Perfect Hashing for Network Applications

Yi Lu, Balaji Prabhakar
Dept. of Electrical Engineering
Stanford University
Stanford, CA 94305
yi.lu,balaji@stanford.edu

Flavio Bonomi
Cisco Systems
175 Tasman Dr
San Jose, CA 95134
flavio@cisco.com

***Abstract*— Hash tables are a fundamental data structure in many network applications, including route lookups, packet classification and monitoring. Often a part of the data path, they need to operate at wire-speed. However, several associative memory accesses are needed to resolve collisions, making them slower than required. This motivates us to consider minimal perfect hashing schemes, which reduce the number of memory accesses to just 1 and are also space-efficient.**

Existing perfect hashing algorithms are not tailored for network applications because they take too long to construct and are hard to implement in hardware.

This paper introduces a hardware-friendly scheme for minimal perfect hashing, with space requirement approaching 3.7 times the information theoretic lower bound. Our construction is several orders faster than existing perfect hashing schemes. Instead of using the traditional mapping-partitioning-searching methodology, our scheme employs a Bloom filter, which is known for its simplicity and speed. We extend our scheme to the dynamic setting, thus handling insertions and deletions.

I. Introduction

Hash tables constitute an integral part of many network applications. For instance, when performing IP address lookup at a router, one or more hash tables are queried to determine the egress port for an arriving packet. Hash tables are also used in packet classification, per-flow state maintenance, and network monitoring. Given the high operating speeds of today's network links, hash tables need to respond to queries in few tens of nanoseconds.

Despite the advance in the embedded memory technology, it is still not possible to accommodate a hash table, often with hundreds of thousands of entries, in an on-chip memory [1]. Therefore, hash tables are stored in larger but slower off-chip memories. It is very important to minimize the number of off-chip memory accesses and there has been much work on this recently. For example, Song et. al. [1] proposed a fast hash table based on Bloom filters [2] and the d-left scheme [3], while Kirsch and Mitzenmacher [4] proposed an on-chip summary that speeds up accesses to an off-chip, multi-level hash table, originally proposed by Broder and Karlin [5].

Our approach differs from the above in the construction phase: we construct a perfect hash function on-chip without consulting the off-chip memory. Moreover, the off-chip memory is a simple list storing each key and its corresponding item; there is no additional structure to the list. Finally, the space we use, both on-chip and off-chip, is smaller and our scheme adapts well to the dynamic situation, allowing us to perform insertions and deletions in constant time. A drawback of our scheme (and, indeed of any perfect hashing scheme) in the dynamic setting is that it requires a complete rebuild if the set of keys changes drastically. We come up with various heuristics for minimizing the probability of rebuilding.

A. Perfect Hashing

1) Definitions:

- *Perfect Hash Function:* Suppose that S is a subset of size n of the universe U. A function h mapping U into the integers is said to be *perfect* for S if, when restricted to S, it is injective [6].
- *Minimal Perfect Hash Function:* Let $|S| = n$ and $|U| = u$. A perfect hash function h is *minimal* if $h(S)$ equals $\{0, ..., n-1\}$ [6].

2) Performance Parameters:

- *Encoding size*: The number of bits needed to store the representation of h.
- *Evaluation time*: The time needed to compute $h(x)$ for $x \in u$.
- *Construction time*: The time needed to compute h.

Previous Work. Fredman and Komlós used a counting argument to prove a worst-case lower bound of $n \log e + \log\log u - O(\log n)$ for the encoding size of a minimal perfect hash function, provided that $u \geq n^{2+\epsilon}$ [7]. The bound is almost tight as the upper bound given by Mehlhorn is $n \log e + \log\log u + O(\log n)$ bits [8]. However, Mehlhorn's algorithm has a construction time of order $n^{\Theta(ne^n u \log u)}$.

One often-used approach to search for a minimal perfect hash function involves three stages: mapping, partitioning and searching. Mapping finds an injective function on S with a smaller range. Partitioning separates the keys into subgroups. And searching finds a hash value for each subgroup so that the resulting function is *perfect*. More details can be found in [9], [7].

