

Rank-Indexed Hashing: A Compact Construction of Bloom Filters and Variants

Nan Hua, Haiquan (Chuck) Zhao
School of Computer Science
Georgia Tech
Email: nanhua, chz@cc.gatech.edu

Bill Lin
Dept. of ECE
UCSD
Email: billin@ece.ucsd.edu

Jun (Jim) Xu
School of Computer Science
Georgia Tech
Email: jx@cc.gatech.edu

Abstract—Bloom filter and its variants have found widespread use in many networking applications. For these applications, minimizing storage cost is paramount as these filters often need to be implemented using scarce and costly (on-chip) SRAM. Besides supporting membership queries, Bloom filters have been generalized to support deletions and the encoding of information. Although a standard Bloom filter construction has proven to be extremely space-efficient, it is unnecessarily costly when generalized. Alternative constructions based on storing fingerprints in hash tables have been proposed that offer the same functionality as some Bloom filter variants, but using less space. In this paper, we propose a new fingerprint hash table construction called Rank-Indexed Hashing that can achieve very compact representations. A rank-indexed hashing construction that offers the same functionality as a counting Bloom filter can be achieved with a factor of three or more in space savings even for a false positive probability of just 1%. Even for a basic Bloom filter function that only supports membership queries, a rank-indexed hashing construction requires less space for a false positive probability as high as 0.1%, which is significant since a standard Bloom filter construction is widely regarded as extremely space-efficient for approximate membership problems.

I. INTRODUCTION

A Bloom filter [3] is a space-efficient randomized data structure for approximating a set in order to support membership queries. Although a Bloom filter may yield false positives, saying an element is in the set when it is not, it provides a very compact representation that can be configured to achieve sufficient accuracies for many applications. For a false positive probability of ϵ , an optimal configuration only requires $1.44(\log 1/\epsilon)$ bits per element, independent of the number of elements in the set. For example, to achieve a false positive probability of $\epsilon = 1\%$, only 10 bits of storage per element is required.

In recent years, there has been a huge surge in the popularity of Bloom filters and variants, especially for network applications [6]. One variant is the counting Bloom filter (CBF) [10], which allows the set to change dynamically via insertions and deletions of elements. Other generalizations of Bloom filters include Spectral Bloom Filter [8], which can

encode approximate counts, the Approximate Concurrent State Machine [4] (also called a stateful Bloom filter), which can encode state information, and the Bloomier filter [7], which can encode arbitrary functions by allowing one to associate values with a subset of the domain elements. In general, many Bloom filter variants that permit the association of values to elements mainly differ in the way how they encode and interpret the values associated.

Although a standard Bloom filter construction is very space-efficient for simple membership queries, it is actually rather inefficient when generalized to support deletions or the encoding of information. In particular, in the standard Bloom filter construction, an array of m bits is used to represent a set S of n elements, where m is chosen to be sufficiently large to ensure a small false positive probability. For example, for a false positive probability of $\epsilon = 1\%$, m is chosen to be 10 times n , resulting in an amortized storage cost of 10 bits per element. When this standard construction is generalized to encode additional information, an array of m locations is used instead of bits. For example, in the counting Bloom filter application where a counter is associated with each location to support insertions and deletions, four counter bits are often used to provide a sufficient level of accuracy [10]. However, this blows up the storage requirement by a factor of four over a standard Bloom filter.

Alternatively, it is well-known that a hash table construction with fingerprints can be used to provide the same functionality as a Bloom filter [6]. In particular, if the set S is static and a perfect hash function can be constructed for a hash table with n locations, then storing a fingerprint with only $\lceil \log 1/\epsilon \rceil$ bits for each element at the corresponding location would suffice to achieve a false positive probability of ϵ . Moreover, for Bloom filter generalizations that support values associations, the encoding of the additional information only needs to be stored once at the corresponding hash table location rather than requiring the encoding of information across multiple locations as required in a standard Bloom filter construction, resulting in substantial savings in space. However, unfortunately, perfect

hashing is very difficult to construct and does not support dynamically changing sets.

In this paper, we propose a new fingerprint hash table construction called *Rank-Indexed Hashing* that provides a compact replacement for Bloom filters, counting Bloom filters, and other Bloom filter variants. Conceptually, our starting point is a conventional chaining-based hash table scheme. However, our proposed solution avoids the costly overhead of pointer storage by employing an efficient indexing scheme called *rank-indexing*. Actually rank-indexing is not a brand-new technique and has been employed by [9] and [15] to construct compact data structures. The name “rank-indexing” here is from [14], which summarizes this kind of operation as rank operation. Using rank-indexed hashing construction, we show that it is possible to outperform a standard Bloom filter construction in storage cost for false positive probabilities at or below just 0.1%, which is significant since a standard Bloom filter construction is widely regarded as a very space-efficient data structure for approximate membership query problems, and it is often desirable to have a false positive probability smaller than 0.1% in many applications.

For the counting Bloom filter application, the rank-indexed hashing construction is able to outperform a standard counting Bloom filter construction in storage cost by a factor of three for a false positive probability of just 1%, and it is able to outperform a recently proposed fingerprint hash table construction called *d-left hashing* [5] in storage cost by 27% at the same false positive probability. Similar storage cost benefits are expected for other Bloom filter variants. Especially for network applications, smaller storage requirement is a central design metric because Bloom filters are often implemented using relatively scarce and expensive (on-chip) SRAM. Although SRAM capacity continues to increase, the rate of traffic growth continues to outpace transistor density, leading to an ever increasing need to reduce storage requirements.

The rest of this paper is organized as follows. Section II summarizes background material on Bloom filters and fingerprint hash table constructions. Section III describes our rank-indexed hashing method. Section IV establishes tail bound probabilities that allow us to bound and optimize the storage cost. Section V evaluates our scheme by presenting numerical results under various parameter settings, including results for both standard and counting Bloom filters. Section VI concludes the paper.

II. BACKGROUND

A. Bloom Filters, Counting Bloom Filter and Compressed Bloom Filter

A Bloom filter represents a set S of n elements from a universe U using an array of m bits, denoted by $\phi[1], \dots, \phi[m]$. Initially, all positions $\phi[i]$ are set to 0. A Bloom filter uses a

group of k independent hash functions h_1, \dots, h_k with range $[m] = \{1, \dots, m\}$. Each hash function independently maps each element in the universe to a random number chosen uniformly over the range. For each element $x \in S$, the bits $\phi[h_i(x)]$ are set to 1 for $1 \leq i \leq k$. To check if an item x is in S , we check whether all $h_i(x)$ are set to 1. If not, then by construction, x is not a member of S . If all $h_i(x)$ are set to 1, then x is assumed to be in S , which may yield a *false positive*.

