, pp. 254–265. 1, 2

[7] T. Hagerup and T. Tholey, "Efficient minimal perfect hashing in nearly minimal space," in *STACS 2001*, ser. Lecture Notes in Computer Science, A. Ferreira and H. Reichel, Eds. Springer Berlin / Heidelberg, 2001, vol. 2010, pp. 317–326, 10.1007/3-540-44693-1_28. [Online]. Available: http://dx.doi.org/10.1007/3-540-44693-1_28 2

[8] M. L. Fredman, J. Komlós, and E. Szemerédi, "Storing a sparse table with O(1) worst case access time," *Journal of the ACM*, vol. 31, no. 3, pp. 538–544, June 1984. [Online]. Available: http://doi.acm.org/10.1145/828.1884 2

[9] M. Dietzfelbinger, K. Mehlhorn, H. Rohnert, A. Karlin, F. Meyer auf der Heide, and R. E. Tarjan, "Dynamic perfect hashing: Upper and lower bounds," *SIAM Journal of Computing*, vol. 23, no. 4, pp. 748–761, 1994. 2

[10] 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

[11] Y. Azar, A. Z. Broder, A. R. Karlin, and E. Upfal, "Balanced allocations," in *SIAM Journal on Computing*, 1994, pp. 593–602. 2, 4.2

[12] M. D. Mitzenmacher, "The power of two choices in randomized load balancing," Ph.D. dissertation, Harvard University, 1996. 2

[13] M. Mitzenmacher, "The power of two choices in randomized load balancing," *IEEE Transactions on Parallel and Distributed Systems*, vol. 12, 2001. 2

[14] F. Bonomi, M. Mitzenmacher, R. Panigrahy, S. Singh, and G. Varghese, "An improved construction for counting bloom filters," in *Proceedings of the 14th conference on Annual European Symposium - Volume 14*. Springer-Verlag, 2006. 2, 7.1

[15] M. Mitzenmacher, "Compressed bloom filters," in *Proc. of the 20th Annual ACM Symposium on Principles of Distributed Computing*, ser. IEEE/ACM Trans. on Networking, 2001, pp. 144–150. [Online]. Available: http://citeseer.ist.psu.edu/461008.html 2

[16] 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, Apr. 2008. 2, 6

[17] A. Broder and M. Mitzenmacher, "Network applications of bloom filters: A survey," in *Internet Mathematics*, vol. 1, 2002, pp. 636–646. [Online]. Available: http://citeseer.ist.psu.edu/broder02network.html 2

[18] 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: ACM, 2003, pp. 241–252. 2

[19] 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: Society for Industrial and Applied Mathematics, 2004, pp. 30–39. [Online]. Available: http://portal.acm.org/citation.cfm?id=982792.982797 2, 3

[20] A. Kirsch and M. Mitzenmacher, "Simple summaries for hashing with multiple choices," in *43rd Annual Allerton Conference on Communication, Control and Computing*, 2005. 2, 3, 3, 8

[21] Y. Lu, B. Prabhakar, and F. Bonomi, "Perfect hashing for network applications," in *2006 IEEE International Symposium on Information Theory*. IEEE press, 2006, pp. 2774–2778. 2

[22] 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: Society for Industrial and Applied Mathematics, 1990, pp. 43–53. 3

List of Figures

1 The two Fast Hash Tables. The Basic FHT (top) replicates every item. The Pruned FHT (bottom) only keeps the 'left'most ('Left' refers to the table entry with the least index) . . . 3
2 Multilevel hash table with on-chip occupancy bitmap . . . 5
3 Single filter (SF) and multiple Bloom filter (MBF) summaries. The SF is a single Bloomier filter representing the type of an item. The MBF is an array of Bloom filters decreasing in size . . . 5
4 Counter value probabilities for different c. For $c > 1.6$ there is no effect on the counter distribution. For $c \leq 1.6$ the probability for higher counters increases . . . 7
5 Efficient Hash Table design overview. The offline update engine precomputes all updates. The online lookup engine is optimized for time/space efficient lookups. The online summary is not exact and compressed. The table is pruned. A small extra memory is used for counter and entry overflows . . . 10
6 Probability of smallest counter value in k' counters for different c . . . 11
7 Example Huffman tree for $\chi = 4$. . . . 15
8 Verifier hashing and buckets with multiple entries . . . 18
9 Real and expected counter frequencies . . . 21
10 Frequencies of used bits per compressed word for $|\omega| = 128$ . . . 22

List of Tables

1 Expected maximum load for different c . . . 8
2 Expected number of CAM entries for different c and χ with $n = 10^6$ inserted items. . . . 11
3 Parameter configurations p of the software simulations . . . 20
4 Entry distribution and expected maximum load . . . 21
5 Real and expected number of CAM entries . . . 21
6 Compression rate . . . 22
7 Comparison of on-chip requirements of different Bloom filter-based summaries . . . 23

List of Algorithms

1 Insert 13
2 Compress 16