Fredman, Komlós and Szemerédi constructed a data structure that uses space $n + o(n)$ and accommodates membership queries in constant time [10]. Fox et. al. [9] constructed an algorithm for large data sets whose encoding size is very close to the theoretical lower bound, i.e., around 2.5 bits per key. They also carried out experiments on 3.8 million keys and the construction time was 6 hours on a NeXT station. Separately, Hagerup and Tholey achieved $n \log e + \log\log u + o(n + \log\log u)$ encoding space, constant lookup time and $O(n + \log\log u)$ expected construction time using similar approaches [6].

The dynamic perfect hashing problem was considered by Dietzfelbinger et. al. [11]. Their scheme takes $O(1)$ worst-case time for lookups and $O(1)$ amortized expected time for insertions and deletions; it uses $O(n)$ space.

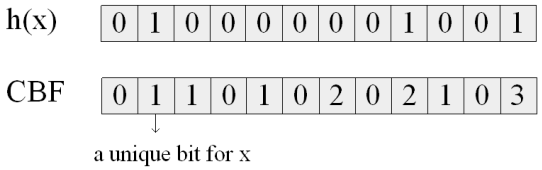

Fig. 1. Counting Bloom Filter and Unique Bits

B. Our Approach

Before setting out our approach, it helps to understand what, precisely, is involved in obtaining a minimal perfect hash function for a set S. Given U and $S \subset U$, there are many (hash) functions which map S onto the set $\{0, 1, ..., n-1\}$. However, a very very small subset of these functions are *injective* on S, and these are the minimal perfect hash functions of interest[1]. Thus, most approaches to finding minimal perfect hashes involve cleverly searching the set of all hash functions and hence are very time consuming.

Our approach is fundamentally different. By using counting Bloom filters (explained below), we recursively find injections for *random* subsets $S_1, S_2, ...$ of S onto a set of integers which is a constant factor larger than n. The key reason for our algorithm's simple construction is that it avoids searching. While Fox et. al. compute a minimal perfect hash function for 3.8 million keys in about 6 hours on a NeXT station, we are able to finish in 7.7 seconds, on a Pentium4 machine for the same number of keys. The construction time on the same machine is 125 milliseconds for a typical Ethernet address table with 100K entries.

We will first describe the counting Bloom filter and our particular way of using it.

Counting Bloom Filter and Unique Bits

Let U denote the universe of keys and let $S = \{x_1, x_2, \ldots, x_n\}$ be a subset of U.

A Counting Bloom Filter (denoted CBF) is a vector B of m counters. Available to us are k (random hash) functions $h_1(\cdot), ..., h_k(\cdot)$ each of which maps an $x \in U$ to a randomly chosen element of the set $\{e_1, ..., e_m\}$, where e_i is an m-bit vector with only its i^{th} bit set to 1. Let $h(x)$ be the sum of $h_1(x), ..., h_k(x)$. We refer to $h(x)$ as the "signature" of x.

Training a CBF involves setting the vector B to the sum of $h(x_1), ..., h(x_n)$, $x_1, ..., x_n \in S$. An example of $h(x)$ and the resulting CBF are shown in Figure 1.

Let the value of each counter be $c_1, ..., c_m$. As in a random ball-bin process, the distribution of c_i approximately follows a Poisson distribution. There is always a portion of positions that only one key is hashed to. We call such a position a *unique bit* for the key. A unique bit is illustrated in Figure 1.

Algorithm Overview

We use a sequence of CBFs of different sizes. The keys without a unique bit in the previous filter are carried over to the next. As a result of our construction, each key finds a hash function $h_i(\cdot)$ that puts it in a position that *no one* has occupied. Equivalently, the set of predetermined hash functions $h_1(\cdot), ..., h_k(\cdot)$ interpolate with one another to give a *perfect* hash functions h. This is not unlike the results of traditional approaches: Each subgroup of keys is assigned a hash value so that together they form a perfect hash function for the group. We do not explicitly split the keys into subgroups, but the CBFs *randomly* produces a subgroup for each hash function it uses.