The false positive probability for an element not in the set can be computed as

$$\epsilon = (1 - (1 - 1/m)^{kn})^k \approx (1 - e^{-kn/m})^k.$$

The storage requirement to satisfy a given false positive probability ϵ is minimized when

$$m = \frac{1}{\ln 2} kn \approx 1.44kn.$$

For example, when $m/n = 10$ and $k = 7$, the false positive probability is about $2^{-7} \approx 0.008$.

A Bloom filter allows for easy insertion, but not deletion. Deleting an element from a Bloom filter cannot be done simply by reverting the corresponding ones back to zeros since each bit may correspond to multiple elements. Deletion can be handled by using a counting Bloom filter (CBF) [10], which uses an array of m counters instead of bits. Counters are incremented on an insertion and decremented on a deletion. The counters are used to track approximately the number of elements currently hashed to the corresponding locations. To avoid overflow, counters must be chosen to be large enough. For most applications, 4 bits per counter have been shown to suffice [10]. However, the obvious disadvantage of counting Bloom filters is that they appear to be quite wasteful of space. Using counters of 4 bits blows up the storage requirement by a factor of four over a standard Bloom filter, even though most counters will be zero. For example, with $k = 7$, $4m/n = 40$ bits per element are needed to achieve a false positive probability of about $2^{-7} \approx 0.008$.

Compressed Bloom Filter [11] is in fact a special application of standard Bloom Filter, rather than a new data structure. It is discovered in [11] that, when the space of standard Bloom Filter is minimized to $1.44kn$, its compressed size, i.e. its information entropy, is also maximized at the same time. In another word, if the space of standard Bloom Filter is not optimized to the minimum, i.e. greater than $1.44kn$, its size after compression would be smaller than $1.44kn$. When the original space is infinitely large, the compressed size would reach the limit kn . Hence, we could sacrifice the original space for smaller compressed size. This property is favored for some online applications such as P2P sharing where the transferring cost is much more important.

B. Fingerprint Hash Table Construction

An alternative construction of Bloom filters and counting Bloom filters is to use a hash table with *fingerprints*. It is well known that if the set S is static, then one can achieve essentially optimal performance by using a perfect hash function and fingerprints [6]. That is, we can find a perfect hash function

$$P : U \rightarrow [n]$$

that maps each element $x \in S$ to a unique location in an array of size n , where $[n]$ denotes the range $\{1, \dots, n\}$. Then, we simply need to store at each location a fingerprint with $\lceil \log 1/\epsilon \rceil$ bits that is computed according to some hash function F . A query on x requires computing $P(x)$ and $F(x)$, and checking whether the fingerprint stored at $P(x)$ matches $F(x)$. When $x \in S$, a correct response is given. But when $x \notin S$, a false positive occurs with probability at most ϵ . This perfect hashing approach achieves the optimal space requirement of $\lceil \log 1/\epsilon \rceil$ bits per element. However, the problem with this approach is that it does not allow the set S to change dynamically, and perfect hash functions are generally very difficult to compute for most applications.

An alternative to perfect hashing is to use an *imperfect* hash function. Suppose we use a single (imperfect) hash function $H : U \rightarrow [B] \times [R]$ to hash the elements in S to a hash table with B buckets. For each element $x \in S$, $H(x)$ returns two parts. The first part in the range $[B]$ corresponds to the bucket index in which the element should be placed. The second part in the range of $[R]$, referred to as the *remainder*, corresponds to a *compressed fingerprint* that gets stored in the corresponding bucket. Using a single hash function, it is possible (and likely) that two different elements x and y are mapped to the same bucket, resulting in a *collision*. One way to resolve collisions is to allocate a fixed number of *cells* per bucket so that the maximum load per bucket is no more than this fixed number with high probability. However, the distribution of load using a single hash function can fluctuate dramatically across buckets, leading to a lot of wasted space.

Another way to resolve collisions is to maintain a dynamically allocated *linked list* of fingerprints that have been hashed to the same bucket, as shown in Figure 1. However, this conventional *chained hashing* approach is also rather inefficient in that the extra pointer storage (required for each fingerprint) is very expensive. For $n = 1$ million elements and a desired false positive probability of $\epsilon = 2^{-7}$, 7 bits per element would suffice assuming perfect hashing, but another $\lceil \log n \rceil = 20$ bits would be required on average per element to implement pointers, or another 40 bits per element to implement doubly linked list pointers to support deletion, increasing the storage requirement by nearly seven-fold.

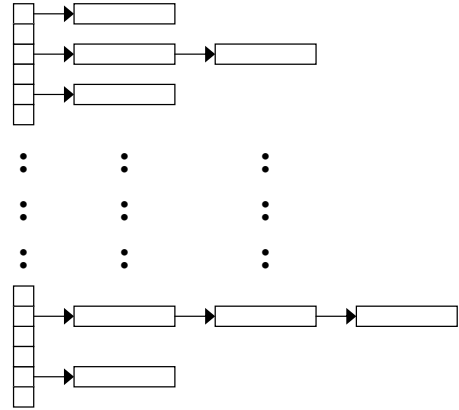


Fig. 1. Hash table with chaining.

A third way that has been recently proposed [5] is to use a balanced allocation approach due to Vöcking [16] called *d-left hashing*. By splitting a hash table into multiple equally-sized subtables, and placing elements in the least-loaded subtable, *d-left hash tables* can be dimensioned statically so that the average load per bucket is close to the maximum load. A good configuration suggested in [5] is to use 4 subtables with a fixed allocation corresponding to an expected maximum load of 8 fingerprints per bucket, with an expected average load of 6 fingerprints per bucket. To check if x is in S , *d-left hashing* requires checking x against all fingerprints stored in the corresponding buckets across all the subtables, requiring the retrieval of $4 \cdot 8 = 32$ fingerprints, with matching on average against $4 \cdot 6 = 24$ fingerprints expected. To achieve a desired false positive probability of $\epsilon = 2^{-7}$, each cell must store a fingerprint with $\lceil \log 24/\epsilon \rceil = \lceil \log 24/2^{-7} \rceil = 12$ bits, adding 5 more bits per element to the “ideal” case of $\lceil \log 1/\epsilon \rceil = 7$ bits, which is significant. Further accounting for the expected fraction of unused cells corresponding to the ratio of expected maximum load over average load, $(8/6) \cdot 12 = 16$ bits per element would be required, increasing over the ideal storage requirement of 7 bits by over two-fold.