Contributions

In Theorem 1, we show that as the number of CBFs goes to infinity, the encoding size goes to a minimum of $2en$ bits. This is 3.7 times the information-theoretic lower bound $n \log e + \log\log u - O(\log n)$, without the requirement $u \geq n^{2+\epsilon}$. A practical construction with a finite number of CBFs gives $8.6n$ bits as the encoding size.

More practical motivations for using CBFs include the ease of implementation in hardware and the small encoding size, which enables the use of a fast on-chip memory. Construction is orders faster than existing schemes as verified by simulation.

In addition, we extend the algorithm to the dynamic situation where encoding size only doubles from the static case, and remains $O(n)$. Both insertions and deletions are handled in constant time. Lookups consist of a single off-chip memory access most of the time and two in the worst case.

II. Minimal Perfect Hashing

Section II-A illustrates the architecture and algorithm of the CBF-based perfect hash. In Section II-B, we show that the minimum encoding size with the random approach goes to $2en$ as n becomes large. We also analyze the tradeoff between encoding size and maximum evaluation time. In Section II-C, we analyze the algorithm's construction time and failure probability. We complete the section with simulation results.

A. Description of Algorithm

1) Architecture: The perfect hash table includes an on-chip structure and a simple off-chip list, as illustrated by Figure 2. The on-chip structure contains d CBFs, $B_1, ..., B_d$, with possibly different sizes, in the top layer. There is an indicator layer in the middle, and an array of counters at the bottom. The indicator layer is a series of bits, with '1' corresponding to a value 1 in the CBF counter above, and '0' for all other values. The purpose of the indicator layer is to denote the presence of a unique bit. The counters in the bottom layer have range n, and are placed beneath every $(\log n)$th indicator bit. In Figure 2, $n = 16$. The off-chip list can accommodate exactly $|S|$ entries, where S is the set of keys we want to store.

2) Construction: Each CBF, B_i, is assigned k_i hash functions. We start by training the first CBF, B_1, with all keys in S, as described in I-B. The indicator layer beneath the first CBF is updated accordingly, i.e., with a '1' indicating a unique bit. A counter in the bottom layer records the number of '1's present in the indicator layer up to its position.

All keys in S are hashed again with the k_1 hash functions. If a key finds a unique bit b in B_1 belonging to its signature,

[1]Knuth [12] also notes the difficulty in computing minimal perfect hash functions. He estimates that to find h for the list of 31 frequently occurring English words, out of the universe of all English words, a search might need to examine 10^{43} possibilities.

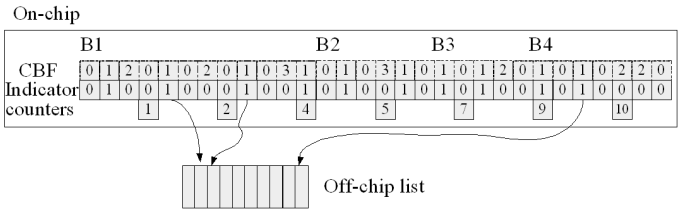

Fig. 2. Minimal Perfect Hash Function

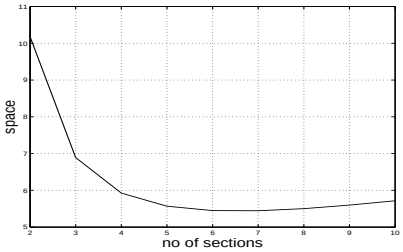

Fig. 3. Tradeoff between space and number of sections

it consults the closest bottom-layer counter before b and determines that b is the j^{th} unique bit. The key is hence inserted into he j^{th} slot of the off-chip list.

The keys without a unique bit in B_1 continue to train the CBF B_2, and the procedure repeats sequentially over all CBFs until all keys are accommodated. Once the construction is complete, only the indicator layer and the bottom counters are needed for subsequent lookups. The CBFs are only required for construction.

In the event where some keys are not accommodated, we denote it a "failure" and repeat the entire construction with a different set of hash functions. We will show in Section II-C that the probability of failure can be made exponentially small with a linear increase in the encoding size. A realistic application can be designed to have a very low failure probability and succeeds with one run of construction most of the time.

3) Lookup: Given a key x, we calculate its signature for each CBF. Once we encounter a unique bit b belonging to its signature, we consult the closest bottom-layer counter before b and calculate the unique bit index j. We retrieve the item from the j^{th} slot of the off-chip list.

B. Encoding Size

Minimum Encoding Size

Theorem 1 The minimum number of bits needed to provide n keys with one unique bit each, with random hashing, goes to en as n becomes large. It is achievable with an infinite number of CBFs with geometrically decreasing size, each with a single hash function.

Proof. Assuming that the hash outputs are perfectly random, the counter value c_i in a CBF converges to a Poisson distribution as n becomes large.

We start with one CBF, and let the CBF contain m counters. Recall that one counter in the CBF corresponds to one bit in the indicator layer in the final encoding. Assume k hash functions are assigned to the CBF. Hence the proportion of unique bits is $f = (nk/m)\exp(-nk/m)$. The proportion f is maximized with $nk/m = 1$, and $f_{max} = e^{-1}$.

Let the number of keys with a unique bit be s. When $k = 1$, $s = fm$; when $k > 1$, $s < fm$, since more than one unique bit might belong to the same signature in the latter case. For a fixed m, $s \leq f_{max}m$. Hence $s_{max} = f_{max}m = m/e$ when $k = 1$. This shows that using one hash function per CBF is the optimal solution.

Since m bits can provide unique bits for at most m/e keys, a minimum of en bits are required to accommodate n keys. The proof also shows how to achieve the minimum encoding size. With $k = 1$, setting $n = m$ for each CBF achieves f_{max}. Hence, letting $m_i = n(1 - e^{-1})^{i-1}$, i.e., each CBF having a size equal to the number of keys remaining, achieves the minimum. We can check that $\sum_{i=1}^{\infty} m_i = en$. ■

Based on the above theorem, the minimum size of the indicator layer is en for n keys. The total size of the counters in the bottom layer is also en since each counter contains $logn$ bits and the counters are $logn$ bits apart. In total, the minimum encoding size is $2en$.

Maximum Evaluation Time vs. Encoding Size

While the infinite sequence of CBFs provides the minimum-space solution, it is impossible to evaluate an infinite number of hashes. This prompts us to look at the tradeoff between encoding size and evaluation time in the finite case.

Since the sizes of CBFs in the infinite sequence is geometrically decreasing, the first few CBFs provide most of the unique bits. For this comparison, we distribute 95% of the entries over the first few CBFs, and over-provide in the last CBF to accommodate the remaining 5%. We focus our attention on the first few CBFs, assuming the over-provision in the last CBF works the same for all cases under comparison.

We consider the case where the number of hashes, k, in each CBF is 1, following the same argument as in Theorem 1. Thus the number of CBFs is the same as the maximum number of hashes to be evaluated. Also, we assume that the load on each CBF is the same, that is, $n_i/m_i = \lambda$, where n_i is the remaining number of entries for CBF i and m_i is the number of counters in CBF i. We will find the space needed when l CBFs are used to accommodate 95% of the entries.

The total number of keys accommodated by the first l CBFs is $t_l = (1 - (1 - e^{-\lambda})^l)n$. Letting $t_l = 0.95n$, we solve $\lambda = -\ln(1 - \sqrt[l]{0.05})$. Hence the proportion of unique bits $q = \lambda\exp(-\lambda)$, and the total space needed is

$$2n/q = -2n[(1 - \sqrt[l]{0.05})\ln(1 - \sqrt[l]{0.05})]^{-1}$$

The tradeoff between space $(2/q)$ and number of sections (l) is plotted in Figure 3. Clearly, $l = 4$ is the optimal tradeoff point between space and number of sections. $l = 7$ is the minimum-space point, which is the same as the answer obtained by equating $(1-(1-e^{-1})^l)$ to 0.95. In summary, a little increase in space reduces the maximum number of hash evaluations by almost half. A similar tradeoff can be exploited in general.