III. RANK-INDEXED HASHING

A. Basic Idea

In this section, we describe our proposed *rank-indexed hashing* approach. Conceptually, our starting point is a conventional chaining-based hash table scheme. However, we employ a *two-level* indexing scheme by using a single hash function $H : U \rightarrow [B] \times [L] \times [R]$ that hashes each element in S into three parts; i.e., for each element $x \in S$, $H(x) = (b, \ell, r)$. As before, the first part b in the range of $[B]$ corresponds to the bucket index in which the element should be placed. The second part ℓ in the range of $[L]$ corresponds to a *hash chain* location within a bucket. This is conceptually depicted in Figure 2. The third part r in the range of $[R]$ corresponds to the

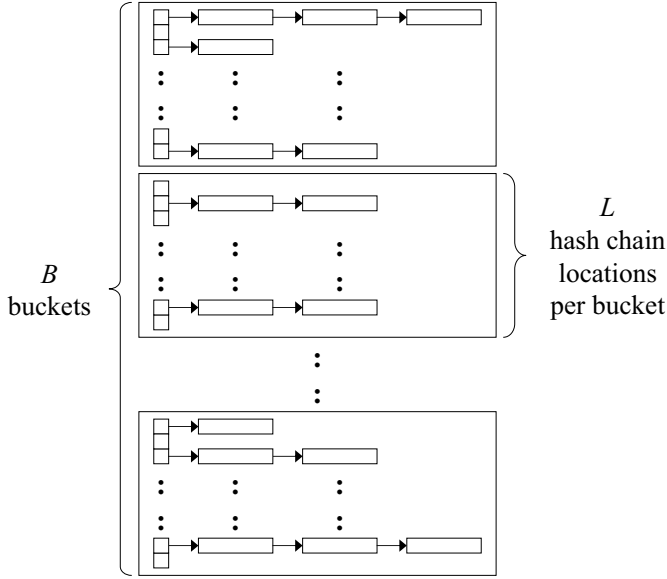


Fig. 2. Buckets and hash chain locations.

(compressed) fingerprint that gets stored in the corresponding hash chain location. In general, the number of buckets B times the number of hash chain locations per bucket L can be different than the number of elements n . We use

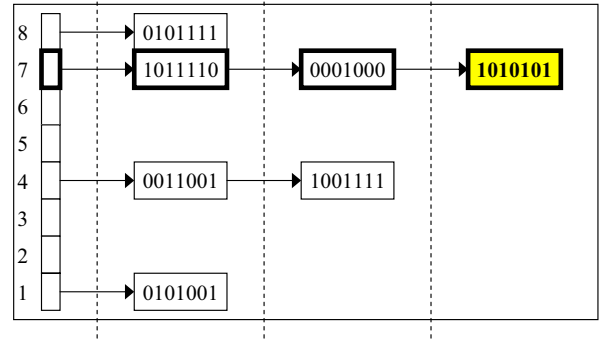
$$\lambda = \frac{n}{BL}$$

to denote the expected average load per hash chain location. If $\lambda = 1$, then a fingerprint with $\lceil \log 1/\epsilon \rceil$ bits suffices to achieve a false positive probability of ϵ . In general, $\lceil \log \lambda/\epsilon \rceil$ bits are needed to achieve a false positive probability of ϵ .

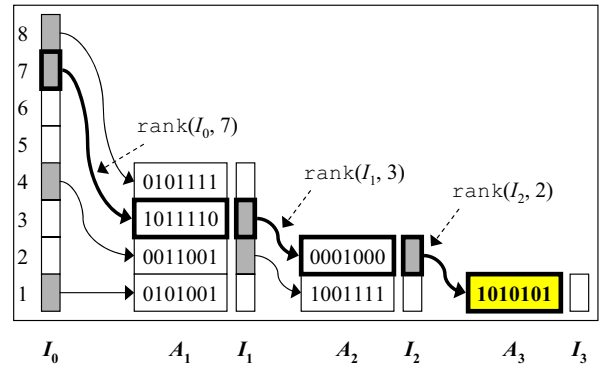
Given λ , the expected average load per bucket is simply $\lambda \cdot L$. Intuitively, each bucket is dimensioned to store a fixed number of $Z > (\lambda \cdot L)$ fingerprints such that the actual load per bucket is less than or equal to Z in a large fraction of buckets. We defer to Section III-C to discuss the handling of cases when the actual load exceeds Z fingerprints. For the moment, we will assume Z is chosen to ensure that actual load is less than or equal to Z with high probability.

A key innovation in our proposed approach is an indexing method that allows us to efficiently realize *dynamic chaining* at each hash chain location *without* the costly overhead of pointer storage. We call this method *rank-indexing*, and this method is illustrated in Figure 3. Consider a bucket with $L = 8$ hash chain locations, as shown in Figure 3(a). In this example, suppose the longest chain has three fingerprints. We can conceptually partition the fingerprints into three levels, in accordance to the depths of the fingerprints in the corresponding chains. We then *pack* the fingerprints at each level together in a corresponding contiguous subarray of fingerprints.

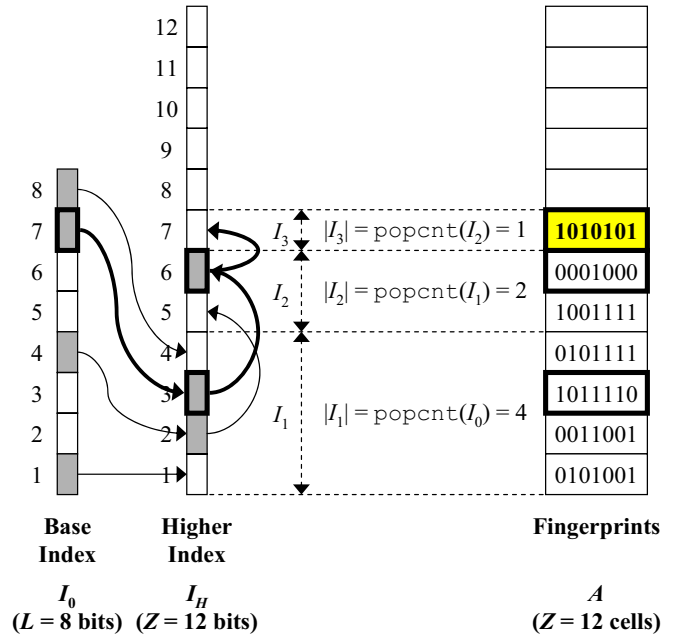
To locate fingerprints in a bucket, we maintain an index bitmap I . Conceptually, I is divided into multiple parts



(a) A bucket.



(b) Rank-indexing.



(c) Packed bucket organization.

Fig. 3. Rank-indexed hashing.

I_0, I_1, \dots, I_d , one part for each level of subarray. Suppose we want to query for the fingerprint “1010101” at the hash chain location $\ell = 7$. We first check $I_0[7]$ to determine if there are fingerprints stored at $\ell = 7$. If $I_0[7]$ is set to 1, as shown in a shaded box in Figure 3(b), then it means there is a non-empty chain at $\ell = 7$. If there are no fingerprints at location ℓ , then the corresponding $I_0[\ell]$ would be set to 0, which is shown as a clear box.

In the example shown in Figure 3(b), the first fingerprint at $\ell = 7$ is located at $A_1[3]$. Rather than expending costly memory to store an explicit pointer from $I_0[7]$ to $A_1[3]$, we *dynamically* compute the location by using an operator called $\text{rank}(s, i)$, which returns the number of ones in the range $s[1] \dots s[i]$ in the bit-string s . Our proposed method exploits the fact that the rank operator can be efficiently implemented using hardware-optimized instructions that are increasingly available in modern processors. In particular, the rank operator can be efficiently implemented by combining a bitwise-AND instruction with another operation called $\text{popcount}(s)$, which returns the number of ones in the bit-string s . Fortunately, the popcount instruction is becoming increasingly common in modern microprocessors and network processors. For example, current generations of 64-bit x86 processors have this popcount instruction built-in [1], [2]. As we shall see in the evaluation section, very compact constructions can be achieved by setting $L \leq 64$. Since $|I_0| = L$, we can directly compute $\text{rank}(I_0, \ell)$ using hardwired 64-bit instructions.

Continuing with the example shown in Figure 3(b), we can compute the location of the first fingerprint at location $\ell = 7$ in A_1 by invoking $a_1 = \text{rank}(I_0, 7)$, which will return $a_1 = 3$. We can then match against the fingerprint stored at $A_1[a_1] = A_1[3]$. Similarly, we can check if the chain has ended by checking $I_1[3]$. If $I_1[3] = 1$, then we can locate the next fingerprint in A_2 by computing $a_2 = \text{rank}(I_1, 3) = 2$. Given that $I_2[2] = 1$, we can locate the next fingerprint in the chain in A_3 by computing $a_3 = \text{rank}(I_2, 2) = 1$. We see that the fingerprint “1010101” is found at $A_3[1]$. Further, we see that the chain has ended since $I_3[1] = 0$. In general, if a chain has extended beyond level j (i.e., $I_j[a_j] = 1$), then the index location to the next subarray A_{j+1} can be simply computed as $a_{j+1} = \text{rank}(I_j, a_j)$. Observe that $|I_0| \geq |I_1| \geq \dots \geq |I_d|$. Therefore, if we use $L \leq 64$, then all rank operations can be directly performed using 64-bit instructions.

B. Packed Bucket Organization

Thus far above, we provided a high-level description of the idea of rank-indexed hashing. In practice, different buckets will have varying load distributions at the different hash chain locations, which means the number of fingerprints at each level will also fluctuate across buckets. To take advantage of statistical multiplexing, our bucket organization packs the

subarrays of fingerprints together into a contiguous array A . This is depicted in Figure 3(c). This way, we can dimension a bucket to store a *fixed* number of Z fingerprints. The Z fingerprint locations can be shared by all the fingerprints in a bucket regardless of their hash chain location (i.e., regardless of which chain a fingerprint is located). We can also fix the size of the bitmap $I = I_0 I_H$ to $L+Z$ bits with L bits set aside for I_0 and Z bits set aside for $I_H = I_1 I_2 \dots I_d$. I_0 is referred to as the base index, and I_H is referred to as the higher index. In the example shown in Figure 3(c), L is fixed at 8 and Z is fixed at 12.

Using this packed bucket organization, the size of a subarray A_j and the corresponding bitmap I_j may dynamically change. The current size of a subarray can be readily computed using the popcount operator. For example, the size of A_1 and I_1 shown in Figure 3(c) can be computed as $\text{popcount}(I_0)$. In general, the size of A_{j+1} and I_{j+1} can be computed as $\text{popcount}(I_j)$.

On insertion or deletion, a fingerprint may be added or removed, respectively (e.g., in the case of a counting Bloom filter, a fingerprint is only removed when the corresponding counter has been decremented to zero). The addition or removal of a fingerprint can increase or decrease the size of a subarray by one. To maintain the packed bucket representation, up to $Z - 1$ fingerprints may have to be shifted in the worst-case. Here again, we can exploit available 64-bit instructions in modern processors to expedite this shifting process. For example, if 8 bits are used to store a remainder, then a 64-bit instruction can shift eight fingerprints at a time. Similarly, for the bitmap I_H , up to $Z - 1$ bits may have to be shifted in the worst-case, but 64 bits can be shifted at a time using 64-bit instructions.

C. Quasi-Dynamic Bucket Sizing

So far, we have assumed that the fixed number of Z fingerprints allocated to each bucket is chosen to ensure that the actual load of a bucket is less than or equal to Z with high probability. In Table III-C, we consider an example configuration in which each bucket has $L = 64$ hash chain locations and $\lambda = 0.875$. In this case, $\lambda \cdot L = 56$ is also the expected average load. Each table entry indicates the fraction of buckets that are expected to have actual loads greater than the corresponding specified threshold. We see that if we set $Z = 64$, then only 13% of the buckets are expected to overflow. If we set $Z = 74$, then just 1% of the buckets are expected to overflow. And if we set $Z = 128$, then no buckets is expected to overflow with high probability.

Suppose we set $Z = 64$ fingerprint cells. This means that the expected fraction of unused fingerprint cells is just $(64/56) = 1.14$, providing a high degree of space efficiency. However, in this example, we need a way to accommodate the

TABLE I

TABLE ENTRIES GIVE THE FRACTION OF BUCKETS WITH ACTUAL LOAD GREATER THAN SOME THRESHOLD FOR A BUCKET WITH $L = 64$ HASH CHAIN LOCATIONS AND $\lambda = 0.875$. ASYMPTOTIC TAIL BOUNDS ARE PROVIDED FOR THREE EXAMPLE THRESHOLDS.

Threshold	Fraction
Load > 64	0.13
Load > 74	0.01
Load > 128	8e-17

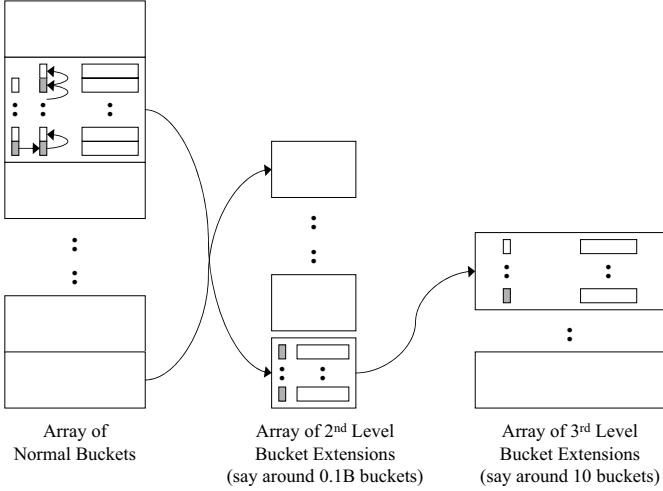


Fig. 4. Quasi-dynamic bucket sizing via bucket extensions.

possibility that 13% of buckets might overflow. Conceptually, when a bucket overflows, meaning that all Z fingerprint slots are already occupied on a new insertion, we dynamically *extend* the bucket size by dynamically linking another “chunk” of memory to it. We refer to these chunks of memories as *bucket extensions*. This is illustrated in Figure 4.