C. Construction Time and Failure Probability

Since we choose to over-provide in the last CBF to accommodate all the remaining entries, we are interested in

the amount of space needed in the last section so that the probability of failure is small.

Theorem 2 Let n be the number of keys remaining for the last section, and m be the space assigned for the section. Then the probability of failure can be made double-exponentially small in m, and the optimal number of hash functions in this section is $k^* = \frac{m}{n} \ln 2$.

Proof. Assuming the last section has k hash functions. For one particular item, the probability of not finding a unique position is

$$P = [1 - (1 - \frac{1}{m})^{(n-1)k}]^k \rightarrow (1 - e^{-\frac{kn}{m}})^k$$

A failure occurs when at least one key cannot find a unique position, so

$$P_{fail} = 1 - (1 - (1 - e^{-\frac{kn}{m}})^k)^n \rightarrow 1 - e^{-n(1-e^{-\frac{kn}{m}})^k}$$

$k^* = \frac{m}{n} \ln 2$ minimizes P_{fail}. The optimized $P_{fail} = 1 - \exp(-n/2^{k^*}) = 1 - \exp(-n(2^{(-ln2/n)})^m)$, hence doubly exponential in m. ∎

The average construction time is closely related to the failure probability. Construction successful in one pass requires $T = O(n)$. However, the actual construction time follows a geometric distribution with parameter $(1 - P_{fail})$. So the average construction time $\overline{T} = T/(1 - P_{fail})$. The fast construction of our algorithm requires P_{fail} to be small. An actual value of P_{fail} is given in section II-D.

D. Simulation Results

The simulation is run on a Pentium4 machine with randomly generated keys. We present a design example to illustrate experimental failure probability, unique bits distribution and average construction time for a large number of keys.

a) Design Specification: Since 4 CBFs give the optimal tradeoff point for 95% entries (discussed in Section II-B), we use a total of 5 CBFs. The corresponding proportion of unique bits is 0.3375.

This gives a space ratio of $1.56 : 0.74 : 0.35 : 0.17 : 1.5$, with a total size of $8.6n$. The number of hashes for the 5 CBFs are 1, 1, 1, 1, 12 respectively.

b) Failure Probability: The experimental failure probability is obtained by running the algorithm with 1000 keys over 10^5 runs. We get $P_{fail} = 0.0012$. This translates into an average construction time $\overline{T} = T/0.9988 \approx T$, where T is the duration of a successful construction with no repetition.

c) Unique Bits Distribution: The number of unique bits in the first four CBFs is very close to what it is designed to be, i.e., 0.3375 of the size of the section. This verifies the correctness of the approximated Poisson distribution. Here are data from arbitrary runs with different number of keys.

Number of Keys	1000	1000000	3800000
Section 1	526	526286	2001952
Section 2	258	249887	948100
Section 3	107	118137	448368
Section 4	63	56810	215679
Section 5	46	48880	185901

TABLE I

UNIQUE BITS DISTRIBUTION FOR DIFFERENT NUMBER OF KEYS

d) Construction Time: Fox et. al. performed experiments on 3.8 million keys, and their algorithm completes in about 6 hours. We run our simulation on 3.8 million keys, with a C program on a Pentium4 machine 100 times. The average time for a successful construction is 7.73 seconds using the "clock" command. It will be significantly faster if implemented in hardware.

For a typical Ethernet address table, the number of keys are in the hundreds of thousands. For a $100K$ keys, the algorithm completes on a Pentium4 machine in 125 milliseconds. Again it can be reduced further in hardware.