To make the example more concrete, suppose $B = 1000$. Then we can statically allocate memory for $J_2 = 0.13B = 130$ second-level buckets. However, these second-level buckets are much smaller than the first-level buckets (and there are fewer number of them). In particular, referring to Table III-C, we see that just 1% of the buckets are expected to have actual loads greater than 74. Suppose we dimension these second-level buckets to hold $Z_2 = 10$ fingerprints. Here, we will use Z_1 rather than Z to indicate the dimensioning of the first-level buckets: i.e., $Z_1 = 64$. Then, for buckets that have been extended to the second-level, $Z = Z_1 + Z_2 = 64 + 10 = 74$ fingerprints are available. In addition, the second-level buckets will provide an additional $Z_2 = 10$ bits for extending the higher index I_H . Only the first-level bucket needs to store the base index I_0 . The linkage from a first-level bucket to a second-level bucket can be implemented simply as a memory pointer, which is affordable since they are small relative to the size of the buckets, and bucket extensions only occur with a

small fraction of buckets. An extension flag can be used to indicate that a bucket has been extended.

With the availability of second-level buckets, with $Z = Z_1 + Z_2 = 74$ fingerprints, then just 1% of buckets are expected to have loads greater than two bucket levels. To accommodate these cases, we can statically allocate memory for $J_3 = 0.01B = 10$ third-level buckets. Suppose we dimension these third-level buckets to provide $Z_3 = 54$ additional fingerprints. Then, for buckets that have been extended to the third-level, $Z = Z_1 + Z_2 + Z_3 = 64 + 10 + 54 = 128$ fingerprints are available. Third-level buckets will also provide an additional $Z_3 = 54$ bits for extending I_H as well. For the parameters shown in Table III-C, as well as in practice, three bucket levels are sufficient, and each bucket level can be dimensioned differently.

Once a bucket has been extended, access to the extended fingerprint array A requires only a small modification. For example, suppose we want to access the fingerprint array location $A[Z_1 + 1]$. Then we simply just use the memory pointer as an offset in computing the memory location.

IV. TAIL BOUND ANALYSIS

In this section, we establish a strict tail bound to bound the probability P_o that the bucket extensions for the overflowed buckets are insufficient.

Let random variable $X_i^{(n)}, 1 \leq i \leq B$, denote the number of fingerprints inserted in the i -th bucket. Let O_2 denote the total number of buckets overflowed into the 2^{nd} level bucket extension array, i.e. $O_2 \equiv \sum_{i=1}^B \mathbf{1}_{\{X_i^{(n)} > Z_1\}}$. We want to bound the probability that O_2 exceeds the number of 2^{nd} level bucket extensions, i.e. $Pr[O_2 > J_2]$.

In general, we want to bound $Pr[O_l > J_l], l = 2, 3, 4$. Let $W_1 = Z_1, W_2 = Z_1 + Z_2$ and $W_3 = Z_1 + Z_2 + Z_3$. Then $O_l = \sum_{i=1}^B \mathbf{1}_{\{X_i^{(n)} > W_{l-1}\}}$. Here we define $J_4 \equiv 0$, and $Pr[O_4 > J_4]$ captures the event that the buckets expand out of the 3^{rd} extension. So P_o is bound by

$$Pr[P_o] \leq \sum_{l=2}^4 Pr[O_l > J_l]$$

In the following we show how to bound $Pr[O_l > J_l]$, and we will use O, W, J instead of O_l, W_l, J_l .

For an analogy, the statistical model for hashing n elements into B buckets is the same as the classical balls-and-bins problem: n balls are thrown independently and uniformly at random into B bins. Then random variable $X_i^{(n)}$ could be translated as the number of balls ended up in the i -th bin after n balls are thrown.

Therefore, $(X_1^{(n)}, \dots, X_B^{(n)})$ follows the multinomial distribution. The marginal distribution for any $X_i^{(n)}$ is

$\text{Binomial}(n, \frac{1}{B})$, where $\text{Binomial}(N, P)$ denotes the binomial distribution resulting from N trials with success probability P . Since n is large and $\frac{1}{B}$ is small, $X_i^{(n)}$ can be approximated by $\text{Poisson}(\frac{n}{B})$, where $\text{Poisson}(\lambda)$ denotes the Poisson distribution with parameter λ .

A naive approach to calculating the probability $\Pr[O > J]$ is to regard $X_i^{(n)}$ as mutually independent. In this case, the sum of the indicator of the events $X_i^{(n)} > Z$ would be under binomial distribution and hence could be easily bounded. However, those random variable $X_i^{(n)}$ are not totally independent since they are conditioned by $\sum_{i=1}^B X_i^{(n)} = n$.

Fortunately, we have the following theorem to decouple the weak correlation among $(X_1^{(n)}, \dots, X_B^{(n)})$ and bound our targeted probability. The theorem is derived from [12, Theorem 5.10] using stochastic ordering, the proof of which is presented in Appendix.

Theorem 1: Let $(X_1^{(n)}, \dots, X_m^{(n)})$ follow the multinomial distribution of throwing n balls into m bins. Let $(Y_1^{(n)}, \dots, Y_m^{(n)})$ be independent and identically distributed random variables with distribution $\text{Poisson}(\frac{n}{m})$. Let $f(x_1, \dots, x_m)$ be a nonnegative function which is increasing in each argument separately. Then

$$E[f(X_1^{(n)}, \dots, X_m^{(n)})] \leq 2E[f(Y_1^{(n)}, \dots, Y_m^{(n)})]$$

Considering the following function

$$f(x_1, \dots, x_B) = 1_{\{(\sum_{i=1}^B 1_{x_i > w}) > J\}}$$

The targeted probability we want to bound, $\Pr[O > J]$, could be regarded as the expectation of $f(X_1^{(n)}, \dots, X_B^{(n)})$, i.e.

$$\Pr[O > J] = E[f(X_1^{(n)}, \dots, X_B^{(n)})].$$

$f(x_1, \dots, x_B)$ is obviously an increasing function. Therefore $E[f(X_1, \dots, X_B)] \leq 2E[f(Y_1, \dots, Y_B)]$, thanks to Theorem 1.

Y_i 's are distributed as $\text{Poisson}(\frac{n}{B})$. Therefore $\Pr[Y_i > W] = \text{Poissontail}(\frac{n}{B}, W)$, where $\text{Poissontail}(\lambda, K)$ denotes the tail probability $\Pr[Y > K]$ where Y has distribution $\text{Poisson}(\lambda)$.

$E[f(Y_1, \dots, Y_B)]$ denotes the probability that more than J of the events $Y_i > W$ happen. Since these events are mutually independent, the probability is $\text{Binotail}(B, \text{Poissontail}(\frac{n}{B}, W), J)$, where $\text{Binotail}(N, P, K)$ denotes the tail probability $\Pr[Z > K]$ where Z has distribution $\text{Binomial}(N, P)$.

So we get

$$\Pr[O > J] \leq 2\text{Binotail}(B, \text{Poissontail}(\frac{n}{B}, W), J). \quad (1)$$

$\Pr[O_4 > J_4]$ is a special case since $J_4 = 0$. We do not need theorem 1, and we have a better bound

TABLE II
REPRESENTATIVE RESULTS FOR RANK-INDEXED HASHING UNDER
VARIOUS PARAMETER SETTINGS ($P_o = 10^{-10}$, $n = 10^5$).

ϵ	λ	R	L	Z_1	Z_2	Z_3	J_2/B	J_3/B
1%	0.64	6 bits	60	45	8	45	17.9%	2.7%
0.1%	0.92	10 bits	64	63	17	50	36.0%	1.7%
0.01%	0.86	13 bits	61	59	13	48	23.3%	1.8%

$$\begin{aligned} \Pr[O_4 > 0] &= \Pr[(\max_i X_i^{(n)}) > W_3] \\ &\leq \sum_{i=1}^B \Pr[X_i^{(n)} > W_3] \\ &= B \times \text{Binotail}(n, \frac{1}{B}, W_3) \end{aligned}$$

V. EVALUATION

In this section, we present a set of numerical results computed from the tail bound theorems that are derived in Section IV and the appendix. For any targeted false positive probability ϵ , we apply the tail bound theorems to derive optimal configurations under different constraints.

One constraint we impose is on the parameter L . Ideally, larger buckets would lead to better statistical multiplexing and better space savings. However, we constrain L to be at most 64 to ensure that the rank and popcount operations described in Section III can be directly implemented using hardware-optimized 64-bit instructions that are readily available in modern microprocessors and network processors. For example, current generations of 64-bit x86 processors support such operations very efficiently [1], [2].

In Table II, we present representative sizing results for rank-indexed hashing under different false positive probabilities. These sizing results assume $n = 100,000$ elements and a probability of $P_o = 10^{-10}$ that the bucket sizing is not enough to store new fingerprints. In particular, the results presented are for false positive probabilities of $\epsilon = 1\%$, 0.1% , and 0.01% . Given the different parameter settings, we apply our tail bounds to optimize the configurations to minimize storage cost. The derived configurations are in terms of the load factor λ , the number of hash chain locations per bucket L , and the number of allocated entries in the bucket and bucket extensions Z_1, Z_2, Z_3 .

For a configuration of these parameters, the amount of memory required is determined as follows:

$$\begin{aligned} \mathcal{S}_1 &= (L + Z_1) + Z_1 r + (1 + \lceil \log J_2 \rceil) \\ \mathcal{S}_2 &= 1 + Z_2 + Z_2 r + (1 + \lceil \log J_3 \rceil) \\ \mathcal{S}_3 &= 1 + Z_3 + Z_3 r \\ \mathcal{S} &= B\mathcal{S}_1 + J_2\mathcal{S}_2 + J_3\mathcal{S}_3 \end{aligned}$$

Here, \mathcal{S}_1 is the storage requirement of one normal (first-level) bucket. The three additive components correspond to the bitmap index, the fingerprints, and the pointer to bucket extensions. (J_2 is the number of pre-allocated second-level bucket extensions). \mathcal{S}_2 and \mathcal{S}_3 are the size of the second-level and third-level bucket extensions. The one bit is for indicating occupancy. Then the total memory cost \mathcal{S} would be the sum of the storage requirements for the B normal buckets, J_2 second-level bucket extensions and J_3 third-level bucket extensions. Then the amortized storage cost per element can be computed as $\frac{\mathcal{S}}{n}$.

In Table III and Table IV, we compare results for the standard Bloom filter function [3] and for the counting Bloom filter function [10]. In addition to comparing with the standard Bloom filter constructions, we also compare with a recently proposed fingerprint hash table construction called d -left hashing [5], [4]. The standard Bloom filter functionality results are shown in Table III. In comparison to the d -left hashing construction, our rank-indexed hashing results outperform it in storage cost by 23.2% to 29.5%. In comparison to the standard Bloom filter construction, our rank-indexed hashing construction is able to outperform a standard Bloom filter construction in storage cost for false positive probabilities at or below just 0.1%. This is significant since a standard Bloom filter construction is widely regarded as a very space-efficient data structure for approximate membership query problems, and it is often desirable to have a false positive probability smaller than 0.1% in many applications.

Table IV present the counting Bloom filter functionality results. In comparison to a standard counting Bloom filter construction, our rank-indexed hashing construction is able to outperform in storage cost by a factor of three for a false positive probability of just 1%, and it is able to outperform the d -left hashing construction 22% to 27% at the same false positive probabilities.

In table V, we present another advantage of Rank-Indexed Bloom Filter, which could save more space after “compression”. When used for transferring, there is no need to send the vacant entries at the end of the higher index array and the fingerprint array in each bucket. After removing all these vacant entries, the trimmed size is only $S_p = 1/\lambda + (r + 1)$ bits per item. It is sufficient to transfer these trimmed buckets without any additional information, since the boundary of each array could be determined by counting the index bitmaps. The trimmed size is smaller than the optimized standard Bloom Filter size. To use Compressed Bloom Filter to achieve the same compression effect, much larger original Bloom Filter size will be needed. In table V, we could see that our Rank-Indexed scheme could easily achieve very compact trimmed size, while a much larger original array is needed for a Compressed Bloom Filter to achieve the same compression

TABLE III
COMPARISONS OF STORAGE COST (IN BITS) PER ELEMENT TO ACHIEVE THE SAME FALSE POSITIVE PROBABILITY ϵ FOR THE STANDARD BLOOM FILTER FUNCTION.

ϵ	standard	d -left	Rank	Comparison	
				vs. standard	vs. d -left
1%	9.6	15.0	10.6	+10.1%	-29.5%
0.1%	14.6	18.0	14.4	-1.1%	-26.3%
0.01%	19.1	22.2	18.2	-4.4%	-23.2%

TABLE IV
COMPARISONS OF STORAGE COST (IN BITS) PER ELEMENT TO ACHIEVE THE SAME FALSE POSITIVE PROBABILITY ϵ FOR THE COUNTING BLOOM FILTER (CBF) FUNCTION.