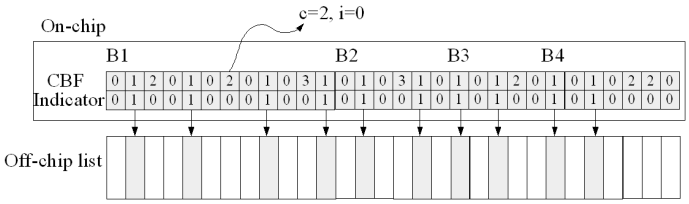

Fig. 4. Dynamic Perfect Hash Function

III. EXTENSION: DYNAMIC PERFECT HASHING

A minimal perfect hash function is specifically optimized for one set S in order to achieve space efficiency. The static nature of the minimal perfect hash makes it perform poorly when S is dynamically changing. We propose an extension of the unique bits idea to the dynamic setting, replacing the minimal perfect hash function with a non-minimal perfect hash function. As a "perfect" hash, it retains an $O(1)$ lookup time.

A. Description of Algorithm

1) Architecture: The architecture of a dynamic perfect hash function is illustrated in Figure 4. The CBF layer and the indicator layer are the same as in the static case. There is no additional counter layer, and both CBF and indicator are retained at all times. The major change is in the off-chip list: Instead of size $|S|$, the list now contains as many slots as the number of bits in the indicator layer. There is also a small CAM for accommodating collisions in a relatively rare event (not shown in figure).

2) Operations:

a) Insertion: At insertion, a key compares the non-negative bits in its signature with the corresponding CBF counter sequentially. At each comparison, it takes action according to the counter value c at the position (illustrated in Figure 4). Let the corresponding indicator bit be i.

Case 1: $c = 0$. This indicates that an empty slot in the off-chip list is found. Change $c = 1$ and $i = 1$, and the item is inserted into the corresponding slot.

Case 2: $c = 1$. This indicates the slot is occupied by another entry and a collision has occurred. There is an option in the algorithm to *rehash*, i.e., change $c = 2$ and $i = 0$. Both keys are re-inserted into the CBF. If they meet other collisions in the process, rehash happens recursively. A rehash is successful if all keys involved find a unique position.

To avoid non-deterministic time for insertion, we limit the levels of rehash to 2. When a rehash fails, the item is entered into the external CAM.

Case 3: $c > 1$. Increment c and move to the next CBF. If this is the last CBF, the item is entered into the CAM.

b) Lookup: In normal situations, the index of the first unique bit for a key yields the correct index into the off-chip list. When there was a collision, or no unique bits were found for the key at insertion, the lookup is redirected to the CAM.

c) Deletion: A lookup is performed first. The entry is erased from the off-chip memory, or the CAM. Its signature bits before its unique bit are subtracted from the CBF. And the indicator for its unique bit is changed to 0.

d) Rebuild: If the CAM overflows, the whole structure is rebuilt just as in the construction process of the minimal perfect hashing.

e) Load balancing: In order to distribute the load over all CBFs, each key chooses a random CBF (using hashing) as *its first* CBF. The insertion process continues sequentially, and wraps around until it covers all CBFs.

B. Performance Evaluation

1) Space: Both the counting Bloom filter and the indicator layer have number of bits equal to a multiple of n. So the space used is $O(n)$. In the simulated design that follows, we use 4 CBFs and each CBF has n counters with depth 4. It consumes 20 bits per key.

2) Insertion: Due to limitation of space, we omit the calculation and instead present numbers for the probability of collision (P_c) and rehash failure (P_r). Both have analytical formulae in terms of the load factor $\lambda = nk/m$, where n is the number of currently active flows, k is the number of hashes in one CBF, and m is the total space. Let the number of CBFs be l.

For $l = 5$ and $\lambda = 0.25$, $P_c = 0.2$ and $P_r = 0.1$. Most of the time, the system does not operate with peak load. At one-fifth the peak load, $\lambda = 0.05$, $P_c = 0.047$ and $P_r = 0.005$.

We design $\lambda_{max} = 0.25$. Hence an empty slot in the off-chip memory is found at least 90% of the time. For the rest 10%, the entry is inserted into the more power-consuming CAM. In both cases, the insertion involves exactly one access to the slower memory.

3) Lookup / Deletion: The complexity of deletion is the same as that of lookup. In most cases, the process involves one access to the off-chip memory *or* the CAM. The only case where there is one access to the memory *and* the CAM is when a collision occurred at insertion, and attempts at rehash failed. Hence, with probability P_r, the process needs two accesses to slower memory, and otherwise one access suffices.

One heuristic we use is moving an entry from the CAM to the off-chip memory, when it finds a unique bit later. This lowers the number of CAM lookups and the probability of a CAM overflow.

C. Trace-driven Simulation

A good application of the dynamic perfect hashing is the flow lookup table in routers. Hence we run the algorithm on a 5 million packet CAIDA trace collected at 9:20am, Aug 14, 2002. There are a total of 417931 flows. The number of concurrently active flows reaches a maximum of 54853.

Since load balancing is used, each section is designed to be the same size, 55000 bits, which is slightly more than the maximum flow number. The total space for the encoding is 1.1Mbits, and there are a total of $220,000$ off-chip slots. The first 3 CBFs have 1 hash, while the last one has 2 hashes. The CAM is assigned a size 2.5% of the maximum flow number. The table below tabulates the experiment output:

	Number	Percentage
Total Insertion	417931	
Total Lookup	4684091	
Insertion into CAM	14799	3.54%
Lookups in CAM	67548	1.44%
Average Hash Check at Insertion	1.52	
Average Hash Check at Lookup	1.57	
Flows Moved from CAM	2729	0.65%

TABLE II

PERFORMANCE PARAMETERS OF DYNAMIC PERFECT HASHING ON TRACE

A rebuild is not necessary in this experiment. Note that despite the use of 5 hashes, on average a unique bit is found between the 1^{st} and 2^{nd} hashes.

IV. CONCLUSION

The paper presented a new approach to minimal perfect hashing via counting Bloom filters. By generating random subgroups for pre-determined hash functions, we avoid the need of searching and as a result, speed up the construction. In the limit, our encoding size is 3.7 times the information-theoretic lower bound. The resulting construction is hardware-friendly and fits the need of high-speed network applications well.

REFERENCES

[1] H. Song, S. Dharmapurikar, J. Turner, and J. Lockwood, "Fast hash table lookup using extended bloom filter: An aid to network processing," *SIGCOMM, (Philadelphia), Aug*, 2005.

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

[3] Andrei Broder and Michael Mitzenmacher, "Using multiple hash functions to improve ip lookups," *Proceedings of IEEE Infocomd*, 2001.

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

[5] A. Broder and A. Karlin, "Multilevel adaptive hashing," *Proceedings of the 1st ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pp. 43–53, 1990.

[6] Torben Hagerup and Torsten Tholey, "Efficient minimal perfect hashing in nearly minimal space," *STACS 2001, LNCS 2001*, pp. 317–326, 2001.

[7] M. Fredman and J. Komlós, "On the size of separating systems and families of perfect hash functions," *SIAM J. Alg. Disc. Meth*, , no. 5, pp. 61–68, 1984.

[8] K. Mehlhorn, "Data structures and algorithms, vol. 1: Sorting and searching," 1984.

[9] Edward A. Fox, Qi Fan Chen, and Lenwood S. Heath, "A faster algorithm for constructing minimal perfect hash functions," *15th Ann Int'l SIGIR Denmark*, 1992.

[10] M. Fredman, J. Komlós, and E. Szemeredi, "Storing a sparse table with $o(1)$ worst case access time," *Journal of the ACM*, vol. 31, no. 3, pp. 538–544, July 1984.

[11] Martin Dietzfelbinger, Annar Karlin, Kurt Melhorn, Friedhelm Meyer auf der Heide, Hans Rohnert, and Robert E. Tarjan, "Dynamic perfect hashing: Upper and lower bounds," *SIAM J. Computing*, 1990.

[12] D. E. Knuth, "The art of computing programming. volume 3: Sorting and searching," pp. 506–507, 1973.