ϵ	standard CBF	d -left CBF	Rank CBF	Comparison	
				vs. standard	vs. d -left
1%	38.3	17.6	13.0	-66%	-27%
0.1%	58.4	22.3	16.8	-71%	-24%
0.01%	76.3	26.4	20.6	-73%	-22%

ratio.

Since d -left Bloom Filter is also fingerprint hash-table based, it could also be “compressed” similarly. However, its compression effect is much weaker, as also presented in table V.

Finally, storage cost comparisons are presented in Figure 5 and Figure 6 for false positive probabilities ranging from 10^{-1} to 10^{-7} . As shown in these plots, the proposed rank-indexing approach performs very competitively over this entire range.

VI. CONCLUSION

In this paper, we have described a new fingerprint hash table construction that can achieve the same functionalities as Bloom filters, counting Bloom filters, and other variants. The construction is based on a new method called Rank-Indexed Hashing that can achieve very compact representations. We have provided analysis and numerical evaluations

TABLE V
COMPARISON OF TRANSFERRING COST (IN BITS) PER ELEMENT AGAINST COMPRESSED BLOOM FILTER AND d -LEFT BLOOM FILTER

ϵ	Scheme	Packed Size	Uncompressed Compression BF ¹	Compression ratio ²
1%	Rank	8.6	30	89%
1%	d -left	11.7	— ³	122%
0.1%	Rank	12.1	150	83%
0.1%	d -left	15.2	— ³	104%
0.01%	Rank	15.2	1.3×10^3	79%
0.01%	d -left	18.3	27	96%

¹ “Uncompressed Compression BF” is the size of the uncompressed original array of a Compressed Bloom Filter to achieve the same compression ratio.

² “Compression ratio” is the ratio of the trimmed size of a Rank-Indexed or d -left scheme vs. the size of an optimized standard Bloom Filter, under the same false positive rate.

³ Omitted since the trimmed size is still larger than the size of a corresponding optimized standard Bloom Filter under the same false positive rate.

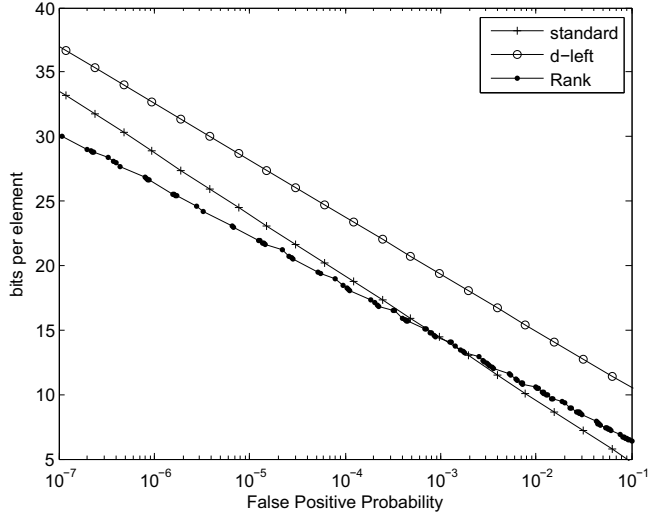


Fig. 5. Graphical plot of storage cost (in bits) per element to achieve the different false positive probabilities ϵ for the standard Bloom filter function.

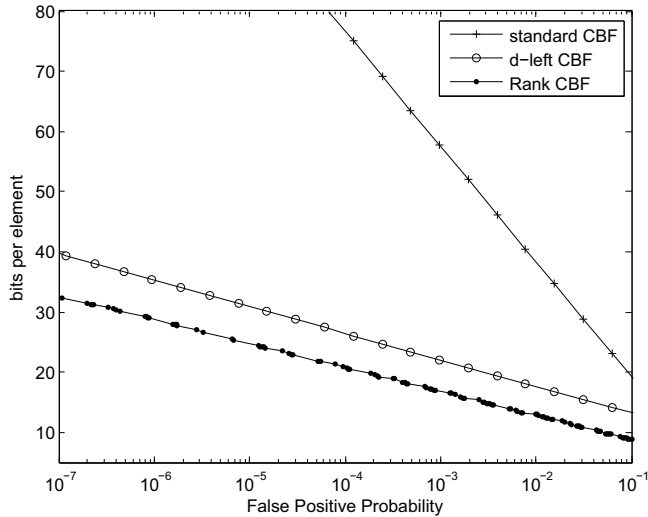


Fig. 6. Graphical plot of storage cost (in bits) per element to achieve the different false positive probabilities ϵ for the counting Bloom filter (CBF) function.

to show the storage performance of the proposed approach. In particular, a rank-indexed hashing construction that offers the same functionality as a counting Bloom filter can be achieved with a factor of three or more in space savings even for a false positive probability of just 1%. Even for a basic Bloom filter function that only supports membership queries, a rank-indexed hashing construction requires less space for a false positive probability as high as 0.1%, which is significant since a standard Bloom filter construction is widely regarded as extremely space-efficient for approximate membership problems.

REFERENCES

- [1] Intel 64 and IA-32 architectures software developer's manual, volume 2B, November 2007. Available at <ftp://download.intel.com/technology/architecture/new-instructions-paper.pdf>.
- [2] Software optimization guide for AMD family 10h processors, December 2007. Available at http://www.amd.com/us-en/assets/content_type/white_papers_and_tech_docs/40546.pdf.
- [3] Burton H. Bloom. Space/time trade-offs in hash coding with allowable errors. *CACM*, 13(7):422–426, 1970.
- [4] Flavio Bonomi, Michael Mitzenmacher, Rina Panigrahy, Sushil Singh, and George Varghese. Beyond bloom filters: from approximate membership checks to approximate state machines. In *SIGCOMM*, pages 315–326, 2006.
- [5] Flavio Bonomi, Michael Mitzenmacher, Rina Panigrahy, Sushil Singh, and George Varghese. An improved construction for counting bloom filters. In *ESA*, pages 684–695, 2006.
- [6] Andrei Z. Broder and Michael Mitzenmacher. Network applications of bloom filters: A survey. *Internet Mathematics*, 1(4), 2003.
- [7] Bernard Chazelle, Joe Kilian, Ronitt Rubinfeld, and Ayellet Tal. The bloomier filter: an efficient data structure for static support lookup tables. In *SODA*, pages 30–39, 2004.
- [8] Saar Cohen and Yossi Matias. Spectral bloom filters. In *SIGMOD Conference*, pages 241–252, 2003.
- [9] Mikael Degermark, Andrej Brodnik, Svante Carlsson, and Stephen Pink. Small forwarding tables for fast routing lookups. In *SIGCOMM*, pages 3–14, 1997.
- [10] Li Fan, Pei Cao, Jussara M. Almeida, and Andrei Z. Broder. Summary cache: a scalable wide-area web cache sharing protocol. *IEEE/ACM Trans. Netw.*, 8(3):281–293, 2000.
- [11] M. Mitzenmacher. Compressed bloom filters. *IEEE/ACM Transactions on Networking*, (5):613–620, October 2002.
- [12] Michael Mitzenmacher and Eli Upfal. *Probability and Computing: Randomized Algorithms and Probabilistic Analysis*. Cambridge University Press, 2005.
- [13] Alfred Müller and Dietrich Stoyan. *Comparison Methods for Stochastic Models and Risks*. Wiley, 2002.
- [14] Anna Pagh, Rasmus Pagh, and S. Srinivasa Rao. An optimal bloom filter replacement. In *SODA*, pages 823–829, 2005.
- [15] Nathan Tuck, Timothy Sherwood, Brad Calder, and George Varghese. Deterministic memory-efficient string matching algorithms for intrusion detection. In *IEEE INFOCOM*, 2004.
- [16] Berthold Vöcking. How asymmetry helps load balancing. In *FOCS*, pages 131–141, 1999.

APPENDIX

Theorem 1 is derived from Theorem 5.10 in [12], which we quote below.

Lemma 1: Let $(X_1^{(n)}, \dots, X_m^{(n)})$ and $(Y_1^{(n)}, \dots, Y_m^{(n)})$ be the same as defined in Theorem 1. Let $f(x_1, \dots, x_m)$ be a nonnegative function such that $E[f(X_1^{(n)}, \dots, X_m^{(n)})]$ is monotonically increasing in n . Then

$$E[f(X_1^{(n)}, \dots, X_m^{(n)})] \leq 2E[f(Y_1^{(n)}, \dots, Y_m^{(n)})]$$

In practice, we are concerned with the probability of some event A , whose indicator random variable is $f(X_1^{(n)}, \dots, X_m^{(n)})$, where n is some parameter. So $Pr[A] = E[f(X_1^{(n)}, \dots, X_m^{(n)})]$. In most cases it is usually intuitive to say that $Pr[A]$ increases as n increases, for example when A is some overflowing event, but it may not be trivial to prove so. On the other hand, it is usually trivial to see that f is an

increasing function. Therefore we introduced the easier-to-use Theorem 1, and we will prove it using stochastic ordering.

Stochastic ordering is a way to compare two random variables. Random variable X is stochastically less than or equal to random variable Y , written $X \leq_{st} Y$, iff $E\phi(X) \leq E\phi(Y)$ for all increasing functions ϕ such that the expectations exists. An equivalent definition of $X \leq_{st} Y$ is that $Pr[X > t] \leq Pr[Y > t]$, $-\infty < t < \infty$. The definition involving increasing functions also applies to random vectors $X = (X_1, \dots, X_h)$ and $Y = (Y_1, \dots, Y_h)$: $X \leq_{st} Y$ iff $E\phi(X) \leq E\phi(Y)$ for all increasing functions ϕ such that the expectations exists. Here ϕ is increasing means that it is increasing in each argument separately with other arguments being fixed. This is equivalent to $\phi(X) \leq_{st} \phi(Y)$. Note this definition is a much stronger condition than $Pr[X_1 > t_1, \dots, X_h > t_h] \leq Pr[Y_1 > t_1, \dots, Y_h > t_h]$ for all $t = (t_1, \dots, t_h) \in \mathcal{R}^n$.

Now we state without proof a fact that will be used to prove Proposition 2. Its proof can be found in all books that deal with stochastic ordering [13].

Proposition 1: Let X and Y be two random variables (or vectors). $X \leq_{st} Y$ iff there exists X' and Y' such that $\mu(X') = \mu(X)$, $\mu(Y') = \mu(Y)$, and $Pr[X' \leq Y'] = 1$. Here $\mu(X)$ means the distribution for X .

Now we are ready to prove the following proposition.

Proposition 2: Let $(X_1^{(n)}, X_2^{(n)}, \dots, X_m^{(n)})$ be the same as defined in Theorem 1. For any $0 \leq n < n'$, we have $(X_1^{(n)}, X_2^{(n)}, \dots, X_m^{(n)}) \leq_{st} (X_1^{(n')}, X_2^{(n')}, \dots, X_m^{(n')})$.

Proof: It suffices to prove it for $n' = n + 1$. Our idea is to find random variables Z and W such that Z has the same distribution as $(X_1^{(n)}, X_2^{(n)}, \dots, X_m^{(n)})$, W has the same distribution as $(X_1^{(n+1)}, X_2^{(n+1)}, \dots, X_m^{(n+1)})$, and $Pr[Z \leq W] = 1$. We will use the probability model that is generated by the “throwing $n+1$ balls into m bins one-by-one” random process. Now given any outcome ω in the probability space Ω , let $Z(\omega) = (Z_1(\omega), Z_2(\omega), \dots, Z_m(\omega))$, where $Z_j(\omega)$ is the number of balls in the j th bucket after we throw n balls into these m bins one by one. Now with all these n balls there, we throw the $(n+1)$ th ball uniformly randomly into one of the bins. We define $W(\omega)$ as $(W_1(\omega), W_2(\omega), \dots, W_m(\omega))$, where $W_j(\omega)$ is the number of balls in the j th bin after we throw in the $(n+1)$ th ball. Clearly we have $Z(\omega) \leq W(\omega)$ for any $\omega \in \Omega$ and therefore $Pr[Z \leq W] = 1$. Finally, we know from the property of the “throwing $n+1$ balls into m bin one-by-one” random process that Z and W have the same distribution as $(X_1^{(n)}, X_2^{(n)}, \dots, X_m^{(n)})$ and $(X_1^{(n+1)}, X_2^{(n+1)}, \dots, X_m^{(n+1)})$ respectively. ■

Now we are ready to prove Theorem 1.

Proof of Theorem 1: Let $f(x_1, \dots, x_m)$ be an increasing function. For any $0 \leq n < n'$, we have $(X_1^{(n)}, X_2^{(n)}, \dots, X_m^{(n)}) \leq_{st} (X_1^{(n')}, X_2^{(n')}, \dots, X_m^{(n')})$ by Proposition 2. By definition of stochastic ordering, we have

$E[f(X_1^{(n)}, \dots, X_m^{(n)})] \leq E[f(X_1^{(n')}, \dots, X_m^{(n')})]$. Therefore $E[f(X_1^{(n)}, \dots, X_m^{(n)})]$ is monotonically increasing in n , and the theorem follows from Lemma 1. ■